MPRI, Typage

Didier Rémy
(With course material from François Pottier)

November 06, 2012

Plan of the course

Introduction

Simply-typed λ-calculus

Polymorphism and System F

Type reconstruction

Type reconstruction

Contents

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Logical versus algorithmic properties

We have viewed a type system as a 3-place *predicate* over a type environment, a term, and a type.

So far, we have been concerned with *logical* properties of the type system, namely subject reduction and progress.

However, one should also study its *algorithmic* properties: is it decidable whether a term is well-typed?

Logical versus algorithmic properties

We have seen three different type systems, simply-typed λ-calculus, ML, and System F, of increasing expressiveness.

In each case, we have presented an explicitly-typed and an implicitly-typed version of the language and show a close correspondence between the two views, thanks to a type-passing semantics.

We argued that the explicitly-typed version is often more convenient for studying the meta-theoretical properties of the language.

Which one should we used for checking well-typedness? That is, in which language should we write programs?

Checking type derivations

The typing judgment is *inductively defined*, so that, in order to prove that a particular instance holds, one exhibits a *type derivation*.

A type derivation is essentially a version of the program where *every* node is annotated with a type.

Checking that a type derivation is correct is usually easy: it basically amounts to checking equalities between types.

However, type derivations are so verbose as to be intractable by humans! Requiring every node to be type-annotated is not practical.

Bottom-up type-checking

A more practical, and quite common, approach consists in requesting just enough annotations to allow types to be reconstructed in a *bottom-up* manner.

In other words, one seeks an *algorithmic reading* of the typing rules, where, in a judgment $\Gamma \vdash M : \tau$, the parameters Γ and M are *inputs*, while the parameter τ is an *output*.

Moreover, typing rules should be such that a type appearing as output in a conclusion should also appear as output in a premise or as input in the conclusion and input in the premises should be input of the conclusion or output of other premises.

This way, types need never be guessed, just looked up into the typing context, instantiated, or checked for equality.

Bottom-up type-checking

This is exactly the situation with explicitly-typed presentations of the typing rules.

This is also the traditional approach of Pascal, C, C++, Java, . . . : formal procedure parameters, as well as local variables, are assigned explicit types. The types of expressions are synthesized bottom-up.

Bottom-up type-checking

However, this implies a lot of redundancies:

- Parameters of *all* functions need to be annotated, even when their types are obvious from context.
- Let-expressions (when not primitive), recursive definitions, Injection into sum types need to be annotated.
- As the language grows, more and more constructs require type annotations, *e.g.* type applications and type abstractions.

Type annotations may quickly obfuscate the code and large explicitly-typed terms are so verbose that they become intractable by humans!

Hence, programming in the implicitly-typed version is more appealing.

Type inference

For simply-typed λ-calculus and ML, it turns out that this is possible: *whether a term is well-typed is decidable*, even when no type annotations are provided!

For System F, this is however undecidable. Since programming in explicitly-typed System F is not practically feasible, some amount of type reconstruction must still be done. Typically, the algorithm is incomplete, *i.e.* it rejects terms that are perhaps well-typed, but the user may always provide more annotations and at worse, the explicitly-typed version is never rejected if well-typed.

Contents

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
    - Constraint-based type inference for ML
    - Constraint solving by example
    - Type reconstruction
- Type annotations
    - Polymorphic recursion
    - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Type inference

The type inference algorithm for simply-typed λ-calculus, is due to Hindley [1969]. The idea behind the algorithm is simple.

Because simply-typed λ-calculus is a *syntax-directed* type system, an unannotated term determines an isomorphic *candidate type derivation*, where all types are unknown: they are distinct *type variables*.

For a candidate type derivation to become an actual, valid type derivation, every type variable must be instantiated with a type, subject to certain *equality constraints* on types.

For instance, at an application node, the type of the operator must match the domain type of the operator.

Type inference

Thus, type inference for the simply-typed λ-calculus decomposes into *constraint generation* followed by *constraint solving*.

Simple types are *first-order terms.* Thus, solving a collection of equations between simple types is *first-order unification.*

First-order unification can be performed incrementally in quasi-linear time, and admits particularly simple *solved forms*.

Constraints

At the interface between the constraint generation and constraint solving phases is the *constraint language*.

It is a *logic:* a *syntax*, equipped with an *interpretation* in a model.

Constraints

There are two syntactic categories: *types* and *constraints*.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \mathsf{F}\ \vec{\tau} \\ C & ::= & \mathit{true} \mid \mathit{false} \mid \tau = \tau \mid C \wedge C \mid \exists \alpha . C \end{array}$$

A type is either a *type variable* α or an arity-consistent application of a *type constructor* F.

(The type constructors are *unit*, ×, +, →, etc.)

An atomic constraint is truth, falsity, or an *equation* between types.

Compound constraints are built on top of atomic constraints via *conjunction* and *existential quantification* over type variables.

Constraints

Constraints are interpreted in the Herbrand universe, that is, in the set of *ground types:*

$$\mathbf{t} \quad ::= \quad \mathsf{F}\,\vec{\mathbf{t}}$$

Ground types contain no variables. The base case in this definition is when F has arity zero.

A *ground assignment* ϕ is a total mapping of type variables to ground types.

A ground assignment determines a total mapping of types to ground types.

Constraints

The interpretation of constraints takes the form of a judgment, $\phi \vdash C$, pronounced: *ϕ satisfies C*, or *ϕ is a solution of C.*

This judgment is inductively defined:

$$\phi \vdash \textit{true} \qquad \frac{\phi\tau_1 = \phi\tau_2}{\phi \vdash \tau_1 = \tau_2} \qquad \frac{\phi \vdash C_1 \qquad \phi \vdash C_2}{\phi \vdash C_1 \wedge C_2} \qquad \frac{\phi[\alpha \mapsto \mathbf{t}] \vdash C}{\phi \vdash \exists\alpha.C}$$

A constraint C is *satisfiable* if and only if there exists a ground assignment ϕ that satisfies C.

We write $C_1 \equiv C_2$ when C_1 and C_2 have the same solutions.

The problem: "given a constraint C, is C satisfiable?" is *first-order unification*.

Constraint generation

Type inference is reduced to constraint solving by defining a mapping of *candidate judgments* to constraints.

$$
\begin{array}{rcl}
\langle\!\langle \Gamma \vdash x : \tau \rangle\!\rangle & = & \Gamma(x) = \tau \\
\langle\!\langle \Gamma \vdash \lambda x.\, a : \tau \rangle\!\rangle & = & \exists \alpha_1 \alpha_2 . (\langle\!\langle \Gamma, x : \alpha_1 \vdash a : \alpha_2 \rangle\!\rangle \wedge \alpha_1 \rightarrow \alpha_2 = \tau) \\
 & & \text{if } \alpha_1, \alpha_2 \mathrel{\#} \Gamma, a, \tau \\
\langle\!\langle \Gamma \vdash a_1\ a_2 : \tau \rangle\!\rangle & = & \exists \alpha . (\langle\!\langle \Gamma \vdash a_1 : \alpha \rightarrow \tau \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash a_2 : \alpha \rangle\!\rangle) \\
 & & \text{if } \alpha \mathrel{\#} \Gamma, a_1, a_2, \tau
\end{array}
$$

Thanks to the use of existential quantification, the names that occur free in $\langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$ are a subset of those that occur free in Γ or τ.

This allows the freshness side-conditions to remain *local* – there is no need to informally require “globally fresh” type variables.

An example

Let us perform type inference for the closed term

$$\lambda fxy.\,(f\;x, f\;y)$$

The problem is to *construct* and *solve* the constraint

$$\langle\!\langle \varnothing \vdash \lambda fxy.\,(f\;x, f\;y) : \alpha_0 \rangle\!\rangle$$

It is possible (and, for a human, easier) to mix these tasks. A machine, however, could generate and solve in two successive phases.

Solving the constraint means to find all possible ground assignments for α_0 that satisfy the constraint.

Typically, this is done by transforming the constraint into successive equivalent constraints until some constraint that is obviously satisfiable and from which solutions may be directly read.

An example

$$\langle\!\langle \varnothing \vdash \lambda fxy.\,(f\ x, f\ y) : \alpha_0 \rangle\!\rangle$$

$$= \quad \exists \alpha_1 \alpha_2. \left(\begin{array}{l} \langle\!\langle f : \alpha_1 \vdash \lambda xy.\ \ldots : \alpha_2 \rangle\!\rangle \\ \alpha_1 \rightarrow \alpha_2 = \alpha_0 \end{array} \right)$$

$$= \quad \exists \alpha_1 \alpha_2. \left(\begin{array}{l} \exists \alpha_3 \alpha_4. \left(\begin{array}{l} \langle\!\langle f : \alpha_1 ; x : \alpha_3 \vdash \lambda y.\ \ldots : \alpha_4 \rangle\!\rangle \\ \alpha_3 \rightarrow \alpha_4 = \alpha_2 \end{array} \right) \\ \alpha_1 \rightarrow \alpha_2 = \alpha_0 \end{array} \right)$$

$$= \quad \exists \alpha_1 \alpha_2. \left(\begin{array}{l} \exists \alpha_3 \alpha_4. \left(\begin{array}{l} \exists \alpha_5 \alpha_6. \left(\begin{array}{l} \langle\!\langle f : \alpha_1 ; x : \alpha_3 ; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_5 \rightarrow \alpha_6 = \alpha_4 \end{array} \right) \\ \alpha_3 \rightarrow \alpha_4 = \alpha_2 \end{array} \right) \\ \alpha_1 \rightarrow \alpha_2 = \alpha_0 \end{array} \right)$$

We have performed constraint generation for the 3 λ-abstractions.

An example

$$\exists \alpha_1 \alpha_2 . \left(\begin{array}{l} \exists \alpha_3 \alpha_4 . \left(\begin{array}{l} \exists \alpha_5 \alpha_6 . \left(\begin{array}{l} \langle\!\langle f : \alpha_1 ; x : \alpha_3 ; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_5 \rightarrow \alpha_6 = \alpha_4 \end{array} \right) \\ \alpha_3 \rightarrow \alpha_4 = \alpha_2 \end{array} \right) \\ \alpha_1 \rightarrow \alpha_2 = \alpha_0 \end{array} \right)$$

$$\equiv \quad \exists \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6 . \left(\begin{array}{l} \langle\!\langle f : \alpha_1 ; x : \alpha_3 ; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_5 \rightarrow \alpha_6 = \alpha_4 \\ \alpha_3 \rightarrow \alpha_4 = \alpha_2 \\ \alpha_1 \rightarrow \alpha_2 = \alpha_0 \end{array} \right)$$

We have hoisted up several existential quantifiers:

$$(\exists \alpha . C_1) \wedge C_2 \equiv \exists \alpha . (C_1 \wedge C_2) \qquad \text{if } \alpha \mathrel{\#} C_2$$

An example

$$\exists \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6.\left(\begin{array}{l} \langle\!\langle f : \alpha_1; x : \alpha_3; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_5 \to \alpha_6 = \alpha_4 \\ \alpha_3 \to \alpha_4 = \alpha_2 \\ \alpha_1 \to \alpha_2 = \alpha_0 \end{array}\right)$$

$$\equiv \quad \exists \alpha_1\alpha_2\alpha_3\alpha_5\alpha_6.\left(\begin{array}{l} \langle\!\langle f : \alpha_1; x : \alpha_3; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_3 \to \alpha_5 \to \alpha_6 = \alpha_2 \\ \alpha_1 \to \alpha_2 = \alpha_0 \end{array}\right)$$

We have eliminated a type variable (α_4) with a defining equation:

$$\exists \alpha.(C \wedge \alpha = \tau) \equiv [\alpha \mapsto \tau]C \qquad \text{if } \alpha \# \tau$$

An example

$$\exists \alpha_1 \alpha_2 \alpha_3 \alpha_5 \alpha_6 . \left(\begin{array}{l} \langle\!\langle f : \alpha_1 ; x : \alpha_3 ; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_6 = \alpha_2 \\ \alpha_1 \rightarrow \alpha_2 = \alpha_0 \end{array} \right)$$

$$\equiv \quad \exists \alpha_1 \alpha_3 \alpha_5 \alpha_6 . \left(\begin{array}{l} \langle\!\langle f : \alpha_1 ; x : \alpha_3 ; y : \alpha_5 \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_1 \rightarrow \alpha_3 \rightarrow \alpha_5 \rightarrow \alpha_6 = \alpha_0 \end{array} \right)$$

We have again eliminated a type variable (α_2) with a defining equation.

In the following, let Γ stand for $(f : \alpha_1 ; x : \alpha_3 ; y : \alpha_5)$.

An example

$$\exists \alpha_1\alpha_3\alpha_5\alpha_6.\left(\begin{array}{l}\langle\!\langle \Gamma \vdash (f\ x, f\ y) : \alpha_6 \rangle\!\rangle \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_6 = \alpha_0\end{array}\right)$$

$$\equiv \quad \exists \alpha_1\alpha_3\alpha_5\alpha_6\alpha_7\alpha_8.\left(\begin{array}{l}\langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash f\ y : \alpha_8 \rangle\!\rangle \\ \alpha_7 \times \alpha_8 = \alpha_6 \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_6 = \alpha_0\end{array}\right)$$

$$\equiv \quad \exists \alpha_1\alpha_3\alpha_5\alpha_7\alpha_8.\left(\begin{array}{l}\langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash f\ y : \alpha_8 \rangle\!\rangle \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0\end{array}\right)$$

We have performed constraint generation for the pair, hoisted the resulting existential quantifiers, and eliminated a type variable (α_6).

Let us now focus on the first application...

An example

$$\begin{array}{rl} & \langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle \\ = & \exists \alpha_9 . \left(\begin{array}{l} \langle\!\langle \Gamma \vdash f : \alpha_9 \to \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash x : \alpha_9 \rangle\!\rangle \end{array} \right) \\ \\ = & \exists \alpha_9 . \left(\begin{array}{l} \alpha_1 = \alpha_9 \to \alpha_7 \\ \alpha_3 = \alpha_9 \end{array} \right) \\ \\ \equiv & \alpha_1 = \alpha_3 \to \alpha_7 \end{array}$$

We have performed constraint generation for the variables f and x, and eliminated a type variable (α_9).

Recall that Γ stands for $(f : \alpha_1 ; x : \alpha_3 ; y : \alpha_5)$.

Now, back to the big picture...

An example

$$\begin{array}{rl} & \exists\alpha_1\alpha_3\alpha_5\alpha_7\alpha_8.\left(\begin{array}{l} \langle\!\langle \Gamma \vdash f\ x : \alpha_7 \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash f\ y : \alpha_8 \rangle\!\rangle \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0 \end{array}\right) \\ \\ \equiv & \exists\alpha_1\alpha_3\alpha_5\alpha_7\alpha_8.\left(\begin{array}{l} \alpha_1 = \alpha_3 \to \alpha_7 \\ \langle\!\langle \Gamma \vdash f\ y : \alpha_8 \rangle\!\rangle \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0 \end{array}\right) \\ \\ \equiv & \exists\alpha_1\alpha_3\alpha_5\alpha_7\alpha_8.\left(\begin{array}{l} \alpha_1 = \alpha_3 \to \alpha_7 \\ \alpha_1 = \alpha_5 \to \alpha_8 \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0 \end{array}\right) \end{array}$$

We have applied the previous simplification under a context:

$$C_1 \equiv C_2 \Rightarrow \mathcal{R}[C_1] \equiv \mathcal{R}[C_2]$$

We have simplified the right-hand application analogously.

An example

$$\exists \alpha_1\alpha_3\alpha_5\alpha_7\alpha_8.\begin{pmatrix} \alpha_1 = \alpha_3 \to \alpha_7 \\ \alpha_1 = \alpha_5 \to \alpha_8 \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0 \end{pmatrix}$$

$$\equiv \quad \exists \alpha_1\alpha_3\alpha_5\alpha_7\alpha_8.\begin{pmatrix} \alpha_1 = \alpha_3 \to \alpha_7 \\ \alpha_3 = \alpha_5 \\ \alpha_7 = \alpha_8 \\ \alpha_1 \to \alpha_3 \to \alpha_5 \to \alpha_7 \times \alpha_8 = \alpha_0 \end{pmatrix}$$

$$\equiv \quad \exists \alpha_3\alpha_7.\left(\ (\alpha_3 \to \alpha_7) \to \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7 = \alpha_0\ \right)$$

We have applied transitivity at α_1, structural decomposition, and eliminated three type variables (α_1, α_5, α_8).

We have now reached a *solved form*.

An example

We have checked the following equivalence:

$$\begin{array}{rl} & \langle\!\langle \varnothing \vdash \lambda fxy.\,(f\ x, f\ y) : \alpha_0 \rangle\!\rangle \\ \equiv & \exists \alpha_3\alpha_7.\Big(\ (\alpha_3 \to \alpha_7) \to \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7 = \alpha_0\ \Big) \end{array}$$

The ground types of $\lambda fxy.\,(f\ x, f\ y)$ are all ground types of the form $(\mathbf{t}_3 \to \mathbf{t}_7) \to \mathbf{t}_3 \to \mathbf{t}_3 \to \mathbf{t}_7 \times \mathbf{t}_7$.

$(\alpha_3 \to \alpha_7) \to \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7$ is a *principal type* for $\lambda fxy.\,(f\ x, f\ y)$.

An example

Objective Caml implements a form of this type inference algorithm:

```
# fun f x y -> (f x, f y);;
- : ('a -> 'b) -> 'a -> 'a -> 'b * 'b = <fun>
```

This technique is used also by Standard ML and Haskell.

An example

In the simply-typed λ-calculus, type inference works just as well for *open* terms. Consider, for instance:

$$\lambda xy.(f\ x, f\ y)$$

This term has a free variable, namely f.

The type inference problem is to *construct* and *solve* the constraint

$$\langle\!\langle f : \alpha_1 \vdash \lambda xy.(f\ x, f\ y) : \alpha_2 \rangle\!\rangle$$

We have already done so... with only a slight difference: α_1 and α_2 are now free, so they cannot be eliminated.

An example

One can check the following equivalence:

$$\begin{array}{rl} & \langle\!\langle f : \alpha_1 \vdash \lambda xy.\,(f\ x, f\ y) : \alpha_2 \rangle\!\rangle \\ \equiv & \exists \alpha_3 \alpha_7. \left(\begin{array}{l} \alpha_3 \to \alpha_7 = \alpha_1 \\ \alpha_3 \to \alpha_3 \to \alpha_7 \times \alpha_7 = \alpha_2 \end{array} \right) \end{array}$$

In other words, the ground *typings* of $\lambda xy.\,(f\ x, f\ y)$ are all ground typings of the form:

$$((f : \mathbf{t}_3 \to \mathbf{t}_7), \mathbf{t}_3 \to \mathbf{t}_3 \to \mathbf{t}_7 \times \mathbf{t}_7)$$

Remember that a typing is a pair of an environment and a type.

Typings

Definition

(Γ, τ) is a *typing* of a if and only if $\mathrm{dom}(\Gamma) = \mathrm{fv}(a)$ and the judgment $\Gamma \vdash a : \tau$ is valid.

The type inference problem is to determine whether a term a admits a typing, and, if possible, to exhibit a description of the set of all of its typings.

Up to a change of universes, the problem reduces to finding the *ground typings* of a term. (For every type variable, introduce a nullary type constructor. Then, ground typings in the extended universe are in one-to-one correspondence with typings in the original universe.)

Constraint generation

Theorem (Soundness and completeness)

$\phi \vdash \langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$ *if and only if* $\phi\Gamma \vdash a : \phi\tau$.

Proof.

By structural induction over a. (Recommended exercise.) □

In other words, assuming $\mathrm{dom}(\Gamma) = \mathrm{fv}(a)$, ϕ satisfies the constraint $\langle\!\langle \Gamma \vdash a : \tau \rangle\!\rangle$ if and only if $(\phi\Gamma, \phi\tau)$ is a (ground) typing of a.

Constraint generation

Corollary

Let $\mathrm{fv}(a) = \{x_1, \ldots, x_n\}$*, where* $n \geq 0$*. Let* $\alpha_0, \ldots, \alpha_n$ *be pairwise distinct type variables. Then, the ground typings of* a *are described by*

$$((x_i : \phi\alpha_i)_{i \in 1..n}, \phi\alpha_0)$$

where ϕ *ranges over all solutions of* $\langle\!\langle (x_i : \alpha_i)_{i \in 1..n} \vdash a : \alpha_0 \rangle\!\rangle$*.*

Corollary

Let $\mathrm{fv}(a) = \varnothing$*. Then,* a *is well-typed if and only if* $\exists\alpha.\langle\!\langle \varnothing \vdash a : \alpha \rangle\!\rangle \equiv$ *true.*

Constraint solving

A constraint solving algorithm is typically presented as a (non-deterministic) system of *constraint rewriting rules*.

The system must enjoy the following properties:

- reduction is meaning-preserving: $C_1 \longrightarrow C_2$ implies $C_1 \equiv C_2$;
- reduction is terminating;
- every normal form is either "*false*" (literally) or satisfiable.

The normal forms are called *solved forms*.

First-order unification as constraint solving

Following Pottier and Rémy [2005, §10.6], we extend the syntax of constraints and replace ordinary binary equations with *multi-equations:*

$$U ::= \textit{true} \mid \textit{false} \mid \epsilon \mid U \wedge U \mid \exists \bar{\alpha}.U$$

A multi-equation ϵ is a multi-set of types. Its interpretation is:

$$\frac{\forall \tau \in \epsilon, \quad \phi\tau = \mathbf{t}}{\phi \vdash \epsilon}$$

That is, ϕ satisfies ϵ if and only if ϕ maps all members of ϵ to a single ground type.

First-order unification as constraint solving

$$\begin{array}{rcll}
(\exists\bar{\alpha}.U_1) \wedge U_2 & \longrightarrow & \exists\bar{\alpha}.(U_1 \wedge U_2) & \text{(extrusion)} \\
 & & \text{if } \bar{\alpha} \# U_2 & \\
\alpha = \epsilon \wedge \alpha = \epsilon' & \longrightarrow & \alpha = \epsilon = \epsilon' & \text{(fusion)} \\
\mathsf{F}\,\vec{\alpha} = \mathsf{F}\,\vec{\tau} = \epsilon & \longrightarrow & \vec{\alpha} = \vec{\tau} \wedge \mathsf{F}\,\vec{\alpha} = \epsilon & \text{(decomposition)} \\
\mathsf{F}\,\tau_1\,\ldots\,\tau_i\,\ldots\,\tau_n = \epsilon & \longrightarrow & \exists\alpha.(\alpha = \tau_i \wedge \mathsf{F}\,\tau_1\,\ldots\,\alpha\,\ldots\,\tau_n = \epsilon) & \text{(naming)} \\
 & & \text{if } \tau_i \text{ is not a variable} \wedge \alpha \# \tau_1, \ldots, \tau_n, \epsilon & \\
\mathsf{F}\,\vec{\tau} = \mathsf{F}'\,\vec{\tau}' = \epsilon & \longrightarrow & \mathit{false} & \text{(clash)} \\
 & & \text{if } \mathsf{F} \neq \mathsf{F}' & \\
U & \longrightarrow & \mathit{false} & \text{(occurs check)} \\
 & & \text{if } U \text{ is cyclic} & \\
\mathcal{U}[\mathit{false}] & \longrightarrow & \mathit{false} & \text{(error propag.)}
\end{array}$$

See [Pottier and Rémy, 2005, §10.6] for additional administrative rules.

The occurs check

α *dominates* β (with respect to U) iff U contains a multi-equation of the form $\mathsf{F}\,\tau_1 \ldots \beta \ldots \tau_n = \alpha = \ldots$

U is *cyclic* iff its domination relation is cyclic.

A cyclic constraint is unsatisfiable: indeed, if ϕ satisfies U and if α is a member of a cycle, then the ground type $\phi\alpha$ must be a strict subterm of itself, a contradiction.

Solved forms

A solved form is either *false* or $\exists\bar{\alpha}.U$, where U is a conjunction of multi-equations, every multi-equation contains at most one non-variable term, no two multi-equations share a variable, and the domination relation is acyclic.

Every solved form that is not *false* is satisfiable – indeed, a solution is easily constructed by well-founded recursion over the domination relation.

Implementation

Viewing a unification algorithm as a system of rewriting rules makes it easy to explain and reason about.

In practice, following Huet [1976], first-order unification is implemented on top of an efficient *union-find* data structure [Tarjan, 1975]. Its time complexity is quasi-linear.

Contents

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
    - Constraint-based type inference for ML
    - Constraint solving by example
    - Type reconstruction
- Type annotations
    - Polymorphic recursion
    - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Two presentations

Two presentations of type inference for Damas and Milner's type system are possible:

- one of Milner's classic algorithms [1978], $\mathcal{W}$ or $\mathcal{J}$; see Pottier's old course notes for details [Pottier, 2002, §3.3];
- a constraint-based presentation [Pottier and Rémy, 2005];

We favor the latter, but quickly review the former first.

Preliminaries

This algorithm expects a pair $\Gamma \vdash a$, produces a type τ, and uses two global variables, $\mathcal{V}$ and φ.

$\mathcal{V}$ is an infinite *fresh supply* of type variables:

$$\begin{array}{rcl} \textit{fresh} & = & \textit{do}\ \alpha \in \mathcal{V} \\ & & \textit{do}\ \mathcal{V} \leftarrow \mathcal{V} \setminus \{\alpha\} \\ & & \textit{return}\ \alpha \end{array}$$

φ is an *idempotent substitution* (of types for type variables), initially the identity.

The algorithm

Here is the algorithm in monadic style:

$$\begin{array}{rcl}
\mathcal{J}(\Gamma \vdash x) & = & \textit{let}\ \forall \alpha_1 \ldots \alpha_n.\tau = \Gamma(x) \\
 & & \textit{do}\ \alpha'_1, \ldots, \alpha'_n = \textit{fresh}, \ldots, \textit{fresh} \\
 & & \textit{return}\ [\alpha_i \mapsto \alpha'_i]_{i=1}^{n}(\tau)\ \textit{– take a fresh instance} \\
\mathcal{J}(\Gamma \vdash \lambda x.\, a_1) & = & \textit{do}\ \alpha = \textit{fresh} \\
 & & \textit{do}\ \tau_1 = \mathcal{J}(\Gamma; x : \alpha \vdash a_1) \\
 & & \textit{return}\ \alpha \to \tau_1\ \textit{– form an arrow type} \\
 & \ldots &
\end{array}$$

The algorithm

$$
\begin{array}{rcl}
 & \ldots & \\
\mathcal{J}(\Gamma \vdash a_1\ a_2) & = & \textit{do}\ \tau_1 = \mathcal{J}(\Gamma \vdash a_1) \\
 & & \textit{do}\ \tau_2 = \mathcal{J}(\Gamma \vdash a_2) \\
 & & \textit{do}\ \alpha = \textit{fresh} \\
 & & \textit{do}\ \varphi \leftarrow \mathrm{mgu}(\varphi(\tau_1) = \varphi(\tau_2 \rightarrow \alpha)) \circ \varphi \\
 & & \textit{return}\ \alpha\ \textit{– solve}\ \tau_1 = \tau_2 \rightarrow \alpha \\
\mathcal{J}(\Gamma \vdash \textit{let}\ x = a_1\ \textit{in}\ a_2) & = & \textit{do}\ \tau_1 = \mathcal{J}(\Gamma \vdash a_1) \\
 & & \textit{let}\ \sigma = \forall \setminus \mathrm{ftv}(\varphi(\Gamma)).\, \varphi(\tau_1)\ \textit{– generalize} \\
 & & \textit{return}\ \mathcal{J}(\Gamma; x : \sigma \vdash a_2)
\end{array}
$$

($\forall \setminus \bar{\alpha}.\, \tau$ quantifies over all type variables *other than* $\bar{\alpha}$.)

Some weaknesses

Algorithm $\mathcal{J}$ mixes *generation* and *solving* of equations. This lack of modularity leads to several weaknesses:

- proofs are more difficult;
- correctness and efficiency concerns are not clearly separated (if implemented literally, the algorithm is exponential in practice);
- adding new language constructs duplicates solving of equations;
- generalizations, such as the introduction of subtyping, are not easy.

Some weaknesses

Algorithm $\mathcal{J}$ works with *substitutions*, instead of *constraints*.

Substitutions are an approximation to solved forms for unification constraints.

Working with substitutions means using *most general unifiers*, *composition*, and *restriction*.

Working with constraints means using *equations*, *conjunction*, and *existential quantification*.

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Road map

Type inference for Damas and Milner's type system involves slightly more than first-order unification: there is also *generalization* and *instantiation* of type schemes.

So, the constraint language must be enriched.

We proceed in two steps:

- still within simply-typed λ-calculus, we present a variation of the constraint language;
- building on this variation, we introduce polymorphism.

A variation on constraints

How about letting the constraint solver, instead of the constraint generator, deal with *environment access* and *construction?*

Let's enrich the syntax of constraints:

$$C ::= \ldots \mid x = \tau \mid \textit{def } x : \tau \textit{ in } C$$

The idea is to interpret constraints in such a way as to validate the equivalence law:

$$\textit{def } x : \tau \textit{ in } C \equiv [x \mapsto \tau]C$$

The *def* form is an *explicit substitution* form.

A variation on constraints

More precisely, here is the new interpretation of constraints.

As before, a valuation ϕ maps type variables α to ground types.

In addition, a valuation ψ maps term variables x to ground types.

The satisfaction judgment now takes the form $\phi, \psi \vdash C$. The new rules of interest are:

$$\frac{\psi x = \phi\tau}{\phi, \psi \vdash x = \tau} \qquad \frac{\phi, \psi[x \mapsto \phi\tau] \vdash C}{\phi, \psi \vdash \textit{def}\ x : \tau\ \textit{in}\ C}$$

(All other rules are modified to just transport ψ.)

A variation on constraints

Constraint generation is now a mapping of an expression a and a type τ to a constraint $\langle\!\langle a : \tau \rangle\!\rangle$. There is no longer a need for the parameter Γ.

$$\begin{array}{rcl}
\langle\!\langle x : \tau \rangle\!\rangle & = & x = \tau \\
\langle\!\langle \lambda x.\, a : \tau \rangle\!\rangle & = & \exists \alpha_1 \alpha_2.(\textit{def}\ x : \alpha_1\ \textit{in}\ \langle\!\langle a : \alpha_2 \rangle\!\rangle \wedge \alpha_1 \rightarrow \alpha_2 = \tau) \\
 & & \text{if } \alpha_1, \alpha_2 \# a, \tau \\
\langle\!\langle a_1\ a_2 : \tau \rangle\!\rangle & = & \exists \alpha.(\langle\!\langle a_1 : \alpha \rightarrow \tau \rangle\!\rangle \wedge \langle\!\langle a_2 : \alpha \rangle\!\rangle) \\
 & & \text{if } \alpha \# a_1, a_2, \tau
\end{array}$$

Look ma, *no environments!*

A variation on constraints

Theorem (Soundness and completeness)

Assume $\mathrm{fv}(a) = \mathrm{dom}(\Gamma)$*. Then,* $\quad \phi, \phi\Gamma \vdash \langle\!\langle a : \tau \rangle\!\rangle$ *if and only if* $\phi\Gamma \vdash a : \phi\tau$*.*

Corollary

Assume $\mathrm{fv}(a) = \varnothing$*. Then,* a *is well-typed if and only if* $\exists\alpha.\langle\!\langle a : \alpha \rangle\!\rangle \equiv$ *true.*

Summary

This variation shows that there is *freedom* in the design of the constraint language, and that altering this design can *shift work* from the constraint generator to the constraint solver, or vice-versa.

Enriching constraints

To permit polymorphism, we must extend the syntax of constraints so that a variable x denotes not just a ground type, but a *set of ground types*.

However, these sets cannot be represented as type schemes $\forall\bar{\alpha}.\tau$, because constructing these simplified forms requires constraint solving.

To avoid mingling constraint generation and constraint solving, we use type schemes that incorporate constraints: *constrained type schemes*.

Enriching constraints

The syntax of *constraints* and of *constrained type schemes* is:

$$\begin{array}{rcl} C & ::= & \tau = \tau \mid C \wedge C \mid \exists \alpha . C \\ & \mid & x \preceq \tau \\ & \mid & \sigma \preceq \tau \\ & \mid & \textit{def } x : \sigma \textit{ in } C \\ \sigma & ::= & \forall \bar{\alpha}[C] . \tau \end{array}$$

$x \preceq \tau$ and $\sigma \preceq \tau$ are *instantiation constraints.* The latter form is introduced so as to make the syntax stable under substitutions of constrained type schemes for variables.

As before, *def* $x : \sigma$ *in* C is an *explicit substitution* form.

Enriching constraints

The idea is to interpret constraints in such a way as to validate the equivalence laws:

$$\textit{def } x : \sigma \textit{ in } C \equiv [x \mapsto \sigma]C$$

$$(\forall \bar{\alpha}[C].\tau) \leq \tau' \equiv \exists \bar{\alpha}.(C \wedge \tau = \tau') \quad \text{if } \bar{\alpha} \# \tau'$$

Using these laws, a closed constraint can be rewritten to a unification constraint (with a possibly exponential increase in size).

The new constructs do not add much expressive power. They add just enough to allow a stand-alone formulation of constraint generation.

Interpreting constraints

A type variable α still denotes a ground type.

A variable x now denotes a *set* of ground types.

Instantiation constraints are interpreted as *set membership.*

$$\frac{\phi\tau \in \psi x}{\phi, \psi \vdash x \preceq \tau} \qquad \frac{\phi\tau \in \binom{\phi}{\psi}\sigma}{\phi, \psi \vdash \sigma \preceq \tau} \qquad \frac{\phi, \psi[x \mapsto \binom{\phi}{\psi}\sigma] \vdash C}{\phi, \psi \vdash \textit{def } x : \sigma \textit{ in } C}$$

Interpreting constrained type schemes

The interpretation of $\forall\bar{\alpha}[C].\,\tau$ under ϕ and ψ is the set of all $\phi'\tau$, where ϕ and ϕ' coincide outside $\bar{\alpha}$ and where ϕ' and ψ satisfy C.

$$\binom{\phi}{\psi}(\forall\bar{\alpha}[C].\,\tau) = \{\phi'\tau \mid (\phi' \setminus \bar{\alpha} = \phi \setminus \bar{\alpha}) \wedge (\phi', \psi \vdash C)\}$$

For instance, the interpretation of $\forall\alpha[\exists\beta.\alpha = \beta \rightarrow \gamma].\,\alpha \rightarrow \alpha$ under ϕ and ψ is the set of all ground types of the form $(t \rightarrow \phi\gamma) \rightarrow (t \rightarrow \phi\gamma)$, where t ranges over ground types.

This is also the interpretation of $\forall\beta.\,(\beta \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma)$.

In fact, every constrained type scheme is equivalent to a standard type scheme.

If $\bar{\alpha}$ and C are empty, then $\binom{\phi}{\psi}\tau$ is $\phi\tau$.

A derived form

Notice that *def* $x : \sigma$ *in* C is equivalent to C whenever x does not appear free in C—whether or not of the constraints appearing in σ are solvable.

To enforce the constraints in σ to be solvable, we use a variant of the *def* construct:

$$\textit{let}\ x : \sigma\ \textit{in}\ C \quad \equiv \quad \textit{def}\ x : \sigma\ \textit{in}\ ((\exists \alpha . x \leq \alpha) \wedge C)$$

Expanding $\sigma \overset{\triangle}{=} \forall \bar{\alpha}[C_0].\tau$ and simplifying, an equivalent definition is:

$$\textit{let}\ x : \forall \bar{\alpha}[C_0].\tau\ \textit{in}\ C \quad \equiv \quad \exists \bar{\alpha}.C_0 \wedge \textit{def}\ x : \forall \bar{\alpha}[C_0].\tau\ \textit{in}\ C$$

It would also be equivalent to provide a direct interpretation of it:

$$\frac{\binom{\phi}{\psi}\sigma \neq \varnothing \qquad \phi, \psi[x \mapsto \binom{\phi}{\psi}\sigma] \vdash C}{\phi, \psi \vdash \textit{let}\ x : \sigma\ \textit{in}\ C}$$

Constraint generation

Constraint generation is now as follows:

$$\begin{array}{rcl}
\langle\!\langle x : \tau \rangle\!\rangle & = & x \preceq \tau \\
\langle\!\langle \lambda x.\, a : \tau \rangle\!\rangle & = & \exists \alpha_1 \alpha_2.(\textit{def}\ x : \alpha_1\ \textit{in}\ \langle\!\langle a : \alpha_2 \rangle\!\rangle \wedge \alpha_1 \to \alpha_2 = \tau) \\
 & & \text{if } \alpha_1, \alpha_2 \# a, \tau \\
\langle\!\langle a_1\ a_2 : \tau \rangle\!\rangle & = & \exists \alpha.(\langle\!\langle a_1 : \alpha \to \tau \rangle\!\rangle \wedge \langle\!\langle a_2 : \alpha \rangle\!\rangle) \\
 & & \text{if } \alpha \# a_1, a_2, \tau \\
\langle\!\langle \textit{let}\ x = a_1\ \textit{in}\ a_2 : \tau \rangle\!\rangle & = & \textit{let}\ x : (\!| a_1 |\!)\ \textit{in}\ \langle\!\langle a_2 : \tau \rangle\!\rangle \\
(\!| a |\!) & = & \forall \alpha \big[\langle\!\langle a : \alpha \rangle\!\rangle \big].\, \alpha
\end{array}$$

$(\!| a |\!)$ is a *principal constrained type scheme* for a: its intended interpretation is the set of all ground types that a admits.

Properties of constraint generation

Lemma

$$\exists\alpha.(\langle\!\langle a : \alpha \rangle\!\rangle \wedge \alpha = \tau) \quad \equiv \quad \langle\!\langle a : \tau \rangle\!\rangle \qquad \textit{if } \alpha \mathrel{\#} \tau.$$

Lemma

$$(\!| a |\!) \preceq \tau \quad \equiv \quad \langle\!\langle a : \tau \rangle\!\rangle.$$

Lemma

$$[x \mapsto (\!| a_1 |\!)]\langle\!\langle a_2 : \tau \rangle\!\rangle \quad \equiv \quad \langle\!\langle [x \mapsto a_1]a_2 : \tau \rangle\!\rangle.$$

Lemma

$$\langle\!\langle \textit{let } x = a_1 \textit{ in } a_2 : \tau \rangle\!\rangle \quad \equiv \quad \langle\!\langle a_1; [x \mapsto a_1]a_2 : \tau \rangle\!\rangle.$$

The constraint associated with a let construct is *equivalent* to the constraint associated with its let-normal form.

Complexity

Lemma

The size of $\langle\!\langle a : \tau \rangle\!\rangle$ *is linear in the sum of the sizes of* a *and* τ*.*

Constraint generation can be implemented in linear time and space.

Soundness and completeness

The statement keeps its previous form, ◂ back but Γ now contains Damas-Milner type schemes. Since Γ binds variables to type schemes, we define $\phi(\Gamma)$ as the point-wise mapping of $(\frac{\phi}{\varnothing})$ to Γ.

Theorem (Soundness and completeness)

Let $\mathrm{fv}(a) = \mathrm{dom}(\Gamma)$. *Then,* $\phi, \phi\Gamma \vdash \langle\!\langle a : \tau \rangle\!\rangle$ *if and only if* $\phi\Gamma \vdash a : \phi\tau$*.*

Summary

Note that

- constraint generation has *linear complexity;*
- constraint generation and constraint solving are *separate;*
- the constraint language remains *small* as the programming language grows.

This makes constraints suitable for use in an efficient and modular implementation.

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

An initial environment

Let Γ_0 stand for $\textit{assoc} : \forall \alpha\beta.\, \alpha \rightarrow \textit{list}\,(\alpha \times \beta) \rightarrow \beta$.

We take Γ_0 to be the *initial environment,* so that the constraints considered next are implicitly wrapped within the context $\textit{def}\ \Gamma_0\ \textit{in}\ []$.

A code fragment

Let a stand for the term

$$\begin{array}{l} \lambda x.\lambda l_1.\lambda l_2. \\ \quad \textit{let assocx} = \textit{assoc}\, x \textit{ in} \\ \quad (\textit{assocx}\ l_1, \textit{assocx}\ l_2) \end{array}$$

One anticipates that *assocx* receives a polymorphic type scheme, which is instantiated twice at different types...

The generated constraint

Let Γ stand for $x : \alpha_0; l_1 : \alpha_1; l_2 : \alpha_2$. Then, the constraint $\langle\!\langle a : \alpha \rangle\!\rangle$ is (with a few minor simplifications):

$$\exists \alpha_0\alpha_1\alpha_2\beta.\left(\begin{array}{l} \alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \mathit{def}\ \Gamma\ \mathit{in} \\ \qquad \mathit{let\ assocx} : \forall\gamma_1[\exists\gamma_2.\left(\begin{array}{l} \mathit{assoc} \le \gamma_2 \to \gamma_1 \\ x \le \gamma_2 \end{array}\right)].\gamma_1\ \mathit{in} \\ \qquad \exists\beta_1\beta_2.\left(\begin{array}{l} \beta = \beta_1 \times \beta_2 \\ \forall i \in \{1,2\},\ \ \exists\gamma_2.(\mathit{assocx} \le \gamma_2 \to \beta_i \wedge l_i \le \gamma_2) \end{array}\right) \end{array}\right)$$

Simplification

Constraint solving can be viewed as a *rewriting process* that exploits *equivalence laws*. Because equivalence is, by construction, a *congruence*, rewriting is permitted within an arbitrary context.

For instance, environment access is allowed by the law

$$\textit{let}\, x : \sigma \;\textit{in}\, \mathcal{R}[x \preceq \tau] \quad \equiv \quad \textit{let}\, x : \sigma \;\textit{in}\, \mathcal{R}[\sigma \preceq \tau]$$

where $\mathcal{R}$ is a context that does not bind x.

Simplification, continued

Thus, within the context *def* $\Gamma_0;\Gamma$ *in* $[]$, the constraint:

$$\left(\begin{array}{c} \mathit{assoc} \leq \gamma_2 \rightarrow \gamma_1 \\ x \leq \gamma_2 \end{array}\right)$$

is equivalent to:

$$\left(\begin{array}{c} \exists\alpha\beta.(\alpha \rightarrow \mathit{list}\,(\alpha \times \beta) \rightarrow \beta = \gamma_2 \rightarrow \gamma_1) \\ \alpha_0 = \gamma_2 \end{array}\right)$$

Simplification, continued

By first-order unification, the constraint:

$$\exists\gamma_2.(\exists\alpha\beta.(\alpha \to \mathit{list}\,(\alpha \times \beta) \to \beta = \gamma_2 \to \gamma_1) \wedge \alpha_0 = \gamma_2)$$

simplifies down successively to:

$$\exists\gamma_2.(\exists\alpha\beta.(\alpha = \gamma_2 \wedge \mathit{list}\,(\alpha \times \beta) \to \beta = \gamma_1) \wedge \alpha_0 = \gamma_2)$$

$$\exists\gamma_2.(\exists\beta.(\mathit{list}\,(\gamma_2 \times \beta) \to \beta = \gamma_1) \wedge \alpha_0 = \gamma_2)$$

$$\exists\beta.(\mathit{list}\,(\alpha_0 \times \beta) \to \beta = \gamma_1)$$

Simplification, continued

The constrained type scheme:

$$\forall\gamma_1[\exists\gamma_2.(\mathit{assoc} \le \gamma_2 \to \gamma_1 \wedge x \le \gamma_2)].\gamma_1$$

is thus equivalent to:

$$\forall\gamma_1[\exists\beta.(\mathit{list}\,(\alpha_0 \times \beta) \to \beta = \gamma_1)].\gamma_1$$

which can also be written:

$$\forall\gamma_1\beta[\mathit{list}\,(\alpha_0 \times \beta) \to \beta = \gamma_1].\gamma_1$$

$$\forall\beta.\mathit{list}\,(\alpha_0 \times \beta) \to \beta$$

Simplification, continued

The initial constraint has now been simplified down to:

$$\exists \alpha_0 \alpha_1 \alpha_2 \beta . \left(\begin{array}{l} \alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta \\ \textit{def } \Gamma \textit{ in} \\ \qquad \textit{let assocx} : \forall \beta . \textit{list}\ (\alpha_0 \times \beta) \rightarrow \beta \textit{ in} \\ \qquad \exists \beta_1 \beta_2 . \left(\begin{array}{l} \beta = \beta_1 \times \beta_2 \\ \forall i \quad \exists \gamma_2 . (\textit{assocx} \preceq \gamma_2 \rightarrow \beta_i \wedge l_i \preceq \gamma_2) \end{array} \right) \end{array} \right)$$

The simplification work spent on *assocx*'s type scheme was well worth the trouble, because we are now going to *duplicate* the simplified type scheme.

Simplification, continued

The sub-constraint:

$$\exists\gamma_2.(\mathit{assocx} \leq \gamma_2 \to \beta_i \wedge l_i \leq \gamma_2)$$

where $i \in \{1, 2\}$, is rewritten:

$$\exists\gamma_2.(\exists\beta.(\mathit{list}\,(\alpha_0 \times \beta) \to \beta = \gamma_2 \to \beta_i) \wedge \alpha_i = \gamma_2)$$

$$\exists\beta.(\mathit{list}\,(\alpha_0 \times \beta) \to \beta = \alpha_i \to \beta_i)$$

$$\exists\beta.(\mathit{list}\,(\alpha_0 \times \beta) = \alpha_i \wedge \beta = \beta_i)$$

$$\mathit{list}\,(\alpha_0 \times \beta_i) = \alpha_i$$

Simplification, continued

The initial constraint has now been simplified down to:

$$\exists\alpha_0\alpha_1\alpha_2\beta.\left(\begin{array}{l} \alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \mathit{def}\ \Gamma\ \mathit{in} \\ \quad \mathit{let}\ \mathit{assocx} : \forall\beta.\, \mathit{list}\,(\alpha_0 \times \beta) \to \beta\ \mathit{in} \\ \quad \exists\beta_1\beta_2.\left(\begin{array}{l} \beta = \beta_1 \times \beta_2 \\ \forall i \quad \mathit{list}\,(\alpha_0 \times \beta_i) = \alpha_i \end{array}\right) \end{array}\right)$$

Now, the context *def* Γ *in let assocx* : … *in* $[\,]$ can be dropped, because the constraint that it applies to contains no occurrences of x, l_1, l_2, or *assocx*.

Simplification, continued

The constraint becomes:

$$\exists \alpha_0 \alpha_1 \alpha_2 \beta . \begin{pmatrix} \alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \exists \beta_1 \beta_2 . \begin{pmatrix} \beta = \beta_1 \times \beta_2 \\ \forall i \quad \mathit{list}\,(\alpha_0 \times \beta_i) = \alpha_i \end{pmatrix} \end{pmatrix}$$

that is:

$$\exists \alpha_0 \alpha_1 \alpha_2 \beta \beta_1 \beta_2 . \begin{pmatrix} \alpha = \alpha_0 \to \alpha_1 \to \alpha_2 \to \beta \\ \beta = \beta_1 \times \beta_2 \\ \forall i \quad \mathit{list}\,(\alpha_0 \times \beta_i) = \alpha_i \end{pmatrix}$$

and, by eliminating a few auxiliary variables:

$$\exists \alpha_0 \beta_1 \beta_2 . (\alpha = \alpha_0 \to \mathit{list}\,(\alpha_0 \times \beta_1) \to \mathit{list}\,(\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2)$$

Simplification, the end

We have shown the following equivalence between constraints:

$$\begin{array}{rl} & \textit{def } \Gamma_0 \textit{ in } \langle\!\langle a : \alpha \rangle\!\rangle \\ \equiv & \exists \alpha_0 \beta_1 \beta_2 . (\alpha = \alpha_0 \to \textit{list}\,(\alpha_0 \times \beta_1) \to \textit{list}\,(\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2) \end{array}$$

That is, the *principal type scheme* of a relative to Γ_0 is

$$\begin{array}{rcl} (\!| a |\!) & = & \forall \alpha \big[\langle\!\langle a : \alpha \rangle\!\rangle \big] . \alpha \\ & = & \forall \alpha_0 \beta_1 \beta_2 . \;\; \alpha_0 \to \textit{list}\,(\alpha_0 \times \beta_1) \to \textit{list}\,(\alpha_0 \times \beta_2) \to \beta_1 \times \beta_2 \end{array}$$

Rewriting strategies

Again, constraint solving can be explained in terms of a *small-step rewrite system*.

Again, one checks that every step is meaning-preserving, that the system is normalizing, and that every normal form is either literally "*false*" or satisfiable.

Different constraint solving *strategies* lead to different behaviors in terms of complexity, error explanation, etc.

See ATTAPL for details on constraint solving [Pottier and Rémy, 2005]. See Jones [1999] for a different presentation of type inference, in the context of Haskell.

Rewriting strategies

In all reasonable strategies, the left-hand side of a let constraint is simplified *before* the let form is expanded away.

This corresponds, in Algorithm $\mathcal{J}$, to computing a principal type scheme before examining the right-hand side of a let construct.

Complexity

Type inference for ML is DEXPTIME-complete [Kfoury et al., 1990; Mairson, 1990], so any constraint solver has exponential complexity.

Nevertheless, under the hypotheses that *types have bounded size* and let forms have bounded left-nesting depth, constraints can be solved in linear time [McAllester, 2003].

This explains why ML type inference *works well in practice.*

An alternative presentation of constraint generation

Using principal contrained type schemes and the following equivalence

$$(\!| a |\!) \triangleq \forall \alpha \big[\langle\!\langle a : \alpha \rangle\!\rangle \big] . \, \alpha \qquad\qquad \langle\!\langle a : \tau \rangle\!\rangle \equiv (\!| a |\!) \le \tau$$

we call also present contraint generation as follows:

$$\begin{array}{rcl}
(\!| x |\!) & = & \forall \alpha [x \le \alpha] . \, \alpha \\
(\!| \lambda x . \, a |\!) & = & \forall \alpha_1 \alpha_2 \big[\mathit{def}\ x : \alpha_1\ \mathit{in}\ (\!| a |\!) \le \alpha_2 \big] . \, \alpha_1 \to \alpha_2 \\
 & & \text{if } \alpha_1, \alpha_2 \# a \\
(\!| a_1\ a_2 |\!) & = & \forall \alpha_1 \alpha_2 \big[(\!| a_1 |\!) \le \alpha_1 \to \alpha_2 \wedge (\!| a_2 |\!) \le \alpha_1 \big] . \, \alpha_2 \\
 & & \text{if } \alpha_1, \alpha_2 \# a_1, a_2 \\
(\!| \mathit{let}\ x = a_1\ \mathit{in}\ a_2 |\!) & = & \forall \alpha \big[\mathit{let}\ x : (\!| a_1 |\!)\ \mathit{in}\ (\!| a_2 |\!) \le \alpha \big] . \, \alpha
\end{array}$$

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Type reconstruction

Type inference should not just infer a principal type for an expression. It should also elaborate the implicitly-typed input term into an explicitly-typed one.

Notice that the elaborated term is not unique:

Type reconstruction

Type inference should not just infer a principal type for an expression. It should also elaborate the implicitly-typed input term into an explicitly-typed one.

Notice that the elaborated term is not unique:

- redundant type abstractions and type applications may be used.
- some non principal type schemes may sometimes be used for local let-bindings.

Type reconstruction

Type inference should not just infer a principal type for an expression. It should also elaborate the implicitly-typed input term into an explicitly-typed one.

Notice that the elaborated term is not unique:

- redundant type abstractions and type applications may be used.
- some non principal type schemes may sometimes be used for local let-bindings.

However, we may seek for a principal derivation in canonical form (as defined in the previous chapter, Damas and Milner's type system).

Type reconstruction Idea

To perform type reconstruction, it suffices to know the types of let bindings and of function parameters.

In constraints, it suffices to remember def and let-constraints and instantiation constraints $x \preceq \tau$: we may just not remove then during constraint resolution.

We also request that let-constraints be not extruded, so that the binding structure of let-constraints and the scopes of program variables remain as in the original constaint.

Type reconstruction Preserving the orginal

We modify equivalences used during constraint resolution, so as to preserve the original constraint—and mark it as resolved.

For instance, environment access becomes:

$$\textit{def}\ x:\sigma\ \textit{in}\ \mathcal{R}[x \le \tau] \quad \equiv \quad \textit{def}\ x:\sigma\ \textit{in}\ \mathcal{R}[x \le \tau \wedge \sigma \le \tau]$$

A binding constraint *def* $x:\sigma$ *in* C can be flagged as presolved when x only appears free in resolved constraints of C:

$$\textit{def}\ x:\sigma\ \textit{in}\ C \quad \equiv \quad \textit{def}\ x:\sigma\ \textit{in}\ C \qquad\qquad x \# (C \smallsetminus C)$$

A resolved form of a constraint C is an equivalent constraint with the same structure as C that is in solved form after dropping all resolved subconstraints.

Example

Let us reuse a defined above as

$$\lambda x.\,\lambda l_1.\,\lambda l_2.\,\mathit{let\ assocx} = \mathit{assoc}\ \ x\ \mathit{in}\ (\mathit{assocx}\ \ l_1, \mathit{assocx}\ \ l_2)$$

The principal type scheme $(\!|a|\!)$ is:

$$\forall\alpha\left[\begin{array}{l} \exists\alpha_0\alpha_1\alpha_2\beta. \\ \quad\left(\begin{array}{l} \alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta \\ \mathit{def}\ \Gamma\ \mathit{in} \\ \quad \mathit{let\ assocx} : \forall\gamma_1\,[\exists\gamma_2.\mathit{assoc} \le \gamma_2 \rightarrow \gamma_1 \wedge x \le \gamma_2].\,\gamma_1\ \mathit{in} \\ \quad \exists\beta_1\beta_2.\left(\begin{array}{l} \beta = \beta_1\times\beta_2 \\ \forall i \in \{1,2\},\ \ \exists\gamma_2.(\mathit{assocx} \le \gamma_2 \rightarrow \beta_i \wedge l_i \le \gamma_2)\end{array}\right)\end{array}\right)\end{array}\right].\,\alpha$$

where:

- Γ stands for $x : \alpha_0; l_1 : \alpha_1; l_2 : \alpha_2$, and the initial environment
- Γ_0 stands for $\mathit{assoc} : \forall\alpha\beta.\,\alpha \rightarrow \mathit{list}\ (\alpha\times\beta) \rightarrow \beta$.

Example

The inner *assocx* type scheme in context Γ can be simplified as follows:

$$\begin{array}{rl}
& \forall\gamma_1\left[\exists\gamma_2.\left(\ \mathit{assoc} \le \gamma_2 \to \gamma_1 \wedge x \le \gamma_2\ \right)\right].\gamma_1 \\
\equiv & \forall\gamma_1\left[\exists\gamma_2.\left(\ \mathit{assoc} \le \gamma_2 \to \gamma_1 \wedge x \le \gamma_2 \wedge \alpha_0 \le \gamma_2\ \right)\right].\gamma_1 \\
\equiv & \forall\gamma_1\left[\ \mathit{assoc} \le \alpha_0 \to \gamma_1 \wedge x \le \alpha_0\ \right].\gamma_1 \\
\equiv & \forall\gamma_1\left[\begin{array}{l} \mathit{assoc} \le \alpha_0 \to \gamma_1 \wedge x \le \alpha_0 \\ \forall\alpha\beta.\alpha \to \mathit{list}\,(\alpha\times\beta) \to \beta \le \alpha_0 \to \gamma_1 \end{array}\right].\gamma_1 \\
\equiv & \forall\gamma_1\left[\begin{array}{l} \mathit{assoc} \le \alpha_0 \to \gamma_1 \wedge x \le \alpha_0 \\ \exists\alpha\beta.(\alpha = \alpha_0 \wedge \mathit{list}\,(\alpha\times\beta) \to \beta = \gamma_1) \end{array}\right].\gamma_1 \\
\equiv & \forall\beta\left[\ \mathit{assoc} \le \alpha_0 \to \mathit{list}\,(\alpha_0\times\beta) \to \beta \wedge x \le \alpha_0\ \right].\mathit{list}\,(\alpha_0\times\beta) \to \beta
\end{array}$$

Example

Simplifying the remaining instantiations similarly $(\!| a |\!)$ is equivalent to:

$$\forall \alpha_0\beta_1\beta_2 \left[\begin{array}{l} \textit{def}\ \ \Gamma\ \textit{in} \\ \quad \textit{let assocx} : \forall\gamma\left[\begin{array}{l} \textit{assoc} \le \alpha_0 \to \textit{list}\,(\alpha_0\times\gamma)\to\gamma) \\ x \le \alpha_0\end{array}\right]. \\ \qquad\qquad \textit{list}\,(\alpha_0\times\gamma)\to\gamma \qquad\qquad \textit{in} \\ \quad \forall i\in\{1,2\},\ \left(\begin{array}{l} (\textit{assocx} \le \textit{list}\,(\alpha_0\times\beta_i)\to\beta_i) \\ l_i \le \textit{list}\,(\alpha_0\times\beta_i)))\end{array}\right)\end{array}\right].$$

$$\alpha_0 \to \textit{list}\,(\alpha_0\times\beta_1)\to\textit{list}\,(\alpha_0\times\beta_2)\to\beta_1\times\beta_2$$

Form which, we may read the elaboration of M:

$$\begin{array}{l} \Lambda\alpha_0\beta_1\beta_2.\lambda x\!:\!\alpha_0.\,\lambda l_1\!:\!\textit{list}\,(\alpha_0\times\beta_1).\,\lambda l_2\!:\!\textit{list}\,(\alpha_0\times\beta_2). \\ \quad \textit{let assocx} = \Lambda\gamma.\textit{assoc}\ \alpha_0\ \gamma\ \ x\ \textit{in} \\ \quad (\textit{assocx}\ \beta_1\ \ l_1, \textit{assocx}\ \beta_2\ \ l_2) \end{array}$$

Type abstrations are read from the principal type scheme.

Type applications are locally inferred from type instantiations.

Contents

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

On type annotations

Damas and Milner's type system has *principal types:* at least in the core language, no type information is required.

This is very lightweight, but a bit extreme: sometimes, it is useful to write types down, and use them as *machine-checked documentation.*

Syntax for type annotations

Let us, then, allow programmers to *annotate* a term with a type:

$$a ::= \dots \mid (a : \tau)$$

Typing and constraint generation are obvious:

$$\textsc{Annot}\quad \frac{\Gamma \vdash a : \tau}{\Gamma \vdash (a : \tau) : \tau} \qquad\qquad \langle\!\langle (a : \tau) : \tau' \rangle\!\rangle = \langle\!\langle a : \tau \rangle\!\rangle \wedge \tau = \tau'$$

Type annotations are *erased* prior to runtime, so the operational semantics is not affected.

(Erasure of type annotations preserves well-typedness.)

Type annotations are restrictive

The constraint $\langle\!\langle (a : \tau) : \tau' \rangle\!\rangle$ *implies* the constraint $\langle\!\langle a : \tau' \rangle\!\rangle$.

That is, in terms of type inference, *type annotations are restrictive:* they lead to a principal type that is less general, and possibly even to ill-typedness.

For instance, $\lambda x. x$ has principal type scheme $\forall \alpha. \alpha \rightarrow \alpha$, whereas $(\lambda x. x : \mathit{int} \rightarrow \mathit{int})$ has principal type scheme $\mathit{int} \rightarrow \mathit{int}$, and $(\lambda x. x : \mathit{int} \rightarrow \mathit{bool})$ is ill-typed.

Type variables within type annotations?

Does it make sense for a type annotation to contain a type variable, as in, say:

$$(\lambda x.\, x : \alpha \to \alpha)$$
$$(\lambda x.\, x + 1 : \alpha \to \alpha)$$
$$\textit{let } f = (\lambda x.\, x : \alpha \to \alpha) \textit{ in } (f\ 0, f\ \textit{true})$$

If so, what does it mean?

Short answer: it does not mean anything, because α is unbound. *"There is no such thing as a free variable"* (Alan Perlis).

How and where

Does it make sense for a type annotation to contain a type variable, as in, say:

$$(\lambda x.\, x : \alpha \to \alpha)$$
$$(\lambda x.\, x + 1 : \alpha \to \alpha)$$
$$\mathit{let}\ f = (\lambda x.\, x : \alpha \to \alpha)\ \mathit{in}\ (f\ 0, f\ \mathit{true})$$

If so, what does it mean?

A longer answer:

It is necessary to specify *how* and *where* type variables are bound.

Type variables within type annotations?

Does it make sense for a type annotation to contain a type variable, as in, say:

$$(\lambda x.\, x : \alpha \to \alpha)$$
$$(\lambda x.\, x + 1 : \alpha \to \alpha)$$
$$\textit{let } f = (\lambda x.\, x : \alpha \to \alpha) \textit{ in } (f\ 0, f\ \textit{true})$$

If so, what does it mean?

How is α bound?

If α is *existentially* bound, or *flexible,* then both $(\lambda x.\, x : \alpha \to \alpha)$ and $(\lambda x.\, x + 1 : \alpha \to \alpha)$ should be well-typed.

If it is *universally* bound, or *rigid,* only the former should be well-typed.

Where

Does it make sense for a type annotation to contain a type variable, as in, say:

$$(\lambda x.\, x : \alpha \rightarrow \alpha)$$
$$(\lambda x.\, x + 1 : \alpha \rightarrow \alpha)$$
$$\textit{let } f = (\lambda x.\, x : \alpha \rightarrow \alpha) \textit{ in } (f\ 0, f\ \textit{true})$$

If so, what does it mean?

Where is α bound?

If α is bound *within* the left-hand side of this “let” construct, then this code:

$$\textit{let } f = (\lambda x.\, x : \alpha \rightarrow \alpha) \textit{ in } (f\ 0, f\ \textit{true})$$

should be well-typed.

On the other hand, if α is bound *outside* this “let” form, then this code should be ill-typed, since no *single* ground value of α is suitable.

Binding type variables

Let's allow programmers to *explicitly bind* type variables:

$$a ::= \dots \mid \exists \bar{\alpha}.a \mid \forall \bar{\alpha}.a$$

It now makes sense for a type annotation $(a : \tau)$ to contain free type variables.

Terms a can now contain free type variables, so some side conditions have to be updated (e.g., $\bar{\alpha} \# \Gamma, a$ in $\textsc{Gen}$).

Binding type variables

The typing rules are as follows:

$$\textsc{Exists}\quad \dfrac{\Gamma \vdash [\vec{\alpha} \mapsto \vec{\tau}]a : \tau}{\Gamma \vdash \exists \bar{\alpha}.a : \tau} \qquad\qquad \textsc{Forall}\quad \dfrac{\Gamma \vdash a : \tau \qquad \bar{\alpha} \# \Gamma}{\Gamma \vdash \forall \bar{\alpha}.a : \forall \bar{\alpha}.\tau}$$

$$\left(\textsc{Gen}\quad \dfrac{\Gamma \vdash a : \tau \qquad \bar{\alpha} \# \Gamma, a}{\Gamma \vdash a : \forall \bar{\alpha}.\tau} \right)$$

Again, these constructs are erased prior to runtime.

(Why is this sound? Easy exercise: define the erasure of a term, and prove that the erasure of a well-typed term is well-typed.)

Constraint generation: existential case

Constraint generation for the existential form is straightforward:

$$\langle\!\langle (\exists\bar{\alpha}.a) : \tau \rangle\!\rangle \quad = \quad \exists\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \quad \text{if } \bar{\alpha} \# \tau$$

The type annotations inside a contain free occurrences of $\bar{\alpha}$. Thus, the constraint $\langle\!\langle a : \tau \rangle\!\rangle$ contains such occurrences as well. They are bound by the existential quantifier.

Constraint generation: existential case

For instance, the expression:

$$\lambda x_1.\,\lambda x_2.\,\exists\alpha.((x_1 : \alpha), (x_2 : \alpha))$$

has principal type scheme $\forall\alpha.\,\alpha \rightarrow \alpha \rightarrow \alpha \times \alpha$. Indeed, the generated constraint contains the pattern:

$$\exists\alpha.(\langle\!\langle x_1 : \alpha \rangle\!\rangle \wedge \langle\!\langle x_2 : \alpha \rangle\!\rangle \wedge \ldots)$$

which requires x_1 and x_2 to *share* a common (unspecified) type.

Constraint generation: universal case

A term a has type scheme, say, $\forall\alpha.\,\alpha \to \alpha$ if and only if a has type $\alpha \to \alpha$ *for every instance of α*, or, equivalently, for an abstract α.

To express this in terms of constraints, we introduce *universal quantification* in the constraint language:

$$C ::= \ldots \mid \forall\alpha.C$$

Its interpretation is standard.

(To solve these constraints, we will use an extension of the unification algorithm called unification under a mixed prefix—see ⏩ forward.)

The need for universal quantification in constraints arises when polymorphism is *required* by the programmer, as opposed to *inferred* by the system.

Constraint generation: universal case

Constraint generation for the universal form is somewhat subtle. A naive definition *fails* (why?):

$$\langle\!\langle \forall \bar{\alpha}.a : \tau \rangle\!\rangle = \forall \bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \qquad \text{if } \bar{\alpha} \mathrel{\#} \tau \qquad \textit{(Wrong)}$$

Constraint generation: universal case

Constraint generation for the universal form is somewhat subtle. A naive definition *fails*:

$$\langle\!\langle \forall\bar{\alpha}.a : \tau \rangle\!\rangle = \forall\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \qquad \text{if } \bar{\alpha} \# \tau \qquad \textit{(Wrong)}$$

This requires τ to be simultaneously equal to *all* of the types that a assumes when $\bar{\alpha}$ varies.

For instance, with this incorrect definition, one would have:

$$\begin{array}{rcl}
\langle\!\langle \forall\alpha.(\lambda x.\,x : \alpha \to \alpha) : \mathit{int} \to \mathit{int} \rangle\!\rangle & = & \forall\alpha.\langle\!\langle (\lambda x.\,x : \alpha \to \alpha) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\
& \equiv & \forall\alpha.(\langle\!\langle \lambda x.\,x : \alpha \to \alpha \rangle\!\rangle \wedge \alpha = \mathit{int}) \\
& \equiv & \forall\alpha.(\mathit{true} \wedge \alpha = \mathit{int}) \\
& \equiv & \mathit{false}
\end{array}$$

Constraint generation: universal case

A correct definition is:

$$\langle\!\langle \forall\bar{\alpha}.a : \tau \rangle\!\rangle = \forall\bar{\alpha}.\exists\gamma.\langle\!\langle a : \gamma \rangle\!\rangle \wedge \exists\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle$$

This requires a to be well-typed *for all* instances of $\bar{\alpha}$ and requires τ to be a valid type for a under *some* instance of $\bar{\alpha}$.

A problem with this definition is...

Constraint generation: universal case

A correct definition is:

$$\langle\!\langle \forall \bar\alpha.a : \tau \rangle\!\rangle = \forall \bar\alpha.\exists\gamma.\langle\!\langle a : \gamma \rangle\!\rangle \wedge \exists\bar\alpha.\langle\!\langle a : \tau \rangle\!\rangle$$

This requires a to be well-typed *for all* instances of $\bar\alpha$ and requires τ to be a valid type for a under *some* instance of $\bar\alpha$.

A problem with this definition is...

The term a is duplicated! This can lead to exponential complexity. Fortunately, this can be avoided modulo a slight extension of the constraint language [Pottier and Rémy, 2003, p. 112].

The solution defines:

$$\langle\!\langle \forall \bar\alpha.a : \tau \rangle\!\rangle = \textit{let}\; x : \forall \vec\alpha, \beta[\langle\!\langle a : \beta \rangle\!\rangle].\, \beta \;\textit{in}\; x \le \tau$$

where the new constrain form satisfies the equivalence:

$$\textit{let}\; x : \forall \vec\alpha, \vec\beta[C_1].\, \tau \;\textit{in}\; C_2 \equiv \forall \vec\alpha.\, \exists \vec\beta.\, C_1 \wedge \textit{def}\; x : \forall \vec\alpha, \vec\beta[C_1].\, \tau \;\textit{in}\; C_2$$

Type schemes as annotations

Annotating a term with a *type scheme,* rather than just a type, is now just syntactic sugar:

$$(a : \forall\bar{\alpha}.\tau) \quad \text{stands for} \quad \forall\bar{\alpha}.(a : \tau) \quad \text{if } \bar{\alpha} \mathrel{\#} a$$

In that particular case, constraint generation is in fact simpler:

$$\langle\!\langle (a : \forall\bar{\alpha}.\tau) : \tau' \rangle\!\rangle \equiv \forall\bar{\alpha}.\langle\!\langle a : \tau \rangle\!\rangle \wedge (\forall\bar{\alpha}.\tau) \leq \tau'$$

(Exercise: check this equivalence.)

Examples

A correct example:

$$\begin{array}{rl} & \langle\!\langle (\exists\alpha.(\lambda x.\,x+1 : \alpha \to \alpha)) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\ = & \exists\alpha.\langle\!\langle (\lambda x.\,x+1 : \alpha \to \alpha) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\ \equiv & \exists\alpha.(\alpha = \mathit{int}) \\ \equiv & \mathit{true} \end{array}$$

The system *infers* that α must be *int*. Because α is a local type variable, it does not appear in the final constraint.

Examples

An incorrect example:

$$\begin{array}{rl} & \langle\!\langle (\forall\alpha.(\lambda x.x+1 : \alpha \to \alpha)) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\ \Vdash & \forall\alpha.\exists\gamma.\langle\!\langle (\lambda x.x+1 : \alpha \to \alpha) : \gamma \rangle\!\rangle \\ \equiv & \forall\alpha.\exists\gamma.(\alpha = \mathit{int} \wedge \alpha \to \alpha = \gamma) \\ \equiv & \forall\alpha.\alpha = \mathit{int} \\ \equiv & \mathit{false} \end{array}$$

The system *checks* that α is used in an abstract way, which is not the case here, since the code implicitly assumes that α is *int*.

Examples

A correct example:

$$\begin{array}{ll} & \langle\!\langle (\forall \alpha.(\lambda x.\,x : \alpha \to \alpha)) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\ = & \forall \alpha.\exists \gamma.\langle\!\langle (\lambda x.\,x : \alpha \to \alpha) : \gamma \rangle\!\rangle \wedge \exists \alpha.\langle\!\langle (\lambda x.\,x : \alpha \to \alpha) : \mathit{int} \to \mathit{int} \rangle\!\rangle \\ \equiv & \forall \alpha.\exists \gamma.\alpha \to \alpha = \gamma \wedge \exists \alpha.\alpha = \mathit{int} \\ \equiv & \mathit{true} \end{array}$$

The system *checks* that α is used in an abstract way, which is indeed the case here.

It also checks that, if α is appropriately instantiated, the code admits the expected type $\mathit{int} \to \mathit{int}$.

Examples

An incorrect example:

$$\begin{array}{ll} & \langle\!\langle \exists\alpha.(\mathit{let}\ f = (\lambda x.\, x : \alpha \to \alpha)\ \mathit{in}\ (f\ 0, f\ \mathit{true})) : \gamma \rangle\!\rangle \\ \equiv & \exists\alpha.(\mathit{let}\ f : \alpha \to \alpha\ \mathit{in} \\ & \qquad \exists\gamma_1\gamma_2.(f \preceq \mathit{int} \to \gamma_1 \wedge f \preceq \mathit{bool} \to \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma)) \\ \equiv & \exists\alpha\gamma_1\gamma_2.(\alpha \to \alpha = \mathit{int} \to \gamma_1 \wedge \alpha \to \alpha = \mathit{bool} \to \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma) \\ \Vdash & \exists\alpha.(\alpha = \mathit{int} \wedge \alpha = \mathit{bool}) \\ \equiv & \mathit{false} \end{array}$$

α is bound *outside* the let construct; f receives the monotype $\alpha \to \alpha$.

Examples

A correct example:

$$\begin{array}{rl}
& \langle\!\langle \mathit{let}\ f = \exists\alpha.(\lambda x.\, x : \alpha \to \alpha)\ \mathit{in}\ (f\ 0, f\ \mathit{true}) : \gamma \rangle\!\rangle \\
\equiv & \mathit{let}\ f : \forall\beta[\exists\alpha.(\alpha \to \alpha = \beta)].\,\beta\ \mathit{in} \\
& \quad \exists\gamma_1\gamma_2.(f \le \mathit{int} \to \gamma_1 \wedge f \le \mathit{bool} \to \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma) \\
\equiv & \mathit{let}\ f : \forall\alpha.\,\alpha \to \alpha\ \mathit{in} \\
& \quad \exists\gamma_1\gamma_2.(\ldots) \\
\equiv & \exists\gamma_1\gamma_2.(\mathit{int} = \gamma_1 \wedge \mathit{bool} = \gamma_2 \wedge \gamma_1 \times \gamma_2 = \gamma) \\
\equiv & \mathit{int} \times \mathit{bool} = \gamma
\end{array}$$

α is bound *within* the let construct; the term $\exists\alpha.(\lambda x.\, x : \alpha \to \alpha)$ has the same principal type scheme as $\lambda x.\, x$, namely $\forall\alpha.\,\alpha \to \alpha$; f receives the type scheme $\forall\alpha.\,\alpha \to \alpha$.

Type annotations in the real world

For historical reasons, in Objective Caml, type variables are not explicitly bound. (Retrospectively, that's *bad!*) They are implicitly *existentially* bound at the nearest enclosing toplevel let construct.

In Standard ML, type variables are implicitly *universally* bound at the nearest enclosing toplevel let construct.

In Glasgow Haskell, type variables are implicitly existentially bound within patterns: *'A pattern type signature brings into scope any type variables free in the signature that are not already in scope'* [Peyton Jones and Shields, 2004].

Constraints help understand these varied design choices uniformly.

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Monomorphic recursion

Recall the typing rule for recursive functions:

$$\textsc{FixAbs}\quad \frac{\Gamma, f : \tau \vdash \lambda x.\, a : \tau}{\Gamma \vdash \mu f.\lambda x.a : \tau}$$

It leads to the following derived typing rule:

$$\textsc{LetRec}\quad \frac{\Gamma, f : \tau_1 \vdash \lambda x.\, a_1 : \tau_1 \qquad \bar\alpha \# \Gamma, a_1 \qquad \Gamma, f : \forall \bar\alpha.\, \tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash \textit{let rec } f\ x = a_1 \textit{ in } a_2 : \tau_2}$$

Any comments?

Monomorphic recursion

These rules require occurrences of f to have *monomorphic type* within the recursive definition (that is, within $\lambda x.\, a_1$).

This is visible also in terms of type inference. The constraint

$$\langle\!\langle \textit{let rec } f\ x = a_1 \textit{ in } a_2 : \tau \rangle\!\rangle$$

is equivalent to

$$\textit{let } f : \forall \alpha\beta[\textit{let } f : \alpha \to \beta; x : \alpha \textit{ in } \langle\!\langle a_1 : \beta \rangle\!\rangle].\, \alpha \to \beta \textit{ in } \langle\!\langle a_2 : \tau \rangle\!\rangle$$

Monomorphic recursion

This is problematic in some situations, most particularly when defining functions over *nested algebraic data types* [Bird and Meertens, 1998; Okasaki, 1999].

Polymorphic recursion

This problem is solved by introducing *polymorphic recursion*, that is, by allowing μ-bound variables to receive a polymorphic type scheme:

$$\textsc{FixAbsPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a : \sigma}{\Gamma \vdash \mu f.\lambda x.a : \sigma} \qquad \textsc{LetRecPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a_1 : \sigma \qquad \Gamma, f : \sigma \vdash a_2 : \tau}{\Gamma \vdash \textit{let rec } f\ x = a_1 \textit{ in } a_2 : \tau}$$

This extension of ML is due to Mycroft [1984].

In System F, there is no problem to begin with; no extension is necessary.

Polymorphic recursion

Polymorphic recursion alters, to some extent, Damas and Milner's type system.

Now, not only *let-bound,* but also *μ-bound* variables receive type schemes. The type system is no longer equivalent, up to reduction to let-normal form, to simply-typed λ-calculus.

This has two consequences:

- *monomorphization,* a technique employed in some ML compilers [Tolmach and Oliva, 1998; Cejtin et al., 2007], is no longer possible;

Polymorphic recursion

Polymorphic recursion alters, to some extent, Damas and Milner's type system.

Now, not only *let-bound,* but also *μ-bound* variables receive type schemes. The type system is no longer equivalent, up to reduction to let-normal form, to simply-typed λ-calculus.

This has two consequences:

- *monomorphization,* a technique employed in some ML compilers [Tolmach and Oliva, 1998; Cejtin et al., 2007], is no longer possible;
- *type inference* becomes problematic!

Polymorphic recursion

Type inference for ML with polymorphic recursion is undecidable [Henglein, 1993]. It is equivalent to the undecidable problem of *semi-unification.*

Polymorphic recursion

Yet, type inference in the presence of polymorphic recursion can be made simple. (How?)

Polymorphic recursion

Yet, type inference in the presence of polymorphic recursion can be made simple. (How?)

By relying on a *mandatory type annotation*. The rules become:

$$\textsc{FixAbsPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a : \sigma}{\Gamma \vdash \mu(f : \sigma).\lambda x.a : \sigma} \qquad \textsc{LetRecPoly}\quad \frac{\Gamma, f : \sigma \vdash \lambda x.\, a_1 : \sigma \qquad \Gamma, f : \sigma \vdash a_2 : \tau}{\Gamma \vdash \textit{let rec}\ (f : \sigma) = \lambda x.\, a_1\ \textit{in}\ a_2 : \tau}$$

The type scheme σ no longer has to be guessed.

With this feature, contrary to what was said earlier ◂ back, *type annotations are not just restrictive:* they are sometimes required for type inference to succeed.

Polymorphic recursion

The constraint generation rule becomes:

$\langle\!\langle \textit{let rec}\ (f : \sigma) = \lambda x.\, a_1\ \textit{in}\ a_2 : \tau \rangle\!\rangle = ?$

Polymorphic recursion

The constraint generation rule becomes:

$$\langle\!\langle \textit{let rec}\,(f : \sigma) = \lambda x.\, a_1 \textit{ in } a_2 : \tau \rangle\!\rangle = \textit{let } f : \sigma \textit{ in } (\langle\!\langle \lambda x.\, a_1 : \sigma \rangle\!\rangle \wedge \langle\!\langle a_2 : \tau \rangle\!\rangle)$$

It is clear that f receives type scheme σ both *inside and outside* of the recursive definition.

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Unification under a mixed prefix

Unification under a mixed prefix means unification in the presence of both existential and universal quantifiers.

We extend the basic unification algorithm with support for universal quantification.

The solved forms are unchanged: universal quantifiers are always *eliminated.*

Unification under a mixed prefix

In short, in order to reduce $\forall\bar{\alpha}.C$ to a solved form, where C is itself a solved form:

- if a rigid variable is equated with a constructed type, fail;
  $\forall\alpha.\exists\beta\gamma.(\alpha = \beta \rightarrow \gamma)$ is false;
- if two rigid variables are equated, fail;
  $\forall\alpha\beta.(\alpha = \beta)$ is false;
- if a free variable dominates a rigid variable, fail;
  $\forall\alpha.\exists\beta.(\gamma = \alpha \rightarrow \beta)$ is false;
- otherwise, one can decompose C as $\exists\bar{\beta}.(C_1 \wedge C_2)$, where $\bar{\alpha}\bar{\beta} \# C_1$ and $\exists\bar{\beta}.C_2 \equiv$ *true*; in that case, $\forall\bar{\alpha}.C$ reduces to just C_1.

  $\forall\alpha.\exists\beta\gamma_1.(\beta = \alpha \rightarrow \gamma_1 \wedge \gamma = \gamma_1 \rightarrow \gamma_1)$ reduces to $\exists\gamma_1.(\gamma = \gamma_1 \rightarrow \gamma_1)$.
  $\forall\alpha.\exists\beta.(\beta = \alpha \rightarrow \gamma)$ is equivalent to *true*.

See [Pottier and Rémy, 2003, p. 109] for details.

Examples

Objective Caml implements a form of unification under a mixed prefix:

```
bash$ ocaml
# let module M : sig val id : ’a → ’a end
                 = struct let id x = x + 1 end
  in M.id;;
Values do not match: val id : int → int
is  not included in val id : ’a → ’a
```

This example gives rise to a constraint of the form $\forall\alpha.\alpha = \mathit{int}$.

Examples

Here is another example:

```
bash$ ocaml
# let r = ref (fun x → x) in
  let module M : sig val id : 'a → 'a end
               = struct let id = !r end
  in M.id;;
Values do not match: val id : '_a → '_a
is not included in val id : 'a → 'a
```

This example gives rise to a constraint of the form $\exists\beta.\forall\alpha.(\alpha = \beta)$.

Contents

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Recursive types

Product and sum types alone do not allow describing *data structures* of *unbounded size*, such as lists and trees.

Indeed, if the grammar of types is $\tau ::= \textit{unit} \mid \tau \times \tau \mid \tau + \tau$, then it is clear that every type describes a *finite* set of values.

For every k, the type of lists of length at most k is expressible using this grammar. However, the type of lists of unbounded length is not.

Equi- versus iso-recursive types

The following definition is inherently *recursive:*

> *"A list is either empty or a pair of an element and a list."*

We need something like this:

$$\mathit{list}\;\alpha \quad \diamond \quad \mathit{unit} + \alpha \times \mathit{list}\;\alpha$$

But what does $\diamond$ stand for? Is it *equality,* or some kind of *isomorphism?*

Equi- versus iso-recursive types

There are two standard approaches to recursive types, dubbed the *equi-recursive* and *iso-recursive* approaches.

In the equi-recursive approach, a recursive type is *equal* to its unfolding.

In the iso-recursive approach, a recursive type and its unfolding are related via explicit *coercions*.

Equi-recursive types

In the equi-recursive approach, the usual syntax of types:

$$\tau ::= \alpha \mid \mathsf{F}\ \vec{\tau}$$

is no longer interpreted inductively. Instead, types are the *regular trees* built on top of this signature.

Finite syntax for equi-recursive types

If desired, it is possible to use *finite syntax* for recursive types:

$$\tau ::= \alpha \mid \mu\alpha.(\mathsf{F}\ \vec{\tau})$$

We do not allow the seemingly more general $\mu\alpha.\tau$, because $\mu\alpha.\alpha$ is meaningless, and $\mu\alpha.\beta$ or $\mu\alpha.\mu\beta.\tau$ are useless. If we write $\mu\alpha.\tau$, it should be understood that τ is *contractive,* that is, τ is a type constructor application.

For instance, the type of lists of elements of type α is:

$$\mu\beta.(\mathit{unit} + \alpha \times \beta)$$

Finite syntax for equi-recursive types

Each type in this syntax denotes a unique regular tree, sometimes known as its *infinite unfolding.* Conversely, every regular tree can be expressed in this notation (possibly in more than one way).

If one builds a type-checker on top of this finite syntax, then one must be able to *decide* whether two types are *equal,* that is, have identical infinite unfoldings.

This can be done efficiently, either via the algorithm for comparing two DFAs, or by unification. (The latter approach is simpler, faster, and extends to the type inference problem.)

Finite syntax for equi-recursive types

One can also prove [Brandt and Henglein, 1998] that equality is the least congruence generated by the following two rules:

Fold/Unfold

$$\mu\alpha.\tau = [\alpha \mapsto \mu\alpha.\tau]\tau$$

Uniqueness

$$\frac{\tau_1 = [\alpha \mapsto \tau_1]\tau \qquad \tau_2 = [\alpha \mapsto \tau_2]\tau}{\tau_1 = \tau_2}$$

In both rules, τ must be contractive.

This axiomatization does not directly lead to an efficient algorithm for deciding equality, though.

Type soundness for equi-recursive types

In the presence of equi-recursive types, structural induction on types is no longer permitted – but *we never used it* anyway.

It remains true that F $\vec{\tau}_1$ = F $\vec{\tau}_2$ implies $\vec{\tau}_1 = \vec{\tau}_2$ – this was used in our Subject Reduction proofs.

It remains true that F$_1$ $\vec{\tau}_1$ = F$_2$ $\vec{\tau}_2$ implies F$_1$ = F$_2$ – this was used in our Progress proofs.

So, the reasoning that leads to *type soundness* is unaffected.

(Exercise: prove type soundness for the simply-typed λ-calculus in Coq. Then, change the syntax of types from `Inductive` to `CoInductive`.)

Type inference for equi-recursive types

How is type inference adapted for equi-recursive types?

The *syntax* of constraints is unchanged: they remain systems of equations between finite first-order types, without μ's. Their *interpretation* changes: they are now interpreted in a universe of regular trees.

As a result,

- constraint generation is *unchanged;*
- constraint solving is adapted by *removing the occurs check.*

(Exercise: describe solved forms and show that every solved form is either *false* or satisfiable.)

Type inference for equi-recursive types

Here is a function that measures the length of a list:

$$
\begin{array}{l}
\mu \mathit{length}.\lambda xs.\mathit{case}\ xs\ \mathit{of} \\
\qquad \lambda().\,0 \\
\quad [\,]\ \lambda(x, xs).\,1 + \mathit{length}\ xs
\end{array}
$$

Type inference gives rise to the *cyclic equation:*

$$\beta = \mathit{unit} + \alpha \times \beta$$

where *length* has type $\beta \to \mathit{int}$.

Type inference for equi-recursive types

That is, *length* has *principal type scheme:*

$$\forall\alpha.\,(\mu\beta.\mathit{unit} + \alpha \times \beta) \to \mathit{int}$$

or, equivalently, principal constrained type scheme:

$$\forall\alpha[\beta = \mathit{unit} + \alpha \times \beta].\,\beta \to \mathit{int}$$

The cyclic equation that characterizes lists was never provided by the programmer, but was inferred.

Type inference for equi-recursive types

Objective Caml implements equi-recursive types upon explicit request:

```
bash$ ocaml −rectypes
# type ('a, 'b) sum = Left of 'a | Right of 'b;;
  type ('a, 'b) sum = Left of 'a | Right of 'b

  # let rec length xs =
      match xs with
      | Left () → 0
      | Right (x, xs) → 1 + length xs ;;
  val length : ((unit, 'b * 'a) sum as 'a) → int = ⟨fun⟩
```

Quiz: why is `-rectypes` only an option?

Drawbacks of equi-recursive types

Equi-recursive types are simple and powerful. In practice, however, they are perhaps *too expressive:*

```
bash$ ocaml −rectypes
# let rec map f = function
  | [] → []
  | x :: xs → map f x :: map f xs;;
val map : 'a → ('b list as 'b) → ('c list as 'c) = ⟨fun⟩

# map (fun x → x + 1) [ 1; 2 ];;
This expression has type int but is used with type 'a list as 'a

# map () [[];[[]]];;
− : 'a list as 'a = [[]; [[]]]
```

Equi-recursive types allow this nonsensical version of map to be accepted, thus delaying the detection of a programmer error.

Half a pint of equi-recursive types

Quiz: why is this accepted?

```
bash$ ocaml
# let f x = x#hello x;;
val f : (< hello : 'a → 'b; .. > as 'a) → 'b = ⟨fun⟩
```

Iso-recursive types

In the iso-recursive approach, the user is allowed to introduce new *type constructors* D via (possibly mutually recursive) *declarations:*

$$D\,\vec{\alpha} \approx \tau \qquad \text{(where } \mathrm{ftv}(\tau) \subseteq \bar{\alpha}\text{)}$$

Each such declaration adds a unary constructor fold_D and a unary destructor unfold_D with the following types:

$$\begin{array}{rcl} \mathit{fold}_D & : & \forall \bar{\alpha}.\, \tau \rightarrow D\,\vec{\alpha} \\ \mathit{unfold}_D & : & \forall \bar{\alpha}.\, D\,\vec{\alpha} \rightarrow \tau \end{array}$$

and the reduction rule:

$$\mathit{unfold}_D\ (\mathit{fold}_D\ v) \longrightarrow v$$

Iso-recursive types

Ideally, iso-recursive types should not have any runtime cost.

One solution is to compile constructors and destructors away into a target language with equi-recursive types.

Another solution is to see iso-recursive types as a restriction of equi-recursive types where the source language does not have equi-recursive types but instead two unary destructors fold_D and unfold_D with the semantics of the identity function.

Subject reduction does not hold in the source language, but only in the full language with iso-recursive types. Applications of destructors can also be reduced at compile time.

Note that iso-recursive types are less expressive than equi-recursive types, as there is no counter-part to the Uniqueness typing rule.

Iso-recursive lists

A parametrized, iso-recursive type of lists is:

$$\mathit{list}\,\alpha \approx \mathit{unit} + \alpha \times \mathit{list}\,\alpha$$

The empty list is:

$$\mathit{fold}_{\mathit{list}}\,(\mathit{inj}_1\,())\,:\,\forall\alpha.\,\mathit{list}\,\alpha$$

A function that measures the length of a list is:

$$\left(\begin{array}{l}\mu\mathit{length}.\lambda xs.\mathit{case}\,(\mathit{unfold}_{\mathit{list}}\;xs)\;\mathit{of}\\ \qquad\lambda().\,0\\ \qquad[]\;\lambda(x,xs).\,1+\mathit{length}\;xs\end{array}\right) : \forall\alpha.\,\mathit{list}\,\alpha\to\mathit{int}$$

One *folds upon construction* and *unfolds upon deconstruction.*

Type inference for iso-recursive types

In the iso-recursive approach, *types remain finite.* The type *list* α is just an application of a type constructor to a type variable.

As a result, *type inference is unaffected.* The occurs check remains.

Algebraic data types

Algebraic data types result of the fusion of iso-recursive types with structural, labeled products and sums.

This suppresses the *verbosity* of explicit folds and unfolds as well as the *fragility* and inconvenience of numeric indices – instead, named *record fields* and *data constructors* are used.

For instance,

$$\mathit{fold}_{\mathit{list}}\,(\mathit{inj}_1\,())\quad\text{is replaced with}\quad \mathit{Nil}\,()$$

Algebraic data type declarations

An algebraic data type constructor D is introduced via a *record type* or *variant type* definition:

$$D\,\vec{\alpha} \approx \prod_{\ell \in L} \ell : \tau_\ell \qquad \text{or} \qquad D\,\vec{\alpha} \approx \sum_{\ell \in L} \ell : \tau_\ell$$

L denotes a finite set of record labels or data constructors.

Algebraic data type definitions can be mutually recursive.

Effects of a record type declaration

The record type definition $D\,\vec{\alpha} \approx \prod_{\ell \in L} \ell : \tau_\ell$ introduces syntax for *constructing* and *deconstructing* records:

$$C ::= \dots \mid \{\ell = \cdot\}_{\ell \in L} \qquad\qquad d ::= \dots \mid \cdot.\ell$$

With the following types

$$\begin{aligned} \{\ell_1 = \cdot, \dots, \ell_n\} :&\quad \forall \vec{\alpha}.\, \tau_{\ell_1} \to \dots \tau_{\ell_n} \to D\,\vec{\alpha} \\ \cdot.\ell :&\quad \forall \vec{\alpha}.\, D\,\vec{\alpha} \to \tau_\ell \end{aligned}$$

Effects of a variant type declaration

The variant type definition $D\,\vec{\alpha} \approx \sum_{\ell \in L} \ell : \tau_\ell$ introduces syntax for *constructing* and *deconstructing* variants:

$$C ::= \ldots \mid \ell \qquad\qquad d ::= \ldots \mid \textit{case} \,\cdot\, \textit{of}\,[\ell : \cdot]_{\ell \in L}$$

With the following types:

$$\begin{array}{rl} \textit{case} \,\cdot\, \textit{of}\,[\ell_1 : \cdot \,[\!]\, \ldots \ell_n : \cdot] : & \forall \vec{\alpha}\beta.\, D\,\vec{\alpha} \to (\tau_{\ell_1} \to \beta) \to \ldots (\tau_{\ell_n} \to \beta) \to \beta \\ \cdot.\ell : & \forall \vec{\alpha}.\, \tau_\ell \to D\,\vec{\alpha} \end{array}$$

An example: lists

Here is an algebraic data type of lists:

$$\mathit{list}\ \alpha \approx \mathit{Nil} : \mathit{unit} + \mathit{Cons} : \alpha \times \mathit{list}\ \alpha$$

This gives rise to:

$$\begin{aligned} \mathit{case}\ \cdot\ \mathit{of}\,[\,\mathit{Nil} : \cdot \,\|\, \ldots \mathit{Cons} : \cdot\,] : &\quad \forall \alpha\beta.\, \mathit{list}\ \ \alpha \to (\mathit{unit} \to \beta) \to \\ &\qquad\qquad ((\alpha \times \mathit{list}\ \alpha) \to \beta) \to \beta \\ \mathit{Nil} : &\quad \forall \alpha.\, \mathit{unit} \to \mathit{list}\ \alpha \\ \mathit{Cons} : &\quad \forall \alpha.\, (\alpha \times \mathit{list}\ \alpha) \to \mathit{list}\ \alpha \end{aligned}$$

A function that measures the length of a list is:

$$\left(\begin{array}{l} \mu \mathit{length}.\lambda xs.\mathit{case}\ xs\ \mathit{of} \\ \qquad \mathit{Nil} : \lambda ().\, 0 \\ \quad\ \| \ \mathit{Cons} : \lambda (x, xs).\, 1 + \mathit{length}\ xs \end{array}\right) : \forall \alpha.\, \mathit{list}\ \alpha \to \mathit{int}$$

A word on mutable fields

In Objective Caml, a record field can be marked *mutable*. This introduces an extra binary destructor for writing this field:

$$(\cdot.\ell \leftarrow \cdot) : \forall \vec{\alpha}. D\, \vec{\tau} \rightarrow \tau_\ell \rightarrow \mathit{unit}$$

This also makes record construction a destructor since, when fully applied it is *not a value* but it allocates a piece of store and returns its location.

Thus, due to the value restriction, the type of such expressions cannot be generalized.

Contents

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

HM(X)

Soundness/completeness of type inference are in fact easier to prove if one adopts a *constraint-based specification* of the type system.

In HM(X), judgments take the form

$$C, \Gamma \vdash a : \tau$$

called a constrained typing judgment and should be read *under the asumption C and typing environment Γ, the program a has type τ.*

Here, C ranges over first-order typing constraints as earlier.

However, we require type constraint to have no free program variables.

In a constrained typing judgement, C contrains free *type* variables of the judgement while Γ provides the type of free *program* variables.

This generalizes Damas and Milner's type system. See Odersky et al. [1999], Pottier and Rémy [2005], Skalka and Pottier [2002] for a detailed treatment.

HM(X) Entailment and subtyping

Typing rules also use an entailment predicate $C \Vdash C'$ between constraints that is more general than constraint equivalence.

Entailment is defined as expected: $C \Vdash C'$ if and only if any ground assignment that satisfies C also satisfies C'.

Then, two constraints are equivalent iff each one entails the other.

Typing judgments for HM(X) are taken up to constraint equivalence.

The parameter X for HM(X) stands for the logic of the constraints.
We have so far only considered constraints with an equality predicate.
Here, we use a more general subtyping predicate $\leq$ that we assume to be contravariant on arrow types:

$$\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' \quad \equiv \quad \tau_2 \leq \tau_2' \wedge \tau_1' \leq \tau_1$$

HM(X) Typing rules

$$\textsc{hm-Var}\quad \frac{\sigma = \Gamma(x)}{\Gamma \vdash x : \sigma}$$

$$\textsc{hm-Abs}\quad \frac{(\Gamma, x : \tau_0) \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}$$

$$\textsc{hm-App}\quad \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\textsc{hm-Let}\quad \frac{\begin{array}{c}\Gamma \vdash a_1 : \sigma \\ (\Gamma, x : \sigma) \vdash a_2 : \tau\end{array}}{\Gamma \vdash \textit{let}\ x = a_1\ \textit{in}\ a_2 : \tau}$$

$$\textsc{hm-Gen}\quad \frac{\Gamma \vdash a : \tau \qquad \vec{\alpha} \# \quad , \Gamma}{\Gamma \vdash a : \forall \bar{\alpha}[\quad].\tau}$$

$$\textsc{hm-Inst}\quad \frac{\Gamma \vdash a : \forall \vec{\alpha}[\quad].\tau}{\Gamma \vdash a : \tau}$$

HM(X)

Typing rules

hm-Var
$$\frac{\sigma = \Gamma(x) \qquad C \Vdash \exists\sigma}{C, \Gamma \vdash x : \sigma}$$

hm-Abs
$$\frac{C, (\Gamma, x : \tau_0) \vdash a : \tau}{C, \Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}$$

hm-App
$$\frac{C, \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad C, \Gamma \vdash a_2 : \tau_2}{C, \Gamma \vdash a_1\ a_2 : \tau_1}$$

hm-Let
$$\frac{\begin{array}{c} C, \Gamma \vdash a_1 : \sigma \\ C, (\Gamma, x : \sigma) \vdash a_2 : \tau \end{array}}{C, \Gamma \vdash \textit{let}\ x = a_1\ \textit{in}\ a_2 : \tau}$$

hm-Gen
$$\frac{C \wedge C_0, \Gamma \vdash a : \tau \qquad \vec{\alpha} \# C, \Gamma}{C \wedge \exists\vec{\alpha}.\, C_0, \Gamma \vdash a : \forall\bar{\alpha}[C_0].\, \tau}$$

hm-Inst
$$\frac{C, \Gamma \vdash a : \forall\vec{\alpha}[C_0].\, \tau}{C \wedge C_0, \Gamma \vdash a : \tau}$$

HM(X)

Typing rules

HM-VAR

$$\frac{\sigma = \Gamma(x) \qquad C \Vdash \exists \sigma}{C, \Gamma \vdash x : \sigma}$$

HM-ABS

$$\frac{C, (\Gamma, x : \tau_0) \vdash a : \tau}{C, \Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}$$

HM-APP

$$\frac{C, \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad C, \Gamma \vdash a_2 : \tau_2}{C, \Gamma \vdash a_1\ a_2 : \tau_1}$$

HM-LET

$$\frac{\begin{array}{c} C, \Gamma \vdash a_1 : \sigma \\ C, (\Gamma, x : \sigma) \vdash a_2 : \tau \end{array}}{C, \Gamma \vdash \textit{let}\ x = a_1\ \textit{in}\ a_2 : \tau}$$

HM-GEN

$$\frac{C \wedge C_0, \Gamma \vdash a : \tau \qquad \vec{\alpha} \# C, \Gamma}{C \wedge \exists \vec{\alpha}.\, C_0, \Gamma \vdash a : \forall \bar{\alpha}[C_0].\, \tau}$$

HM-INST

$$\frac{C, \Gamma \vdash a : \forall \vec{\alpha}[C_0].\, \tau}{C \wedge C_0, \Gamma \vdash a : \tau}$$

HM-SUB

$$\frac{C, \Gamma \vdash a : \tau_1 \qquad C \Vdash \tau_1 \leq \tau_2}{C, \Gamma \vdash a : \tau_2}$$

HM-EXISTS

$$\frac{C, \Gamma \vdash a : \tau \qquad \vec{\alpha} \# \Gamma, \tau}{\exists \vec{\alpha}.\, C, \Gamma \vdash a : \tau}$$

HM(X)

The constraint $\exists\sigma$ when σ is a type scheme $\forall\bar{\alpha}[C_0].\tau$ means $\exists\bar{\alpha}.C_0$, *i.e.* that the type scheme is non empty (in premisse of Rule HM-VAR).

A *valid* judgment is one that has a derivation with those typing rules.

In a valid judgment, C may not be satisfiable.

A program is well-typed in environment Γ if the judgment $C, \Gamma \vdash a : \tau$ is valid and C is *satisfiable*.

HM(=) Compared with ML

When considering equality only constraints, HM(=) is equivalent to ML:

HM(=) is a conservative extension of ML:

If Γ and τ contain only Damas-Milner's type schemes, then

$$\Gamma \vdash a : \tau \in \mathit{ML} \quad \Longleftrightarrow \quad \mathit{true}, \Gamma \vdash a : \tau \in \mathit{HM}(=)$$

HM(=) does not add expressiveness to ML:

If $C, \Gamma \vdash a : \tau \in$ HM(=) and φ is an idempotent solution of C, then $\Gamma_\varphi \vdash a : \tau_\varphi \in$ ML.

where $(\cdot)_\varphi$ translates HM(=) type schemes into ML type schemes, applying the substitution φ on the fly.

PCB(X)

As for ML, there is a syntax directed presentation of typing rules.

However, we may take advantage of program variables in constraints to go one step further and mix the constraint C (without free program variables) and the typing environment Γ into a single constraint C now allowing free program variables.

Judgments take the form $C \vdash a : \tau$ where C both constrains type variables and assigns constrained type schemes to program variables.

The type system, called PCB(X), is equivalent to HM(X)—see Pottier and Rémy [2005] a detailed presentation.

PCB(X)

$$\textsc{pcb-Var}\quad \frac{C \Vdash x \preceq \tau}{C \vdash x : \tau} \qquad \textsc{pcb-Abs}\quad \frac{C \vdash a : \tau}{\mathit{let}\ x : \tau_0\ \mathit{in}\ C \vdash a : \tau_0 \rightarrow \tau} \qquad \textsc{pcb-App}\quad \frac{\begin{array}{c} C_1 \vdash a_1 : \tau_2 \rightarrow \tau_1 \\ C_2 \vdash a_2 : \tau_2 \end{array}}{C_1 \wedge C_2 \vdash a_1\ a_2 : \tau_1}$$

$$\textsc{pcb-Let}\quad \frac{C_1 \vdash a_1 : \tau_1 \qquad C_2 \vdash a_2 : \tau_2}{\mathit{let}\ x : \forall \mathcal{V}[C_1].\,\tau_1\ \mathit{in}\ C_2 \vdash \mathit{let}\ x = a_1\ \mathit{in}\ a_2 : \tau_2}$$

$$\textsc{pcb-Sub}\quad \frac{C \vdash a : \tau_1}{C \wedge \tau_1 \leq \tau_2 \vdash a : \tau_2} \qquad \textsc{pcb-Exists}\quad \frac{C \vdash a : \tau \qquad \alpha \mathrel{\#} \tau}{\exists \alpha.\, C \vdash a : \tau}$$

Soundness and completeness of PCB(=)

The type inference algorithm for ML is sound and complete for PCB(=):

- *Soundness:* $\langle\!\langle a : \tau \rangle\!\rangle \vdash a : \tau$.
  The constraint inferred for a typing validates the typing.
- *Completeness:* If $C \vdash a : \tau$ then $C \Vdash \langle\!\langle a : \tau \rangle\!\rangle$.
  The constraint inferred for a typing is more general than any constraint that validates the typing.

HM(≤)

In the presence of subtyping, we must recheck type soundness.

Contents

- Introduction
- Type inference for simply-typed λ-calculus
- Type inference for ML
  - Constraint-based type inference for ML
  - Constraint solving by example
  - Type reconstruction
- Type annotations
  - Polymorphic recursion
  - Unification under a mixed prefix
- Equi- and iso-recursive types
- HM(X)
- System F

Full type inference

Type inference has long been an open problem for System F, until Wells [1999] showed that it is in fact undecidable by showing it is equivalent to the semi-unification problem which was earlier proved undecidable.

Type-checking in explicitly-typed System F is indeed feasible and easy (still, an implementation must be careful with renaming of variables when applying substitutions).

However, we have seen that programming with fully-explicit types is unpractical.

Several solutions for *partial type inference* are used in practice. They may alleviate the need for a lot of redundant type annotations. However, none of them is fully satisfactory.

Type inference and second-order unification

The full type-inference problem is not directly related to second-order unification but rather to semi-unification.

However, it becomes equivalent to second-order unification if the positions of type abstractions and type applications are explicit. That is, terms are

$$M ::= x \mid \lambda x:?.\,M \mid M\ M \mid \Lambda ?.M \mid M\ ?$$

where the question marks stand for type variables names and types to infer.

Second-order unification is still undecidable, but their are semi-algorithms that often work well in some cases.

This method was proposed by Pfenning [1988].

Implicit type arguments

Derived from this solution, one can add decorations to let-bindings to indicate that some type arguments are left implicit.

Then, every occurrence of such a variable automatically adds type applications holes at the corresponding positions and type parameters will be inferred using second-order unification.

Other type applications must be passed explicitly.

Bidirectional type inference

What makes type-checking easy is that typing rules have an algorithmic reading. This implies that they are syntax directed, but also that judgments can be read as functions where some arguments are inputs and others are output.

Typically, Γ and a would be inputs and τ is an ouput in the judgement $\Gamma \vdash a : \tau$.

However, although the rules for simply-typed λ-calculus are syntax directed they do not have an algorithmic reading;

The rule for abstraction is

$$\frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}\;\textsc{Abs}$$

Then τ_0 is used both as input and output in the premise.

Bidirectional type inference

However, in some cases, the type of the function may be known, *e.g.* when the function is an argument to an expression of a known type.

In such cases, it suffices to check the proposed type is indeed correct.

Formally, the typing judgment $\Gamma \vdash a : \tau$ may be split into two judgments $\Gamma \vdash a \Downarrow \tau$ to check that a may be assigned the type τ and $\Gamma \vdash a \Uparrow \tau$ to infer the type τ of a.

Bidirectional type inference simple types

$$\frac{\tau = \Gamma(x)}{\Gamma \vdash x \Uparrow \tau}\ \textsc{Var-I} \qquad \frac{\Gamma, x : \tau_0 \vdash a \Downarrow \tau}{\Gamma \vdash \lambda x.\, a \Downarrow \tau_0 \to \tau}\ \textsc{Abs-C} \qquad \frac{\Gamma \vdash a_1 \Uparrow \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 \Downarrow \tau_2}{\Gamma \vdash a_1\ a_2 \Uparrow \tau_1}\ \textsc{App-I}$$

Bidirectional type inference simple types

Var-I
$$\frac{\tau = \Gamma(x)}{\Gamma \vdash x \Uparrow \tau}$$

Abs-C
$$\frac{\Gamma, x : \tau_0 \vdash a \Downarrow \tau}{\Gamma \vdash \lambda x.\, a \Downarrow \tau_0 \rightarrow \tau}$$

App-I
$$\frac{\Gamma \vdash a_1 \Uparrow \tau_2 \rightarrow \tau_1 \qquad \Gamma \vdash a_2 \Downarrow \tau_2}{\Gamma \vdash a_1\, a_2 \Uparrow \tau_1}$$

I-C
$$\frac{\Gamma \vdash a \Uparrow \tau}{\Gamma \vdash a \Downarrow \tau}$$

Checking mode can use inference mode.

Bidirectional type inference

simple types

$$\textsc{Var-I}\ \frac{\tau = \Gamma(x)}{\Gamma \vdash x \Uparrow \tau} \qquad \textsc{Abs-C}\ \frac{\Gamma, x : \tau_0 \vdash a \Downarrow \tau}{\Gamma \vdash \lambda x.\, a \Downarrow \tau_0 \to \tau} \qquad \textsc{App-I}\ \frac{\Gamma \vdash a_1 \Uparrow \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 \Downarrow \tau_2}{\Gamma \vdash a_1\ a_2 \Uparrow \tau_1}$$

$$\textsc{I-C}\ \frac{\Gamma \vdash a \Uparrow \tau}{\Gamma \vdash a \Downarrow \tau} \qquad \textsc{Annot-I}\ \frac{\Gamma \vdash a \Downarrow \tau}{\Gamma \vdash (a : \tau) \Uparrow \tau}$$

Checking mode can use inference mode.

Annotations turn inference mode into checking mode.

Bidirectional type inference

simple types

$$\textsc{Var-I}\ \frac{\tau = \Gamma(x)}{\Gamma \vdash x \Uparrow \tau} \qquad \textsc{Abs-C}\ \frac{\Gamma, x : \tau_0 \vdash a \Downarrow \tau}{\Gamma \vdash \lambda x.\, a \Downarrow \tau_0 \to \tau} \qquad \textsc{App-I}\ \frac{\Gamma \vdash a_1 \Uparrow \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 \Downarrow \tau_2}{\Gamma \vdash a_1\ a_2 \Uparrow \tau_1}$$

$$\textsc{I-C}\ \frac{\Gamma \vdash a \Uparrow \tau}{\Gamma \vdash a \Downarrow \tau} \qquad \textsc{Annot-I}\ \frac{\Gamma \vdash a \Downarrow \tau}{\Gamma \vdash (a : \tau) \Uparrow \tau} \qquad \textsc{Abs-I}\ \frac{\Gamma, x : \tau_0 \vdash a \Uparrow \tau}{\Gamma \vdash \lambda x{:}\tau_0.\, a \Uparrow \tau_0 \to \tau}$$

Checking mode can use inference mode.

Annotations turn inference mode into checking mode.

Annotations on type abstractions enable the inference mode.

Bidirectional type inference

Simple types

Example: Let τ be $(\tau_1 \to \tau_1) \to \tau_2$. and Γ be $f : \tau$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f \Uparrow \tau}\,\text{Var-I} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, x : \tau_1 \vdash x \Uparrow \tau_1}\,\text{Var-I}}{\Gamma, x : \tau_1 \vdash x \Downarrow \tau_1}\,\text{C-I}}{\Gamma \vdash \lambda x.\, x \Downarrow \tau_1 \to \tau_1}\,\text{Abs-C}}{\Gamma \vdash f\ (\lambda x.\, x) \Uparrow \tau_2}\,\text{App-I}}{\Gamma \vdash f\ (\lambda x.\, x) \Downarrow \tau_2}\,\text{I-C}}{\varnothing \vdash \lambda f{:}\tau.\, f\ (\lambda x.\, x) \Downarrow \tau \to \tau_2}\,\text{Abs-C}$$

Bidirectional type inference

Polymorphic types

The method can be extended to deal with polymorphic types.

The idea is due to [Cardelli, 1993] and is still being improved [Dunfield, 2009]. However, it is quite complicated.

Predicative polymorphism is an interesting subcase where partial type inference can be reduced to typing constraints under a mixed prefix. Unfortunately, predicative polymorphism is too restrictive for programming languages (See [Rémy, 2005]).

A simpler approach proposed by Pierce and Turner [2000] and improved by Odersky et al. [2001] is to perform bidirectional type inference only from a small context surounding each node.

Interestingly, bidirectional type inference can easily be extended to work in the presence of subtyping.

Partial type inference

MLF

MLF follows another approach that amounts to performing first-order unification of higher-order types.

- only parameters of functions that are used polymorphically need to be annotated.
- type abstractions and type annotation are always implicit

However, MLF goes beyond System F: for the purpose of type inference, it introduces richer types that enable to write "more principal types", but that are also harder to read. See [Rémy and Yakobowski, 2008].

The type inference method for MLF can be seen as a generalization of type constraints for ML to handle polymorphic types—still with first-order unification.

Bibliography I

(Most titles have a clickable mark “▷” that links to online versions.)

▷ Richard Bird and Lambert Meertens. Nested datatypes. In *International Conference on Mathematics of Program Construction (MPC)*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

▷ Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33: 309–338, 1998.

Luca Cardelli. An implementation of f¡:. Technical report, DEC Systems Research Center, 1993.

▷ Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. The MLton compiler, 2007.

Bibliography II

Joshua Dunfield. Greedy bidirectional polymorphism. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 15–26, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-509-3. doi: http://doi.acm.org/10.1145/1596627.1596631.

▷ Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2): 253–289, April 1993.

▷ J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

Gérard Huet. *Résolution d'équations dans des langages d'ordre* 1, 2, $\ldots$, ω. PhD thesis, Université Paris 7, September 1976.

▷ Mark P. Jones. Typing Haskell in Haskell. In *Haskell workshop*, October 1999.

Bibliography III

▷ Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, pages 206–220. Springer, May 1990.

▷ Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 382–401, 1990.

▷ David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer, June 2003.

▷ Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

Bibliography IV

- Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, April 1984.
- Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1): 35–55, 1999.
- Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 41–53, 2001.
- Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1999.
- Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Manuscript, April 2004.

Bibliography V

Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: http://doi.acm.org/10.1145/62678.62697.

▷ Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

▷ François Pottier. Notes du cours de DEA "Typage et Programmation", December 2002.

▷ François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

Bibliography VI

▷ François Pottier and Didier Rémy. The essence of ML type inference. Draft of an extended version. Unpublished, September 2003.

▷ Didier Rémy. Simple, partial type-inference for System F based on type-containment. In *Proceedings of the tenth International Conference on Functional Programming*, September 2005.

Didier Rémy and Boris Yakobowski. Efficient Type Inference for the MLF language: a graphical and constraints-based approach. In *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 63–74, Victoria, BC, Canada, September 2008. doi: http://doi.acm.org/10.1145/1411203.1411216.

▷ Christian Skalka and François Pottier. Syntactic type soundness for HM(X). In *Workshop on Types in Programming (TIP)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, July 2002.

Bibliography VII

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

▷ Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

▷ J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.