Where is ML type inference headed?

Constraint solving meets local shape inference

François Pottier

September 2005

INRIA

Types are good

A *type* is a concise description of the behavior of a program fragment.

Typechecking provides *safety* or *security* guarantees.

It also encourages *modularity* and *abstraction*.

Type inference is good

Types can be extremely cumbersome when they have to be explicitly and repeatedly provided.

This leads to (partial or full) *type inference*...

... which is sometimes *hard*, but so... *addictive*.

Constraints are elegant

Type inference problems are naturally *expressed* in terms of *constraints* made up of predicates on types, conjunction, existential and universal quantification, and possibly more.

This allows reducing type inference to *constraint solving*.

Mandatory type annotations can help

Constraint solving can be *intractable* or *undecidable* for some (interesting) type systems.

In that case, *mandatory type annotations* can help. Full type inference is abandoned. In return, the reduction of (now partial) type inference to constraint solving is preserved.

One might wish to go further...

Stratified type inference

Local shape inference can be used to *propagate* type information in *ad hoc* ways through the program and automatically produce some of the required annotations.

This leads to *stratified type inference*, a pragmatic approach to hard type inference problems.

Overview

The talk is planned as follows:

1. Constraint-based type inference for ML
2. Stratified type inference for generalized algebraic data types

Part I

Type inference for ML

The simply-typed λ-calculus

Hindley and Milner's type system

Specification

The simply-typed λ-calculus is specified using a set of rules that allow deriving *judgements*:

Var
$$\Gamma \vdash x : \Gamma(x)$$

Abs
$$\frac{\Gamma; x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

App
$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

The specification is *syntax-directed*.

Substitutions versus constraints

Traditional presentations of type inference are based on substitutions, which means working with *most general unifiers*, *composition*, and *restriction*.

Reasoning in terms of constraints means working with *equations*, *conjunction*, and *existential quantification*.

Let's use the latter.

Constraints

In order to reduce type inference to constraint solving, we introduce a *constraint* language:

$$C ::= \tau = \tau \mid C \wedge C \mid \exists \alpha . C$$

Constraints are *interpreted* by defining when a valuation ϕ *satisfies* a constraint C.

Constraint solving is *first-order unification*.

Constraint generation

Type inference is reduced to constraint solving by defining a mapping $[\![\cdot]\!]$ of *pre-judgements* to constraints.

$$
\begin{array}{rcl}
[\![\Gamma \vdash x : \tau]\!] & = & \Gamma(x) = \tau \\
[\![\Gamma \vdash \lambda x.e : \tau]\!] & = & \exists \alpha_1 \alpha_2.([\![\Gamma; x : \alpha_1 \vdash e : \alpha_2]\!] \wedge \alpha_1 \rightarrow \alpha_2 = \tau) \\
[\![\Gamma \vdash e_1\ e_2 : \tau]\!] & = & \exists \alpha.([\![\Gamma \vdash e_1 : \alpha \rightarrow \tau]\!] \wedge [\![\Gamma \vdash e_2 : \alpha]\!])
\end{array}
$$

Constraints, revisited

How about letting the constraint solver, instead of the constraint generator, deal with *environments?*

Let's enrich the syntax of constraints:

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

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

$$\text{def } x : \tau \text{ in } C \equiv [\tau/x]C$$

The def form is an *explicit substitution* form.

Constraint generation, revisited

Constraint generation is now a mapping of an expression e and a type τ to a constraint $[\![e : \tau]\!]$.

$$
\begin{array}{rcl}
[\![x : \tau]\!] & = & x = \tau \\
[\![\lambda x.e : \tau]\!] & = & \exists \alpha_1 \alpha_2. \left(\begin{array}{l} \mathsf{def}\ x : \alpha_1\ \mathsf{in}\ [\![e : \alpha_2]\!] \\ \alpha_1 \rightarrow \alpha_2 = \tau \end{array} \right) \\
[\![e_1\ e_2 : \tau]\!] & = & \exists \alpha.([\![e_1 : \alpha \rightarrow \tau]\!] \wedge [\![e_2 : \alpha]\!])
\end{array}
$$

Look ma, *no environments!*

The point of introducing the def form will become apparent in Hindley and Milner's type system...

The simply-typed λ-calculus

Hindley and Milner's type system

Specification

Three new typing rules are introduced *in addition* to those of the simply-typed λ-calculus:

Gen

$$\frac{\Gamma \vdash e : \tau \qquad \bar{\alpha} \# \mathrm{ftv}(\Gamma)}{\Gamma \vdash e : \forall \bar{\alpha}.\tau}$$

Inst

$$\frac{\Gamma \vdash e : \forall \bar{\alpha}.\tau}{\Gamma \vdash e : [\vec{\tau}/\vec{\alpha}]\tau}$$

Let

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma; x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : \tau}$$

Type schemes now occur in environments and judgements.

Constraints

Let's extend the syntax of constraints so that a variable x can stand for a *type scheme*.

To avoid mingling constraint generation and constraint solving, we allow type schemes to *carry* constraints.

Turning a constraint into a (constrained) type scheme is then a purely *syntactic* construction—no solving is required.

Constraints, continued

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

$$\begin{array}{lcl} C & ::= & \tau = \tau \mid C \wedge C \mid \exists \alpha.C \mid x \preceq \tau \mid \mathit{def}\ x : \varsigma \text{ in } C \\ \varsigma & ::= & \forall \bar{\alpha}[C].\tau \end{array}$$

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

$$\mathit{def}\ x : \varsigma \text{ in } C \equiv [\varsigma / x]C$$

$$(\forall \bar{\alpha}[C].\tau) \preceq \tau' \equiv \exists \bar{\alpha}.(C \wedge \tau = \tau')$$

Constraint generation

Constraint generation is modified as follows:

$$\begin{aligned} [\![x : \tau]\!] &= x \preceq \tau \\ [\![\text{let } x = e_1 \text{ in } e_2 : \tau]\!] &= \text{def } x : \forall\alpha[[\![e_1 : \alpha]\!]].\alpha \text{ in } [\![e_2 : \tau]\!] \end{aligned}$$

The constrained type scheme $\forall\alpha[[\![e_1 : \alpha]\!]].\alpha$ is *principal* for e_1...

Statement

Theorem (Soundness and completeness)

Let Γ be an environment whose domain is $\mathrm{fv}(e)$. The expression e is well-typed relative to Γ iff

$$\mathit{def}\ \Gamma\ \mathit{in}\ \exists a.[\![e : a]\!]$$

is satisfiable.

Taking constraints seriously

Constraints are suitable for use in an efficient and modular implementation, because:

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

Part II

Generalized algebraic data types

Introducing generalized algebraic data types

Typechecking: MLGI

Simple, constraint-based type inference: MLGX

Local shape inference

Example

Here is *typed* abstract syntax for a simple object language.

$$
\begin{aligned}
Lit &:: \text{int} \rightarrow \text{term int} \\
Inc &:: \text{term int} \rightarrow \text{term int} \\
IsZ &:: \text{term int} \rightarrow \text{term bool} \\
If &:: \forall \alpha.\text{term bool} \rightarrow \text{term } \alpha \rightarrow \text{term } \alpha \rightarrow \text{term } \alpha \\
Pair &:: \forall \alpha\beta.\text{term } \alpha \rightarrow \text{term } \beta \rightarrow \text{term } (\alpha \times \beta) \\
Fst &:: \forall \alpha\beta.\text{term } (\alpha \times \beta) \rightarrow \text{term } \alpha \\
Snd &:: \forall \alpha\beta.\text{term } (\alpha \times \beta) \rightarrow \text{term } \beta
\end{aligned}
$$

This is *not* an ordinary algebraic data type...

Example, continued

This definition allows writing an evaluator that performs no tagging or untagging of object-level values, that is, *no runtime checks*:

$$
\begin{array}{l}
\mu(eval : \forall a.\mathsf{term}\ a \rightarrow a).\lambda t. \\
\quad \mathit{case}\ t\ \mathit{of} \\
\quad\quad \mid \mathit{Lit}\ i \rightarrow (*\ a = \mathsf{int}\ *)\ i \\
\quad\quad \mid \mathit{Inc}\ t \rightarrow (*\ a = \mathsf{int}\ *)\ \mathit{eval}\ t + 1 \\
\quad\quad \mid \mathit{IsZ}\ t \rightarrow (*\ a = \mathsf{bool}\ *)\ \mathit{eval}\ t = 0 \\
\quad\quad \mid \mathit{If}\ b\ t\ e \rightarrow \mathsf{if}\ \ \mathit{eval}\ b\ \ \mathsf{then}\ \ \mathit{eval}\ t\ \ \mathsf{else}\ \ \mathit{eval}\ e \\
\quad\quad \mid \mathit{Pair}\ a\ b \rightarrow (*\ \exists a_1 a_2.a = a_1 \times a_2\ *)\ (\mathit{eval}\ a, \mathit{eval}\ b) \\
\quad\quad \mid \mathit{Fst}\ t \rightarrow \mathit{fst}\ (\mathit{eval}\ t) \\
\quad\quad \mid \mathit{Snd}\ t \rightarrow \mathit{snd}\ (\mathit{eval}\ t)
\end{array}
$$

From type inference to constraint solving

In the presence of generalized algebraic data types, reducing type inference to constraint solving remains reasonably straightforward.

For *eval*, the constraint looks like this, after several simplification steps:

$$\forall a. \begin{pmatrix} a = \mathrm{int} \Rightarrow \mathrm{int} = a \ \text{// Lit} \\ \ldots \\ \forall a_1 a_2 . a = a_1 \times a_2 \Rightarrow a_1 \times a_2 = a \ \text{// Pair} \\ \ldots \end{pmatrix}$$

This eventually simplifies down to true, so *eval* is well-typed.

It looks as if there is *no problem?*

Implications of implication

Adding implication to the constraint language yields the *first-order theory of equality of trees*, whose satisfiability problem is decidable, but *intractable*.

For *eval*, solving seemed easy because enough explicit information was available.

Furthermore, introducing implication means that constraints *no longer have most general unifiers*, as the next example shows...

Implications of implication, continued

What types does this function admit?

$$
\begin{aligned}
&Eq :: \forall a.eq\ a\ a \\
&cast = \\
&\quad \forall a\beta.\lambda(w : eq\ a\ \beta).\lambda(x : a). \\
&\qquad \text{case } w \text{ of} \\
&\qquad\quad Eq \longrightarrow (*\ a = \beta\ *)\ x
\end{aligned}
$$

Implications of implication, continued

All three type schemes below are correct:

$$\forall \alpha\beta.\mathrm{eq}\ \alpha\ \beta \rightarrow \alpha \rightarrow \alpha$$
$$\forall \alpha\beta.\mathrm{eq}\ \alpha\ \beta \rightarrow \alpha \rightarrow \beta$$
$$\forall \gamma.\mathrm{eq}\ \mathrm{int}\ \mathrm{bool} \rightarrow \mathrm{int} \rightarrow \gamma$$

but *none* is principal! The principal *constrained* type scheme produced by constraint solving would be

$$\forall \alpha\beta\gamma[\alpha = \beta \Rightarrow \alpha = \gamma].\mathrm{eq}\ \alpha\ \beta \rightarrow \alpha \rightarrow \gamma$$

which indeed subsumes the previous three.

The system *does not have principal types* in the standard sense.

A solution

I am now about to present a solution where principal types are recovered by means of *mandatory type annotations* and where a *local shape inference* layer is added so as to allow omitting some of these annotations.

This is joint work with Yann Régis-Gianas.

Introducing generalized algebraic data types

Typechecking: MLGI

Simple, constraint-based type inference: MLGX

Local shape inference

MLGI

Let's first define the programs that we deem *sound* and would like to accept, without thinking about type inference.

This is *MLGI*—*ML* with *g*eneralized algebraic data types in *i*mplicit style.

MLGI is Core ML with polymorphic recursion, generalized algebraic data types, and explicit type annotations.

Specification

MLGI's typing judgments take the form

$$E, \Gamma \vdash e : \sigma$$

where E is a *system of type equations*.

Most of the rules are standard, modulo introduction of E...

Specification, continued

E is exploited via *implicit type conversions*:

$$\frac{a = \mathrm{int}, \Gamma \vdash i : \mathrm{int} \qquad a = \mathrm{int} \Vdash \mathrm{int} = a}{a = \mathrm{int}, \Gamma \vdash i : a}$$

The symbol $\Vdash$ stands for *constraint entailment*.

Specification, continued

$$\frac{\mathit{Pair}\ a\ b : \mathit{term}\ a \vdash (a_1 a_2, a = a_1 \times a_2, a : a_1; b : a_2) \qquad a_1 a_2 \# \mathrm{ftv}(\Gamma, a) \qquad a = a_1 \times a_2, (\Gamma; a : a_1; b : a_2) \vdash e : a}{\mathrm{true}, \Gamma \vdash (\mathit{Pair}\ a\ b).e : \mathrm{term}\ a \rightarrow a}$$

Inside each clause, confronting the pattern with the (*actual*) type of the scrutinee yields *new (abstract) type variables*, *type equations*, and *environment entries*.

Determining E and inferring types are interdependent activities...

Introducing generalized algebraic data types

Typechecking: MLGI

Simple, constraint-based type inference: MLGX

Local shape inference

MLGX

Let's require sufficiently many type annotations to ensure that E is *known* at all times, without any guessing. Let's also make all type conversions *explicit*.

This is *MLGX*—*ML* with *g*eneralized algebraic data types in *ex*plicit style.

Specification

$$\frac{E, \Gamma \vdash (e : \text{term } a) : \text{term } a \qquad \forall i \quad E, \Gamma \vdash (p_i : \text{term } a).e_i : \text{term } a \rightarrow a}{E, \Gamma \vdash \mathit{case}\,(e : \text{term } a) \text{ of } p_1.e_1 \ldots p_n.e_n : a}$$

We require a type annotation at *case* constructs and pass it down to the rule that examines individual clauses...

Specification, continued

The rule that checks clauses now exploits the type annotation:

$$\frac{\begin{array}{c} Pair\ a\ b : \text{term}\ a \vdash (a_1 a_2, a = a_1 \times a_2, a : a_1; b : a_2) \\ a_1 a_2 \ \#\ \text{ftv}(\Gamma, a) \qquad a = a_1 \times a_2, (\Gamma; a : a_1; b : a_2) \vdash e : a \end{array}}{\text{true}, \Gamma \vdash (Pair\ a\ b : \text{term}\ a).e : \text{term}\ a \longrightarrow a}$$

The pattern is now confronted with the *type annotation* to determine which new type equations arise. *No guessing* is involved.

Specification, continued

E is now exploited *only* through an *explicit coercion* form:

$$\frac{a = \text{int}, \Gamma \vdash i : \text{int} \qquad a = \text{int} \Vdash \text{int} = a}{a = \text{int}, \Gamma \vdash (i : (\text{int} \triangleright a)) : a}$$

This rule is syntax-directed.

Type inference for MLGX

Type inference for MLGX *decomposes* into two separate tasks:

- *compute* E *everywhere* and *check* that *every explicit coercion* is valid;
- forget E and follow the *standard* reduction to constraint solving. A *coercion* (int $\triangleright$ a) is just a constant *of* type int $\longrightarrow a$.

No implication constraints are involved. MLGX has *principal types*.

In short, MLGX marries *type inference* for Hindley and Milner's type system with *typechecking* for generalized algebraic data types. I believe its design is *robust*.

Programming in MLGX

In MLGX, *eval* is written:

$$
\begin{array}{l}
\mu(eval : \forall a.\mathrm{term}\ a \rightarrow a).\forall a.\lambda t. \\
\quad case\ (t : \mathrm{term}\ a)\ \mathrm{of} \\
\qquad \mid Lit\ i \rightarrow (i : (\mathrm{int} \triangleright a)) \\
\qquad \mid Inc\ t \rightarrow (eval\ t + 1 : (\mathrm{int} \triangleright a)) \\
\qquad \mid IsZ\ t \rightarrow (eval\ t = 0 : (\mathrm{bool} \triangleright a)) \\
\qquad \mid If\ b\ t\ e \rightarrow \mathrm{if}\ \ eval\ b\ \ \mathrm{then}\ \ eval\ t\ \ \mathrm{else}\ \ eval\ e \\
\qquad \mid Pair\ a_1\ a_2\ a\ b \rightarrow ((eval\ a, eval\ b) : (a_1 \times a_2 \triangleright a)) \\
\qquad \mid Fst\ \beta_2\ t \rightarrow fst\ (eval\ t) \\
\qquad \mid Snd\ \beta_1\ t \rightarrow snd\ (eval\ t)
\end{array}
$$

This is nice, but *redundant*... how about some *local shape inference?*

Introducing generalized algebraic data types

Typechecking: MLGI

Simple, constraint-based type inference: MLGX

Local shape inference

Shapes

Shapes are defined by

$$s ::= \bar{\gamma}.\tau$$

The *flexible* type variables $\bar{\gamma}$ (bound within τ) represent *unknown* or *polymorphic* types.

That is, the shape $\gamma.\gamma \rightarrow \gamma$ adequately describes the integer successor function as well as the polymorphic identity function.

This shape is much more precise than $\gamma_1\gamma_2.\gamma_1 \rightarrow \gamma_2$, which describes *any* function.

Shapes, continued

Shapes can have *free* type variables; these are interpreted as *known* types. For instance, the shape

$$\gamma.a \times \gamma$$

describes a pair whose first component has type a, where the type variable a was *explicitly* and *universally* bound by the programmer, and whose second component has unknown type.

Ordering shapes

Shapes are equipped with a standard *instantiation* ordering.

For instance,

$$(\gamma_1.a \times \gamma_1) \preceq (\gamma_2.a \times (a \rightarrow \gamma_2))$$

The *uninformative* shape $\gamma.\gamma$, written $\bot$, is the least element.

Ordering shapes, continued

When two shapes have an upper bound, they have a *least upper bound*, computed via first-order unification.

For instance,

$$(\gamma.\gamma \to \gamma) \sqcup (\gamma.\mathrm{int} \to \gamma) = \mathrm{int} \to \mathrm{int}$$

This allows local shape inference to find that "applying the identity function to an integer yields an integer" — reasoning that requires *instantiation*.

Yet, this use of unification is *local*, because flexible type variables are *never shared* between shapes.

Algorithm Z, judgements

Here is a very rough overview of a shape inference algorithm.

Judgements take the form

$$E, \Gamma \vdash e \downarrow s \uparrow s' \rightsquigarrow e'$$

where Γ (which maps variables to shapes) and s are *provided*, while s' is *inferred* and at least as informative, that is, $s \preceq s'$ holds.

Algorithm Z, mission statement

The transformed term e' is identical to e, except

- *type coercions* are inserted at variables and at case clauses,
- *new type annotations* are inserted around case scrutinees,
- existing type annotations are normalized.

Algorithm Z, in one slide

This is an instance of the rule that deals with clauses:

$$\frac{\ldots \qquad a = a_1 \times a_2, (\Gamma; a : a_1; b : a_2) \vdash e \downarrow a_1 \times a_2 \rightsquigarrow e'}{\begin{array}{l} \text{true}, \Gamma \vdash (Pair\ a_1\ a_2\ a\ b : \text{term}\ a).e \downarrow a \\ \qquad \rightsquigarrow (Pair\ a_1\ a_2\ a\ b).(e' : (a_1 \times a_2 \triangleright a)) \end{array}}$$

The clause is expected to return a value of type a. The equation $a = a_1 \times a_2$ is available inside it. The body e is examined with the *normalized* expected shape $a_1 \times a_2$. We insert an explicit coercion to *let MLGX know* about the equation that we are exploiting.

Programming in MLGX

This explains roughly how the surface language version of *eval* is transformed into:

$$
\begin{array}{l}
\mu^\star(eval : \forall a.\mathrm{term}\ a \longrightarrow a).\lambda t. \\
\quad case\,(t : \mathrm{term}\ a)\ \mathrm{of} \\
\qquad \mid Lit\ i \longrightarrow (i : (\mathrm{int} \triangleright a)) \\
\qquad \mid Inc\ t \longrightarrow (eval\ t + 1 : (\mathrm{int} \triangleright a)) \\
\qquad \mid IsZ\ t \longrightarrow (eval\ t = 0 : (\mathrm{bool} \triangleright a)) \\
\qquad \mid If\ b\ t\ e \longrightarrow \mathrm{if}\ \ eval\ b\ \ \mathrm{then}\ \ eval\ t\ \ \mathrm{else}\ \ eval\ e \\
\qquad \mid Pair\ a_1\ a_2\ a\ b \longrightarrow ((eval\ a, eval\ b) : (a_1 \times a_2 \triangleright a)) \\
\qquad \mid Fst\ a_2\ t \longrightarrow fst\ (eval\ t) \\
\qquad \mid Snd\ a_1\ t \longrightarrow snd\ (eval\ t)
\end{array}
$$

Soundness

Theorem (Soundness for Algorithm Z)

Assume e has type σ in MLGI. If Z infers that e has shape s and rewrites e into e', then $s \preceq \sigma$ holds and e' has type σ in MLGI.

The transformed program *can* be ill-typed in MLGX, but *never* because Z inserted incorrect annotations.

It's still unclear how relevant this theorem is in practice, but I like it.

Part III

Conclusion

Constraint-based type inference

Constraint-based type inference is a *versatile tool* that can deal with many language features while relying on a single constraint solver.

The solver's implementation can be complex, but its behavior remains *predictable* because it is *correct* and *complete* with respect to the logical interpretation of constraints.

Mandatory type annotations

Some constraint languages have *intractable* or *undecidable* satisfiability problems.

Instead of relying on an *incomplete* constraint solver, I suggest modifying the *constraint generation* process so as to take advantage of user-provided *hints*—typically, mandatory type annotations.

Stratified type inference

If the necessary hints are so numerous that they become a burden, a *local shape inference* algorithm can be used to automatically produce some of them.

Although its design is usually *ad hoc*, it should remain predictable if it is sufficiently *simple*.

Thank you.

Some questions

- is stratified type inference *the way of the future*, or a pis aller?
- is local shape inference *really predictable?*
- how do we explain *type errors* in a stratified system?
- can we allow some inferred type information to be *fed back* into shape inference, without losing predictability?

Selected References

François Pottier and Didier Rémy.
The Essence of ML Type Inference.
In *Advanced Topics in Types and Programming Languages*, MIT Press, 2005.

Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich.
Wobbly types: type inference for generalised algebraic data types.
Manuscript, 2004.

François Pottier and Yann Régis-Gianas.
Stratified type inference for generalized algebraic data types.
Manuscript, 2005.