A modern eye on ML type inference

Old techniques and recent developments

François Pottier

September 2005

INRIA

Types

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*.

These aspects are *not* the topic of these lectures.

Type inference

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

This leads to *type inference*...

Constraints

A program fragment is well-typed iff *each* of its own sub-fragments is itself well-typed and if their types are consistent with respect to *one another*.

Thus, a *type inference* problem is naturally *expressed* as a *constraint* made up of predicates about types, conjunction, and existential quantification.

Type annotations and propagation

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

In that case, *mandatory type annotations* can help. Full type inference is abandoned.

If desired, type annotations can be *locally propagated* in ad hoc ways to reduce the number of required annotations.

Overview

The lectures are organized as follows:

1. Type inference for ML (now)
2. Arbitrary-rank polymorphism (this afternoon)
3. Generalized algebraic data types (tomorrow morning)

Part I

Type inference for ML

The simply-typed λ-calculus

Hindley and Milner's type system, with Algorithm $\mathcal{J}$

Hindley and Milner's type system, with constraints

Constraint solving by example

Data structures

Recursion

Optional type annotations

The simply-typed λ-calculus

Hindley and Milner's type system, with Algorithm $\mathcal{J}$

Hindley and Milner's type system, with constraints

Constraint solving by example

Data structures

Recursion

Optional type annotations

Syntax

Expressions are given by

$$e ::= x \mid \lambda x.e \mid e\ e$$

where x denotes a variable. *Types* are given by

$$\tau ::= \alpha \mid \tau \rightarrow \tau$$

where α denotes a type variable.

Specification

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

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

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

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

The specification is *syntax-directed*.

Constraints

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

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

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

A *valuation* is a (total) mapping of the type variables to *ground types*. Ground types form a Herbrand universe, that is, they are finite trees.

Constraint generation

Type inference is reduced to constraint solving by defining a mapping of *pre-judgements* to constraints.

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

The *freshness* conditions are *local*, but left implicit for brevity.

Constraint generation, continued

(Γ, τ) is a *typing* of e iff $\Gamma \vdash e : \tau$ holds.

Theorem (Soundness and completeness)

ϕ is a solution of $[\![\Gamma \vdash e : \tau]\!]$ iff $(\phi\Gamma, \phi\tau)$ is a typing of e.

Principal typings

Corollary (Reduction)

Let (Γ, α) be composed of pairwise distinct type variables. Let the domain of Γ coincide with the free variables of e.

Then, e is typable iff $[\![\Gamma \vdash e : \alpha]\!]$ is satisfiable.

Furthermore, if ϕ is a principal solution of $[\![\Gamma \vdash e : \alpha]\!]$, then $(\phi\Gamma, \phi\alpha)$ is a principal typing of e.

Corollary (Principal typings)

If e is typable, then e admits a principal typing.

A variation on constraints

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

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.

Logical interpretation

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

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

ϕ and ψ satisfy $x = \tau$ iff $\psi x = \phi\tau$ holds.

ϕ and ψ satisfy $\mathsf{def}\ x : \tau\ \mathsf{in}\ C$ iff ϕ and $\psi[x \mapsto \phi\tau]$ satisfy C.

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!*

Constraint generation revisited, continued

Theorem (Reduction)

e is typable iff $[\![e : a]\!]$ is satisfiable.

This statement is marginally simpler than its earlier analogue.

But the true point of introducing the def form becomes apparent only in *Hindley and Milner's type system*...

The simply-typed λ-calculus

Hindley and Milner's type system, with Algorithm $\mathcal{J}$

Hindley and Milner's type system, with constraints

Constraint solving by example

Data structures

Recursion

Optional type annotations

Syntax

ML extends the λ-calculus with a new *let* form:

$$e ::= \ldots \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$$

A *type scheme* is a type where zero or more type variables are universally quantified:

$$\sigma ::= \forall \bar{\alpha}.\tau$$

Specification

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

Gen

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

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.

Milner's Algorithm $\mathcal{J}$

This type inference algorithm expects a *pre-judgement* $\Gamma \vdash e$, produces a *type* τ, and uses two global variables, $\mathcal{V}$ and ϕ.

$\mathcal{V}$ is an (infinite) *fresh name supply*:

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

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

Milner's Algorithm $\mathcal{J}$, continued

Here is the algorithm in monadic style:

$$
\begin{array}{rcl}
\mathcal{J}(\Gamma \vdash x) & = & \text{let } \forall a_1 \ldots a_n.\tau = \Gamma(x) \\
 & & \text{do } a'_1, \ldots, a'_n = \text{fresh}, \ldots, \text{fresh} \\
 & & \text{return } [a'_i / a_i]_{i=1}^{n}(\tau) \text{ — \textit{take a fresh instance}} \\
\mathcal{J}(\Gamma \vdash \lambda x.e_1) & = & \text{do } a = \text{fresh} \\
 & & \text{do } \tau_1 = \mathcal{J}(\Gamma; x : a \vdash e_1) \\
 & & \text{return } a \rightarrow \tau_1 \text{ — \textit{form an arrow type}} \\
 & \ldots &
\end{array}
$$

Milner's Algorithm $\mathcal{J}$, continued

$$
\begin{array}{rcl}
\mathcal{J}(\Gamma \vdash e_1\ e_2) & = & \mathrm{do}\ \tau_1 = \mathcal{J}(\Gamma \vdash e_1) \\
 & & \mathrm{do}\ \tau_2 = \mathcal{J}(\Gamma \vdash e_2) \\
 & & \mathrm{do}\ \alpha = \mathrm{fresh} \\
 & & \mathrm{do}\ \phi \leftarrow \mathrm{mgu}(\phi(\tau_1) = \phi(\tau_2 \rightarrow \alpha)) \circ \phi \\
 & & \mathrm{return}\ \alpha \ \text{— } \textit{solve}\ \ \tau_1 = \tau_2 \rightarrow \alpha \\
\mathcal{J}(\Gamma \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2) & = & \mathrm{do}\ \tau_1 = \mathcal{J}(\Gamma \vdash e_1) \\
 & & \mathrm{let}\ \sigma = \overline{\forall}\,\mathrm{ftv}(\phi(\Gamma)).\phi(\tau_1) \ \text{— } \textit{generalize} \\
 & & \mathrm{return}\ \mathcal{J}(\Gamma; x : \sigma \vdash e_2)
\end{array}
$$

Generation and *solving* of equations are intermixed.

Correctness of algorithm $\mathcal{J}$

Theorem (Correctness)

If $\mathcal{J}(\Gamma \vdash e)$ *terminates in state* (ϕ, V) *and returns* τ, *then* $\phi(\Gamma) \vdash e : \phi(\tau)$ *is a judgement.*

Completeness of algorithm $\mathcal{J}$

Theorem (Completeness)

Let Γ be an environment. Let (ϕ_0, V_0) be a state that satisfies the algorithm's invariant. Let θ_0 and τ_0 be such that $\theta_0\phi_0(\Gamma) \vdash e : \tau_0$ is a judgement. Then, the execution of $\mathcal{J}(\Gamma \vdash e)$ out of the initial state (ϕ_0, V_0) succeeds. Let (ϕ_1, V_1) be its final state and τ_1 be its result. Then, there exists a substitution θ_1 such that $\theta_0\phi_0$ and $\theta_1\phi_1$ coincide outside V_0 and such that τ_0 equals $\theta_1\phi_1(\tau_1)$.

Completeness of algorithm $\mathcal{J}$ (excerpt of proof)

[...] We have

$$\theta_1\phi_1(\gamma) = \theta_1\psi\phi'_2(\gamma) = \theta''_2\phi'_2(\gamma).$$

Since α is fresh for γ and ϕ'_2, we can pursue with

$$\theta''_2\phi'_2(\gamma) = \theta'_2\phi'_2(\gamma) = \theta'_1\phi'_1(\gamma) = \theta_0\phi_0(\gamma).$$

Thus, $\theta_1\phi_1$ and $\theta_0\phi_0$ coincide outside V_0 [...]

Substitutions versus constraints

Reasoning in terms of substitutions means working with *most general unifiers*, *composition*, and *restriction*.

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

Relative typings (terminology)

A typing (Γ', τ) is *relative to Γ* iff its first component Γ' is an instance of Γ.

A typing of e is *principal relative to Γ* iff it is relative to Γ and every typing of e relative to Γ is an instance of it.

Relative principal typings

Corollary (Relative principal typings)

The execution of $\mathcal{J}(\Gamma \vdash e)$ succeeds iff e admits a typing relative to Γ.

Furthermore, if ϕ_1 and τ_1 are the algorithm's results, then $(\phi_1(\Gamma), \phi_1(\tau_1))$ is a typing of e and is principal relative to Γ.

This is also known as the *principal types* property.

The simply-typed λ-calculus

Hindley and Milner's type system, with Algorithm $\mathcal{J}$

Hindley and Milner's type system, with constraints

Constraint solving by example

Data structures

Recursion

Optional type annotations

Constraints

We must 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 must allow type schemes to incorporate constraints.

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

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

Constraints, continued

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

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

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

Interpreting constraints

A type variable α still denotes a ground type.

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

ϕ and ψ satisfy $x \preceq \tau$ iff $\psi x \ni \phi\tau$ holds.

ϕ and ψ satisfy $\mathsf{def}\ x : \varsigma\ \mathsf{in}\ C$ iff ϕ and $\psi[x \mapsto {}^{\psi}_{\phi}(\varsigma)]$ satisfy C.

Interpreting 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.

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 $\forall\beta.(\beta \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma)$. Every constrained type scheme is equivalent to a standard type scheme.

Constraint generation, first attempt

Constraint generation is modified as follows:

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

$\forall \alpha[[\![e_1 : \alpha]\!]].\alpha$ can be thought of as a *principal constrained type scheme* for e_1.

This definition is correct under a *call-by-name* semantics.

Constraint generation, correct attempt

Constraint generation is defined by

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

where, by definition,

$$\text{let } x : \varsigma \text{ in } C \equiv \mathit{def}\ x : \varsigma \text{ in } (\exists \alpha.x \preceq \alpha \wedge C)$$

$[\![\text{let } x = e_1 \text{ in } e_2 : \tau]\!]$ now implies $\exists \alpha.[\![e_1 : \alpha]\!]$, which guarantees that e_1 is well-typed. This definition is correct under a *call-by-value* semantics.

Constraint generation, continued

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* $\mathrm{def}\ \Gamma\ \mathrm{in}\ \exists a.[\![e : a]\!]$ *is satisfiable.*

Taking constraints seriously

Note that

- constraint generation has *linear complexity;*
- constraint generation and constraint solving are *separate.*

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

The constraint language will remain simple as the programming language grows.

The simply-typed λ-calculus

Hindley and Milner's type system, with Algorithm $\mathcal{J}$

Hindley and Milner's type system, with constraints

Constraint solving by example

Data structures

Recursion

Optional type annotations

An initial environment

Let Γ_0 stand for $\mathit{assoc} : \forall\alpha\beta.\alpha \longrightarrow \mathrm{list}\,(\alpha \times \beta) \longrightarrow \beta$.

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

A code fragment

Let e stand for the expression

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

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

The generated constraint

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

$$\exists \alpha\alpha_1\alpha_2\beta.\left(\begin{array}{l} \epsilon = \alpha \longrightarrow \alpha_1 \longrightarrow \alpha_2 \longrightarrow \beta \\ \mathit{def}\ \Gamma\ \mathrm{in} \\ \qquad \mathrm{let}\ \mathit{assocx} : \forall\gamma[\exists\delta.\left(\begin{array}{l} \mathit{assoc} \preceq \delta \longrightarrow \gamma \\ x \preceq \delta \end{array}\right)].\gamma\ \mathrm{in} \\ \qquad \exists\beta_1\beta_2.\left(\begin{array}{l} \beta = \beta_1 \times \beta_2 \\ \forall i \quad \exists\delta.(\mathit{assocx} \preceq \delta \longrightarrow \beta_i \wedge l_i \preceq \delta) \end{array}\right) \end{array}\right)$$

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

$$\text{let } x : \varsigma \text{ in } \mathcal{C}[x \preceq \tau] \equiv \text{let } x : \varsigma \text{ in } \mathcal{C}[\varsigma \preceq \tau]$$

where $\mathcal{C}$ is an arbitrary context.

Simplification, continued

Thus, within the context $\mathsf{def}\ \Gamma_0; \Gamma\ \mathsf{in}\ []$,

$$assoc \preceq \delta \rightarrow \gamma \wedge x \preceq \delta$$

can be rewritten

$$\exists \alpha\beta.(\alpha \rightarrow \mathsf{list}\,(\alpha \times \beta) \rightarrow \beta = \delta \rightarrow \gamma) \wedge \alpha = \delta$$

Simplification, continued

$$\exists\delta.(\exists\alpha\beta.(\alpha \to \mathsf{list}\,(\alpha \times \beta) \to \beta = \delta \to \gamma) \wedge \alpha = \delta)$$

simplifies down to

$$\exists\delta.(\exists\alpha\beta.(\alpha = \delta \wedge \mathsf{list}\,(\alpha \times \beta) \to \beta = \gamma) \wedge \alpha = \delta)$$
$$\exists\delta.(\exists\beta.(\mathsf{list}\,(\delta \times \beta) \to \beta = \gamma) \wedge \alpha = \delta)$$
$$\exists\beta.(\mathsf{list}\,(\alpha \times \beta) \to \beta = \gamma)$$

This is first-order unification.

Simplification, continued

The constrained type scheme

$$\forall\gamma[\exists\delta.(\mathit{assoc} \preceq \delta \rightarrow \gamma \wedge x \preceq \delta)].\gamma$$

is thus equivalent to

$$\forall\gamma[\exists\beta.(\mathrm{list}\,(\alpha \times \beta) \rightarrow \beta = \gamma)].\gamma$$

which can also be written

$$\forall\gamma\beta[\mathrm{list}\,(\alpha \times \beta) \rightarrow \beta = \gamma].\gamma$$
$$\forall\beta.\mathrm{list}\,(\alpha \times \beta) \rightarrow \beta$$

Simplification, continued

The initial constraint has now been simplified down to

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

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

$$\exists\delta.(\mathit{assocx} \preceq \delta \rightarrow \beta_i \wedge l_i \preceq \delta)$$

is rewritten

$$\begin{gathered}\exists\delta.(\exists\beta.(\mathrm{list}\,(\alpha \times \beta) \rightarrow \beta = \delta \rightarrow \beta_i) \wedge \alpha_i = \delta) \\ \exists\beta.(\mathrm{list}\,(\alpha \times \beta) \rightarrow \beta = \alpha_i \rightarrow \beta_i) \\ \exists\beta.(\mathrm{list}\,(\alpha \times \beta) = \alpha_i \wedge \beta = \beta_i) \\ \mathrm{list}\,(\alpha \times \beta_i) = \alpha_i\end{gathered}$$

Simplification, continued

The initial constraint has now been simplified down to

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

Now, the context $\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ \mathit{assocx} : \ldots\ \mathsf{in}\ []$ can be dropped, because the constraint that it applies to contains no occurrences of *assoc*, x, l_1, or l_2.

Simplification, continued

The constraint becomes

$$\exists \alpha\alpha_1\alpha_2\beta.\left(\begin{array}{l} \epsilon = \alpha \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta \\ \exists \beta_1\beta_2.\left(\begin{array}{ll} \beta = \beta_1 \times \beta_2 & \\ \forall i & \mathsf{list}\,(\alpha \times \beta_i) = \alpha_i \end{array}\right) \end{array}\right)$$

that is,

$$\exists \alpha\alpha_1\alpha_2\beta\beta_1\beta_2.\left(\begin{array}{ll} \epsilon = \alpha \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta & \\ \beta = \beta_1 \times \beta_2 & \\ \forall i & \mathsf{list}\,(\alpha \times \beta_i) = \alpha_i \end{array}\right)$$

This is a *solved* form...

Simplification, the end

... and can be written, for better readability,

$$\exists \alpha\beta_1\beta_2.(\epsilon = \alpha \rightarrow \text{list}\,(\alpha \times \beta_1) \rightarrow \text{list}\,(\alpha \times \beta_2) \rightarrow \beta_1 \times \beta_2)$$

This constraint is equivalent to $[\![e : \epsilon]\!]$ under the context def Γ_0 in [].

In other words, the *principal type scheme* of e relative to Γ_0 is

$$\forall \alpha\beta_1\beta_2.\alpha \rightarrow \text{list}\,(\alpha \times \beta_1) \rightarrow \text{list}\,(\alpha \times \beta_2) \rightarrow \beta_1 \times \beta_2$$

Rewriting strategies

Explaining constraint solving in terms of a *small-step rewrite system* makes its correctness and completeness proof easier—it suffices to check that every step is justified by a constraint equivalence law.

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

The simply-typed λ-calculus

Hindley and Milner's type system, with Algorithm $\mathcal{J}$

Hindley and Milner's type system, with constraints

Constraint solving by example

Data structures

Recursion

Optional type annotations

Products and sums

New *type constructors*:

$$\tau ::= \ldots \mid \tau \times \tau \mid \tau + \tau$$

New *term constants*:

$$\begin{array}{rcl}
(\cdot, \cdot) & : & \forall \alpha_1 \alpha_2 . \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_1 \times \alpha_2 \\
\pi_i & : & \forall \alpha_1 \alpha_2 . \alpha_1 \times \alpha_2 \rightarrow \alpha_i \\
\mathsf{inj}_i & : & \forall \alpha_1 \alpha_2 . \alpha_i \rightarrow \alpha_1 + \alpha_2 \\
\mathit{case} & : & \forall \alpha_1 \alpha_2 \alpha . \alpha_1 + \alpha_2 \rightarrow (\alpha_1 \rightarrow \alpha) \rightarrow (\alpha_2 \rightarrow \alpha) \rightarrow \alpha
\end{array}$$

Constraint generation is unaffected.

Recursive types

Products and sums alone do not allow describing *data strutures* of *unbounded size*, such as lists and trees.

Recursive types are required. Two standard approaches: *equi-recursive* and *iso-recursive* types.

Equi-recursive types

New syntax for recursive types:

$$\tau ::= \ldots \mid \mu a.\tau$$

Well-formedness conditions rule out *bad guys* such as $\mu a.a$, whose infinite unfolding isn't well-defined.

We write $\tau_1 =_\mu \tau_2$ if the infinite unfoldings of τ_1 and τ_2 coincide.

Equi-recursive types, continued

The type system is modified by adding just one *conversion* rule:

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 =_\mu \tau_2}{\Gamma \vdash e : \tau_2}$$

The constraint generation rules are *unchanged*, but constraints are now interpreted in a universe of *regular* terms, requiring a simple change to the solver: the "occur-check" is removed.

This approach is simple and powerful, but tends to accept some pieces of code that are really *broken*, that is, do not work as intended...

Iso-recursive types

The user is allowed to introduce new *type constructors* T via (possibly recursive, or even mutually recursive) *declarations*:

$$T\,\vec{a} \approx \tau$$

Each such declaration adds two new *term constants*:

$$\begin{array}{rcl} \mathrm{fold}_T & : & \forall \bar{a}.\tau \rightarrow T\,\vec{a} \\ \mathrm{unfold}_T & : & \forall \bar{a}.T\,\vec{a} \rightarrow \tau \end{array}$$

Constraint generation and constraint solving are unaffected.

Iso-recursive types (example)

Combining structural products and sums with iso-recursive types, one can declare

$$\mathrm{list}\, a \approx \mathrm{unit} + a \times \mathrm{list}\, a$$

Then, the empty list is written

$$\mathrm{fold}_{\mathrm{list}}\, (\mathrm{inj}_1\, ())$$

A list l of type $\mathrm{list}\, a$ is deconstructed by

$$\mathit{case}\ (\mathrm{unfold}_{\mathrm{list}}\, l)\ (\lambda n. \ldots)\ (\lambda c.\mathrm{let}\ hd = \pi_1\, c\ \mathrm{in}\ \mathrm{let}\ tl = \pi_2\, c\ \mathrm{in}\ \ldots)$$

Algebraic data types

In ML, structural products and sums are fused with iso-recursive types, yielding so-called *algebraic data types*.

The idea is to avoid requiring both a (type) *name* and a (field or tag) *number*, as in

$$\mathsf{fold}_{\mathsf{list}}\ (\mathsf{inj}_1\ ())$$

Indeed, this is verbose and fragile. Instead, it would be desirable to mention *a single name*, as in

$$Nil\ ()$$

Declaring an algebraic data type

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

$$T\,\vec{\alpha} \approx \sum_{i=1}^{k} \ell_i : \tau_i \qquad \text{or} \qquad T\,\vec{\alpha} \approx \prod_{i=1}^{k} \ell_i : \tau_i$$

Effects of a record type declaration

The definition

$$T\,\vec{\alpha} \approx \prod_{i=1}^{k} \ell_i : \tau_i$$

introduces the *term constants*

$$\begin{array}{rcll} \ell_i & : & \forall\bar{\alpha}.T\,\vec{\alpha} \rightarrow \tau_i & \qquad i \in \{1, \ldots, k\} \\ \mathsf{make}_T & : & \forall\bar{\alpha}.\tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow T\,\vec{\alpha} & \end{array}$$

In concrete syntax, we write $e.\ell$ for $(\ell\ e)$. When $k > 0$, we write $\{\ell_i = e_i\}_{i=1}^{k}$ for $(\mathsf{make}_T\ e_1\ \ldots\ e_k)$.

Effects of a variant type declaration

The definition

$$T\,\vec{\alpha} \approx \sum_{i=1}^{k} \ell_i : \tau_i$$

introduces the *term constants*

$$\begin{array}{rcll} \ell_i & : & \forall\bar{\alpha}.\tau_i \to T\,\vec{\alpha} & i \in \{1,\ldots,k\} \\ \mathrm{case}_T & : & \forall\bar{\alpha}\gamma.T\,\vec{\alpha} \to (\tau_1 \to \gamma) \to \ldots(\tau_k \to \gamma) \to \gamma & \end{array}$$

In concrete syntax, we write $\mathrm{case}\ e\ [\ell_i : e_i]_{i=1}^{k}$ for $(\mathrm{case}_T\ e\ e_1\ \ldots\ e_n)$ when $k > 0$.

Algebraic data types (example)

One can now declare

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

This gives rise to

$$\begin{array}{rcl} \mathit{Nil} & : & \forall\alpha.\text{unit} \rightarrow \text{list}\,\alpha \\ \mathit{Cons} & : & \forall\alpha.\alpha \times \text{list}\,\alpha \rightarrow \text{list}\,\alpha \\ \mathit{case}_{\text{list}} & : & \forall\alpha\gamma.\text{list}\,\alpha \rightarrow (\text{unit} \rightarrow \gamma) \rightarrow (\alpha \times \text{list}\,\alpha \rightarrow \gamma) \rightarrow \gamma \end{array}$$

Algebraic data types (example, continued)

Then, the empty list is written

$$\mathit{Nil}\ ()$$

A list l of type $\mathrm{list}\, a$ is deconstructed by

$$\begin{array}{l} \mathit{case}\ l\ [\\ \quad\ \mathit{Nil} : \lambda n.\ldots \\ \quad \mid \mathit{Cons} : \lambda c.\mathrm{let}\ \mathit{hd} = \pi_1\, c\ \mathrm{in}\ \mathrm{let}\ \mathit{tl} = \pi_2\, c\ \mathrm{in}\ \ldots \\ ] \end{array}$$

The simply-typed λ-calculus

Hindley and Milner's type system, with Algorithm $\mathcal{J}$

Hindley and Milner's type system, with constraints

Constraint solving by example

Data structures

Recursion

Optional type annotations

fix

Recursion can be introduced via the *term constant*

$$\text{fix} : \forall \alpha\beta.((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)) \rightarrow \alpha \rightarrow \beta$$

This allows defining

$$\text{letrec } f = \lambda x.e_1 \text{ in } e_2$$

as syntactic sugar for

$$\text{let } f = \text{fix } (\lambda f.\lambda x.e_1) \text{ in } e_2$$

Monomorphic recursion

As a result of these definitions, the constraint

$$[\![\text{letrec } f = \lambda x.e_1 \text{ in } e_2 : \tau]\!]$$

is equivalent to

$$\text{let } f : \forall \alpha\beta[\text{let } f : \alpha \rightarrow \beta; x : \alpha \text{ in } [\![e_1 : \beta]\!]].\alpha \rightarrow \beta \text{ in } [\![e_2 : \tau]\!]$$

The variable f is considered *monomorphic* while typechecking e_1. It receives a *polymorphic* type scheme only while typechecking e_2.

Polymorphic recursion

Mycroft suggested extending Hindley and Milner's type system with the rule

$$\frac{\Gamma; f : \sigma \vdash \lambda x.e_1 : \sigma \qquad \Gamma; f : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{letrec } f = \lambda x.e_1 \text{ in } e_2 : \tau}$$

where σ is an arbitrary type scheme.

Polymorphic recursion, continued

Type inference in the presence of polymorphic recursion appears to require *guessing a type scheme*, which first-order unification cannot do.

In fact, the problem is inter-reducible with *semi-unification*, an undecidable problem.

Polymorphic recursion, continued

Yet, type inference in the presence of polymorphic recursion is easy if one is willing to rely on a *mandatory type annotation*.

Let's modify the type system's specification:

$$\frac{\Gamma; f : \sigma \vdash \lambda x.e_1 : \sigma \qquad \Gamma; f : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{letrec } f : \sigma = \lambda x.e_1 \text{ in } e_2 : \tau}$$

so that σ is no longuer guessed.

Polymorphic recursion, continued

Then, define

$$[\![\mathsf{letrec}\ f : \sigma = \lambda x.e_1\ \mathsf{in}\ e_2 : \tau]\!]$$

as

$$\mathsf{let}\ f : \sigma\ \mathsf{in}\ ([\![\lambda x.e_1 : \sigma]\!] \wedge [\![e_2 : \tau]\!])$$

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

There remains to define the new notation $[\![e : \sigma]\!]$...

Universal quantification

It should be intuitively clear that e admits the type scheme $\forall a.a \rightarrow a$ iff e has type $a \rightarrow a$ *for every possible instance of a*, or, equivalently, for an abstract a.

To express this in the constraint language, one introduces *universal quantification*:

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

Its interpretation is standard.

Universal quantification, continued

One can then define

$$[\![e : \forall \bar{\alpha}.\tau]\!]$$

as syntactic sugar for

$$\forall \bar{\alpha}.[\![e : \tau]\!]$$

The need for universal quantification arises when polymorphism is *asserted* by the programmer—as opposed to *inferred* by the system.

The simply-typed λ-calculus

Hindley and Milner's type system, with Algorithm $\mathcal{J}$

Hindley and Milner's type system, with constraints

Constraint solving by example

Data structures

Recursion

Optional type annotations

Optional type annotations

Optional type annotations are useful as a means of *documenting* programs.

Because ML has full type inference, optional type annotations *do not help* accept more programs. Erasing all optional annotations in a well-typed program yields another well-typed program, possibly with a more general type!

Specification

Optional type annotations are introduced by the rule

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e : \tau) : \tau}$$

Here, τ must be a ground type, because we have not (yet) introduced any means of *binding* type variables in expressions.

Constraint generation

It is easy for constraint generation to take optional annotations into account:

$$[\![(e : \tau) : \tau']\!] = [\![e : \tau]\!] \wedge \tau = \tau'$$

It is not difficult to check that this constraint entails

$$[\![e : \tau']\!]$$

which means that the annotation makes the constraint *more specific*.

Introducing type variables

What about *non-ground* type annotations? These make perfect sense, provided the programmer is allowed to *bind type variables*.

A type variable represents an unknown type. But does the programmer mean that the program should be well-typed for *some* or for *all* instances of this variable?

Specification

Two new expression forms allow binding type variables *existentially* or *universally*:

$$\frac{\Gamma \vdash [\tau/a]e : \sigma}{\Gamma \vdash \exists a.e : \sigma} \qquad\qquad \frac{\Gamma \vdash e : \sigma \qquad a \notin \mathrm{ftv}(\Gamma)}{\Gamma \vdash \forall a.e : \forall a.\sigma}$$

Constraint generation

Constraint generation for the first form is straightforward:

$$[\![\exists a.e : \tau]\!] = \exists a.[\![e : \tau]\!]$$

The type annotations inside e can now contain free occurrences of a. Thus, the constraint $[\![e : \tau]\!]$ itself can contain such occurrences. They are given meaning by the existential quantifier.

Constraint generation (example)

For instance, the expression

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

has principal type scheme

$$\forall a.a \rightarrow a \rightarrow a \times a$$

Indeed, the generated constraint contains the pattern

$$\exists a.([\![x_1 : a]\!] \wedge [\![x_2 : a]\!] \wedge \ldots)$$

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

Constraint generation, continued

Constraint generation for the second form is somewhat more subtle. A naïve definition *fails*:

$$[\![\forall\bar{\alpha}.e : \tau]\!] = \forall\bar{\alpha}.[\![e : \tau]\!]$$

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

Constraint generation, continued

One can instead define

$$[\![\forall\bar{a}.e : \tau]\!] = \forall\bar{a}.\exists\gamma.[\![e : \gamma]\!] \wedge \exists\bar{a}.[\![e : \tau]\!]$$

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

The trouble with this definition is that e is duplicated... but this can be avoided with a slight extension of the constraint language (exercise!).

Conclusion of Part I

Type inference for the core of ML and for many of its extensions (not all of which were reviewed here) reduces to *first-order unification under a mixed prefix*.

Some features (such as polymorphic recursion) require *mandatory type annotations*.

Selected References

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

François Pottier.
A modern eye on ML type inference: old techniques and recent developments.
Lecture notes, APPSEM Summer School, 2005.

Part II

Arbitrary-rank polymorphism

The problem

Iso-universal types

Arbitrary-rank predicative polymorphism

The problem

Iso-universal types

Arbitrary-rank predicative polymorphism

A problematic term

Consider the function *apply2*, defined as

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

In Hindley and Milner's type system, f must receive a *monomorphic* type. This leads to the type scheme

$$\forall \alpha\beta.(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \alpha \rightarrow \beta \times \beta$$

where x and y must have identical type.

A problematic term, continued

But perhaps

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

was intended? Or perhaps

$$\forall \alpha\beta\delta.(\forall \gamma.\gamma \to \delta) \to \alpha \to \beta \to \delta \times \delta$$

was the desired type? Or perhaps

$$\forall \alpha\beta\delta.(\forall \gamma.\gamma \to \gamma \times \delta) \to \alpha \to \beta \to (\alpha \times \delta) \times (\beta \times \delta),$$

or perhaps, or perhaps...

System F

None of these types are instances of one another. System F, where each of these types is valid, *does not* have the *principal types* property.

In fact, type inference for System F is *undecidable*...

Rank-1 polymorphism

This explains why Hindley and Milner's type system is restricted to *rank-1* polymorphism.

The need for higher-rank polymorphism is mitigated by features such as ML's *module system* and Haskell's *type classes*, but these workarounds are not always convenient.

The problem

Iso-universal types

Arbitrary-rank predicative polymorphism

Iso-universal types

A simple route for introducing arbitrary-rank polymorphism into Hindley and Milner's type system, without compromising type inference, is to follow the *up-to-explicit-isomorphism* approach that was used for recursive types.

Specification

This leads to *"iso-universal"* types that require an explicit declaration:

$$T\,\vec{\alpha} \approx \forall\bar{\beta}.\tau$$

One would like each such declaration to add two new term constants:

$$\begin{array}{rcl} \mathsf{fold}_T & : & \forall\bar{\alpha}.(\forall\bar{\beta}.\tau) \rightarrow T\,\vec{\alpha} \\ \mathsf{unfold}_T & : & \forall\bar{\alpha}.T\,\vec{\alpha} \rightarrow \forall\bar{\beta}.\tau \end{array}$$

But these *aren't* valid type schemes...

Specification, continued

Though fold_T cannot be viewed as a constant, it can be introduced as a new construct:

$$\frac{\Gamma \vdash e : [\vec{\tau}/\vec{\alpha}](\forall \bar{\beta}.\tau)}{\Gamma \vdash \mathsf{fold}_T\ e : T\ \vec{\tau}}$$

Constraint generation is as follows:

$$[\![\mathsf{fold}_T\ e : \tau']\!] = \exists \bar{\alpha}.([\![e : \forall \bar{\beta}.\tau]\!] \wedge T\ \vec{\alpha} = \tau')$$

Specification, continued

Though

$$\mathsf{unfold}_T : \forall\bar\alpha.T\,\vec\alpha \longrightarrow \forall\bar\beta.\tau$$

doesn't literally make sense,

$$\mathsf{unfold}_T : \forall\bar\alpha\bar\beta.T\,\vec\alpha \longrightarrow \tau$$

does, and achieves the desired effect.

Example

For instance, one can declare

$$Id \approx \forall \gamma.\gamma \rightarrow \gamma$$

and modify *apply2* as follows:

$$\lambda f.\lambda x.\lambda y.\mathrm{let}\ f = \mathrm{unfold}_{Id}\ f\ \mathrm{in}\ (f\ x, f\ y)$$

It is invoked by

$$apply2\ (\mathrm{fold}_{Id}\ (\lambda x.x))$$

Summary

Uses of fold_T and unfold_T can be viewed as *mandatory type annotations* and indicate where type abstraction and type application should be performed.

“Iso-universal” types are usually fused with algebraic data types, so that type abstraction and application are performed at data construction and deconstruction time.

The same technique can be applied to existential types.

Limitations

This mechanism allows encoding all System F programs.

However, it is somewhat *verbose*. Furthermore, the encoding is *non-modular*. A single System F type can have many distinct encodings, and one must explicitly convert between them.

The problem

Iso-universal types

Arbitrary-rank predicative polymorphism

An idea

How about relying on mandatory type annotations, *without* going through the detour of declaring iso-universal types?

For instance, for this version of *apply2*:

$$\lambda f : \forall \gamma.\gamma \rightarrow \gamma.\lambda x.\lambda y.(f\ x, f\ y)$$

it should not be difficult to infer the type

$$\forall \alpha\beta.(\forall \gamma.\gamma \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \alpha \times \beta$$

should it?

An idea, continued

The idea is to allow arbitrary-rank polymorphic types *when mandated by an explicit type annotation*.

One establishes the convention—known as *predicativity*—that *type variables stand for monotypes*. In terms of type inference, this means that *polymorphic types are never inferred*.

I am now about to present an approach that draws on ideas by Läufer and Odersky, Peyton Jones and Shields, and Rémy.

Syntax

$$\begin{array}{rcl} \tau & ::= & a \mid \tau \rightarrow \tau \\ \rho & ::= & a \mid \sigma \rightarrow \sigma \\ \sigma & ::= & \forall \bar{a}.\rho \end{array}$$

τ ranges over *monotypes*. σ ranges over *polytypes*. ρ ranges over the subset of polytypes that have no outermost quantifiers.

Syntax, continued

$$\begin{array}{rcl} \theta & ::= & \exists\bar{\alpha}.\sigma \\ e & ::= & x \mid \lambda x.e \mid e\;(e : \theta) \mid \mathsf{let}\; x = (e : \theta)\;\mathsf{in}\; e \end{array}$$

Application and let nodes must now be *annotated* so that terms can be *typechecked* (top-down) *without guessing a polytype*.

Type annotations can be made optional by interpreting the absence of an annotation as the *trivial annotation* $\exists\beta.\beta$, which means *"any monomorphic type"*.

Specification: typing rules

$$\Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \lambda x.e : \sigma_1 \to \sigma_2}$$

$$\frac{\Gamma \vdash e_1 : [\vec{\tau}/\vec{\beta}]\sigma_2 \to \sigma_1 \qquad \Gamma \vdash e_2 : [\vec{\tau}/\vec{\beta}]\sigma_2}{\Gamma \vdash e_1\ (e_2 : \exists\bar{\beta}.\sigma_2) : \sigma_1}$$

$$\frac{\Gamma \vdash e_1 : \forall\bar{a}.[\vec{\tau}/\vec{\beta}]\sigma_1 \qquad \Gamma, x : \forall\bar{a}.[\vec{\tau}/\vec{\beta}]\sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \text{let } x = (e_1 : \exists\bar{\beta}.\sigma_1) \text{ in } e_2 : \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma \qquad a \notin \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall a.\sigma} \qquad \frac{\Gamma \vdash e : \sigma' \qquad \sigma' \leq \sigma}{\Gamma \vdash e : \sigma}$$

Specification: type containment

A subset, identified by Läufer and Odersky, of the type containment relation studied by Mitchell for System F^η.

$$a \leq a \qquad \frac{\sigma_1' \leq \sigma_1 \qquad \sigma_2 \leq \sigma_2'}{\sigma_1 \rightarrow \sigma_2 \leq \sigma_1' \rightarrow \sigma_2'} \qquad \frac{\begin{array}{c}\sigma \leq \sigma' \\ a \notin \mathrm{ftv}(\sigma)\end{array}}{\sigma \leq \forall a.\sigma'} \qquad \frac{[\tau/a]\sigma \leq \rho}{(\forall a.\sigma) \leq \rho}$$

Here, instantiation is *predicative*. Furthermore, Mitchell's axiom

$$\forall a.\sigma \rightarrow \sigma' \leq (\forall a.\sigma) \rightarrow \forall a.\sigma'$$

is *not* included.

Comments

It is fairly clear that the type system is *sound*, since it can be embedded within System F^{η}.

It is not so clear at first how to perform type inference, since the system has *two non-syntax-directed rules*, but a syntax-directed reformulation exists...

Syntax-directed specification

$$\frac{\Gamma(x) \leq \rho}{\Gamma \vdash x : \rho} \qquad \frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \lambda x.e : \sigma_1 \rightarrow \sigma_2} \qquad \frac{\Gamma \vdash e : \sigma \qquad a \notin \mathrm{ftv}(\Gamma)}{\Gamma \vdash e : \forall a.\sigma}$$

$$\frac{\Gamma \vdash e_1 : [\vec{\tau}/\vec{\beta}]\sigma_2 \rightarrow \rho_1 \qquad \Gamma \vdash e_2 : [\vec{\tau}/\vec{\beta}]\sigma_2}{\Gamma \vdash e_1\ (e_2 : \exists\bar{\beta}.\sigma_2) : \rho_1}$$

$$\frac{\Gamma \vdash e_1 : [\vec{\tau}/\vec{\beta}]\sigma_1 \qquad \bar{a} \# \mathrm{ftv}(\Gamma) \qquad \Gamma, x : \forall\bar{a}.[\vec{\tau}/\vec{\beta}]\sigma_1 \vdash e_2 : \rho_2}{\Gamma \vdash \mathrm{let}\ x = (e_1 : \exists\bar{\beta}.\sigma_1)\ \mathrm{in}\ e_2 : \rho_2}$$

Comments

The rules on the previous slide can be read as an algorithm by viewing the type in the conclusion as an *input*—an expected type.

Then, one can check that *only monotypes are guessed*. Explicit annotations are required at every node where a polytype would otherwise have to be guessed.

When all type annotations are omitted (that is, $\exists\beta.\beta$), the specification coincides with that of Hindley and Milner's type system. Thus, this is a *conservative extension*.

Constraints

Let's extend the syntax of constraints to allow for polytypes.

$$\begin{array}{lcl} C & ::= & \tau = \tau \mid \sigma \leq \sigma \mid C \wedge C \mid \exists a.C \mid \forall a.C \mid x \preceq \sigma \mid \mathrm{def}\ x : \varsigma\ \mathrm{in}\ C \\ \varsigma & ::= & \forall \bar{a}[C].\sigma \end{array}$$

Type variables still denote *monotypes*, so the constraint solver is essentially unchanged, except it now needs to reduce *ordering constraints* $\sigma_1 \leq \sigma_2$...

Reducing ordering constraints

This reduction is due to Läufer and Odersky.

$$
\begin{array}{rcl}
\tau \leq \tau' & \rightarrow & \tau = \tau' \\
\sigma_1 \rightarrow \sigma_2 \leq a & \rightarrow & \exists a_1 a_2 . \left(\begin{array}{l} \sigma_1 \rightarrow \sigma_2 \leq a_1 \rightarrow a_2 \\ a_1 \rightarrow a_2 = a \end{array} \right) \\
a \leq \sigma_1 \rightarrow \sigma_2 & \rightarrow & \exists a_1 a_2 . \left(\begin{array}{l} a_1 \rightarrow a_2 \leq \sigma_1 \rightarrow \sigma_2 \\ a = a_1 \rightarrow a_2 \end{array} \right) \\
\sigma_1 \rightarrow \sigma_2 \leq \sigma'_1 \rightarrow \sigma'_2 & \rightarrow & \sigma'_1 \leq \sigma_1 \wedge \sigma_2 \leq \sigma'_2 \\
(\forall a.\sigma) \leq \rho & \rightarrow & \exists a.(\sigma \leq \rho) \\
\sigma \leq \forall a.\sigma' & \rightarrow & \forall a.(\sigma \leq \sigma')
\end{array}
$$

Constraint generation

$$\begin{array}{rcl}
[\![x : \rho]\!] & = & x \preceq \rho \\
[\![\lambda x.e : a]\!] & = & \exists a_1 a_2. \left(\begin{array}{l} [\![\lambda x.e : a_1 \rightarrow a_2]\!] \\ a_1 \rightarrow a_2 = a \end{array} \right) \\
[\![\lambda x.e : \sigma_1 \rightarrow \sigma_2]\!] & = & \text{let } x : \sigma_1 \text{ in } [\![e : \sigma_2]\!] \\
[\![e_1\ (e_2 : \exists\bar{\beta}.\sigma_2) : \rho_1]\!] & = & \exists\bar{\beta}. \left(\begin{array}{l} [\![e_1 : \sigma_2 \rightarrow \rho_1]\!] \\ [\![e_2 : \sigma_2]\!] \end{array} \right) \\
[\![\text{let } x = (e_1 : \exists\bar{\beta}.\sigma_1) \text{ in } e_2 : \rho_2]\!] & = & \text{let } x : \forall\bar{\beta}[[\![e_1 : \sigma_1]\!]].\sigma_1 \text{ in} \\
& & [\![e_2 : \rho_2]\!] \\
[\![e : \forall a.\sigma]\!] & = & \forall a.[\![e : \sigma]\!]
\end{array}$$

Dealing with side effects

In the presence of the *value restriction*, the syntax-directed typing rules and the constraint generation rules are slightly different. Some details change, but the general ideas are the same.

Limitation

In the calculus that we just studied, one can write:

$$\begin{array}{l} \text{let } \mathit{apply2} = \\ \quad (\lambda f.\lambda x.\lambda y.(f\ x, f\ y) \\ \qquad : \exists\delta.(\forall\gamma.\gamma \longrightarrow \gamma) \longrightarrow \delta) \text{ in} \\ \mathit{apply2}\ (\lambda z.z : \forall\gamma.\gamma \longrightarrow \gamma) \end{array}$$

and let the system infer that this expression has type

$$\forall\alpha\beta.\alpha \longrightarrow \beta \longrightarrow \alpha \times \beta$$

This is nice, but *redundant*: the type scheme for f is given twice.

Introducing local propagation

The two annotations are *so clearly* related to one another that only one annotation should suffice...

Peyton Jones and Shields suggested enhancing the system with the ability of *locally propagating polymorphic types* so as to reduce the amount of necessary annotations.

Local propagation—also known as *elaboration* or *local inference*—can be viewed as a preprocessing step that turns a program expressed in a surface language into a program expressed in the redundant core calculus.

The surface language

In the surface language, we allow function parameters to carry a type annotation, and conversely, allow application and let nodes to carry *no* annotation.

$$e ::= \ldots \mid \lambda x : \theta . e \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$$

Here, the absence of an annotation is *not* interpreted as the trivial annotation $\exists \beta . \beta$. Indeed, local type inference might be able to supply a nontrivial annotation.

Shapes

Local inference is *not concerned with monotypes* at all, since traditional type inference for the core calculus is perfectly capable of finding out about them.

Local inference deals with *shapes*, which by definition are *closed* polytypes extended with a special constant #.

Shapes, continued

A type σ is turned into a shape $\lceil \sigma \rceil$ by replacing all of its free variables with # and exploiting the equation $\# \rightarrow \# = \#$.

For instance,

$$\begin{array}{rl} & \lceil \forall \alpha_1.(\forall \alpha_2.(\alpha_1 \rightarrow \alpha_2) \rightarrow (\beta_0 \rightarrow \beta_0)) \rightarrow (\beta_1 \rightarrow \beta_2) \rceil \\ = & \forall \alpha_1.(\forall \alpha_2.(\alpha_1 \rightarrow \alpha_2) \rightarrow \#) \rightarrow \# \end{array}$$

Shapes, continued

A type annotation is turned into a shape by setting

$$\lceil \exists \bar{\beta}.\sigma \rceil = \lceil \sigma \rceil$$

Conversely, a shape S is turned into a type annotation by replacing each occurrence of # with a distinct type variable and by existentially quantifying these type variables up front.

For instance,

$$\begin{array}{rl} & \lfloor \forall \alpha_1.(\forall \alpha_2.(\alpha_1 \rightarrow \alpha_2) \rightarrow \#) \rightarrow \# \rfloor \\ = & \exists \beta_1 \beta_2.\forall \alpha_1.(\forall \alpha_2.(\alpha_1 \rightarrow \alpha_2) \rightarrow \beta_1) \rightarrow \beta_2 \end{array}$$

Shapes, continued

Last, instantiating the root quantifiers of a shape S with $\#$ yields a new shape $S^\flat$.

For instance,

$$\begin{aligned} & (\forall \alpha_1.(\forall \alpha_2.\alpha_2 \to \alpha_1 \to \alpha_1) \to \#)^\flat \\ = \quad & (\forall \alpha_2.\alpha_2 \to \#) \to \# \end{aligned}$$

Design of a shape inference algorithm

We are now ready to design an algorithm that *infers shapes* and uses them to produce an *annotated* program in the core calculus.

The design is necessarily *ad hoc*, and aims for *simplicity* and *predictability*.

Bidirectionality

Following a long tradition, shape inference operates in one of two modes: *synthesis* and *checking*.

$\Gamma \vdash_{\uparrow} e : S \Rightarrow e'$ synthesis, S is *inferred*

$\Gamma \vdash_{\downarrow} e : S \Rightarrow e'$ checking, S is *provided*

The two judgements are defined in a mutually recursive way.

e is a surface language expression, while e' is a core language expression. Γ maps variables to shapes.

Specification

Each construct in the surface language comes in two flavors: *annotated or unannotated*, and can be examined in two modes: *synthesis or checking*.

That's a lot of rules... let's look at just a few.

Specification, continued

Unannotated abstraction, synthesis mode:

$$\frac{\Gamma, x : \# \vdash_{\uparrow} e : S \Rightarrow e'}{\Gamma \vdash_{\uparrow} \lambda x.e : \# \rightarrow S \Rightarrow \lambda x.e'}$$

The shape of the argument is unknown. No annotation is produced.

Specification, continued

Annotated abstraction, synthesis mode:

$$\frac{\Gamma, x : \lceil\sigma\rceil \vdash_{\uparrow} e : S \Rightarrow e'}{\begin{array}{c}\Gamma \vdash_{\uparrow} \lambda(x : \exists\bar{\beta}.\sigma).e : \lceil\sigma\rceil \rightarrow S \\ \Rightarrow \lambda x.\mathrm{let}\ x = (x : \exists\bar{\beta}.\sigma)\ \mathrm{in}\ e'\end{array}}$$

The shape of the argument is found in the existing annotation. *An annotation is produced* so as to allow x to have nontrivial shape in the core calculus.

Specification, continued

Unannotated let definition, either mode:

$$\frac{\Gamma \vdash_{\uparrow} e_1 : S_1 \Rightarrow e'_1 \qquad \Gamma, x : S_1 \vdash_{\updownarrow} e_2 : S_2 \Rightarrow e'_2}{\Gamma \vdash_{\updownarrow} \text{let } x = e_1 \text{ in } e_2 : S_2 \Rightarrow \text{let } x = (e'_1 : \lfloor S_1 \rfloor) \text{ in } e'_2}$$

$\updownarrow$ stands for one of $\uparrow$ and $\downarrow$.

The synthesized shape S_1 is used when examining the right-hand side. *No generalization is performed*, since shapes do not contain type variables.

An annotation is produced so as to allow x to have nontrivial shape in the core calculus.

Specification, continued

Unannotated application, synthesis mode:

$$\frac{\Gamma \vdash_{\uparrow} e_1 : S \Rightarrow e_1' \qquad S^{\flat} = S_2 \rightarrow S_1 \qquad \Gamma \vdash_{\downarrow} e_2 : S_2 \Rightarrow e_2'}{\Gamma \vdash_{\uparrow} e_1\; e_2 : S_1 \Rightarrow e_1'\; (e_2' : \lfloor S_2 \rfloor)}$$

The function's shape is *synthesized* and its domain shape is used to examine the argument in *checking* mode.

An annotation is produced so as to allow the argument to have nontrivial shape in the core calculus.

Example

In the surface language, one can write:

$$\begin{array}{l}\text{let } \mathit{apply2} = \\ \quad \lambda(f : \forall\gamma.\gamma \rightarrow \gamma).\lambda x.\lambda y.(f\ x, f\ y) \text{ in} \\ \mathit{apply2}\ (\lambda z.z)\end{array}$$

The system finds that *apply2* has shape

$$(\forall\gamma.\gamma \rightarrow \gamma) \rightarrow \#$$

This in turn allows determining that $\lambda z.z$ should have shape

$$(\forall\gamma.\gamma \rightarrow \gamma)$$

A core calculus term with *two annotations* is produced—the one we saw before.

Are we happy?

In the surface language, one can define and use functions with arbitrary-rank polymorphic types, modulo a reasonable amount of *explicit type annotations*, and without giving up *Hindley-Milner type inference*.

So? ...

Predicativity

This is a *predicative* type system. For instance, if *id* has type

$$\forall \alpha.\alpha \rightarrow \alpha$$

then it *cannot* be implicitly coerced to type

$$(\forall \gamma.\gamma \rightarrow \gamma) \rightarrow (\forall \gamma.\gamma \rightarrow \gamma)$$

Predicativity, continued

Indeed,

$$(\forall a.a \rightarrow a) \leq (\forall \gamma.\gamma \rightarrow \gamma) \rightarrow (\forall \gamma.\gamma \rightarrow \gamma)$$

simplifies down to

$$\exists a.(a \rightarrow a \leq (\forall \gamma.\gamma \rightarrow \gamma) \rightarrow (\forall \gamma.\gamma \rightarrow \gamma))$$

$$\exists a.\left(\begin{array}{c} (\forall \gamma.\gamma \rightarrow \gamma) \leq a \\ a \leq (\forall \gamma.\gamma \rightarrow \gamma) \end{array}\right)$$

$$\exists a.\left(\begin{array}{c} \exists \gamma.\gamma \rightarrow \gamma = a \\ \forall \gamma.a = \gamma \rightarrow \gamma \end{array}\right)$$

false

Re-introducing impredicativity

In practice, impredicativity is *essential*.

The easiest way of re-introducing it is via an *explicit impredicative instantiation* construct. This is heavy, though.

One can resort to *more local inference* to guess where impredicative instantiation is required...

... or, more ambitiously, forget about local inference and *build* some measure of *impredicative instantiation into the constraint language*.

Selected References I

- Martin Odersky and Konstantin Läufer.
  Putting Type Annotations To Work.
  POPL, 1996.

- Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields.
  Practical type inference for arbitrary-rank types.
  Manuscript, 2005.

- Didier Rémy.
  Simple, partial type inference for System F based on type containment.
  ICFP, 2005.

Selected References II

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones.
Boxy types: type inference for higher-rank types and impredicativity.
Manuscript, 2005.

Didier Le Botlan and Didier Rémy.
MLF: Raising ML to the power of System *F*.
ICFP, 2003.

Part III

Generalized algebraic data types

Introducing generalized algebraic data types

Typechecking: MLGI

Simple type inference: MLGX

Shape inference

Introducing generalized algebraic data types

Typechecking: MLGI

Simple type inference: MLGX

Shape inference

Algebraic data types (reminder)

The data constructors associated with an *ordinary* algebraic data type constructor T receive type schemes of the form:

$$K :: \forall \bar{a}.\tau_1 \times \ldots \times \tau_n \rightarrow T\ \bar{a}$$

For instance,

$$\begin{aligned} \mathsf{Leaf} &:: \forall a.\mathsf{tree}(a) \\ \mathsf{Node} &:: \forall a.\mathsf{tree}(a) \times a \times \mathsf{tree}(a) \rightarrow \mathsf{tree}(a) \end{aligned}$$

Matching a value of type $\mathsf{tree}(a)$ against the pattern $\mathsf{Node}(l, v, r)$ binds l, v, and r to values of types $\mathsf{tree}(a)$, a, and $\mathsf{tree}(a)$.

Iso-existential types

In Läufer and Odersky's extension of Hindley and Milner's type system with iso-existential types, the data constructors receive type schemes of the form:

$$K :: \forall \bar{\alpha}\bar{\beta}.\tau_1 \times \ldots \times \tau_n \rightarrow T\,\bar{\alpha}$$

For instance,

$$\mathsf{Key} :: \forall \beta.\beta \times (\beta \rightarrow \mathsf{int}) \rightarrow \mathsf{key}$$

Matching a value of type key against the pattern Key (v, f) binds v and f to values of type β and $\beta \rightarrow \mathsf{int}$, *for an unknown* β.

Generalized algebraic data types

Let us now go further and remove the restriction that the parameters to T should be distinct type variables:

$$K :: \forall\bar{\beta}.\tau_1 \times \ldots \times \tau_n \rightarrow T\,\bar{\tau}$$

Instead, they can be arbitrary types, with $\mathrm{ftv}(\bar{\tau}) \subseteq \bar{\beta}$.

Matching a value of type $T\,\bar{a}$ against the pattern $K\,(x_1, \ldots, x_n)$ binds x_i to a value of type τ_i, *for some unknown types $\bar{\beta}$ that satisfy the constraint $\bar{\tau} = \bar{a}$*.

Generalized algebraic data types, continued

With generalized algebraic data types, pattern matching introduces *new type equations* that can be exploited to establish well-typedness.

In other words, the success of a *dynamic* test can yield extra *static* type information.

Generalized algebraic data types are very much like the *inductive types* found in theorem provers, but have only recently received interest in the programming languages community.

Applications

Applications of generalized algebraic data types include:

- *Generic programming* (Xi, Cheney and Hinze)
- *Typed meta-programming* (Pfenning and Lee, Xi, Sheard)
- *Tagless automata* (Pottier and Régis-Gianas)
- Type-preserving *defunctionalization* (Pottier and Gauthier)
- and more...

Example

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

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

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

Example, continued

This definition allows writing an evaluator that does not perform any tagging or untagging of object-level values:

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

From type inference to constraint solving

In the presence of generalized algebraic data types, reducing type inference to constraint solving remains reasonably straightforward (Simonet and Pottier, Stuckey and Sulzmann).

For *eval*, the constraint could look like this...

Example, continued

$$
\begin{array}{l}
\text{let } \mathit{eval} : \forall a.\text{term } a \longrightarrow a \text{ in}\\
\quad \forall a.\\
\qquad \text{let } t : \text{term } a \text{ in}\\
\qquad \exists\beta. \left(\begin{array}{l}
t \preceq \text{term } \beta\\
\beta = \text{int} \Rightarrow\\
\quad \text{let } i : \text{int in } i \preceq a\\
\ldots\\
\forall a_1 a_2.\beta = a_1 \times a_2 \Rightarrow\\
\quad \text{let } a : \text{term } a_1 ; b : \text{term } a_2 \text{ in}\\
\quad \exists \beta_1\beta_2.\left(\begin{array}{l}
\exists\gamma_1.(\mathit{eval} \preceq \gamma_1 \longrightarrow \beta_1 \wedge a \preceq \gamma_1)\\
\exists\gamma_2.(\mathit{eval} \preceq \gamma_2 \longrightarrow \beta_2 \wedge b \preceq \gamma_2)\\
\beta_1 \times \beta_2 = a
\end{array}\right)
\end{array}\right)
\end{array}
$$

Example, continued

$$
\begin{array}{l}
\text{let } eval : \forall a.\text{term } a \rightarrow a \text{ in} \\
\quad \forall a. \\
\quad\quad \left(\begin{array}{l}
a = \text{int} \Rightarrow \\
\quad \text{let } i : \text{int in } i \preceq a \\
\ldots \\
\forall a_1 a_2 . a = a_1 \times a_2 \Rightarrow \\
\quad \text{let } a : \text{term } a_1 ; b : \text{term } a_2 \text{ in} \\
\quad \exists \beta_1 \beta_2 . \left(\begin{array}{l}
\exists \gamma_1 .(eval \preceq \gamma_1 \rightarrow \beta_1 \wedge a \preceq \gamma_1) \\
\exists \gamma_2 .(eval \preceq \gamma_2 \rightarrow \beta_2 \wedge b \preceq \gamma_2) \\
\beta_1 \times \beta_2 = a
\end{array} \right)
\end{array} \right)
\end{array}
$$

Example, continued

$$\forall\alpha. \left(\begin{array}{l} \alpha = \mathrm{int} \Rightarrow \mathrm{int} = \alpha \\ \ldots \\ \forall\alpha_1\alpha_2.\alpha = \alpha_1 \times \alpha_2 \Rightarrow \\ \qquad \exists\beta_1\beta_2. \left(\begin{array}{l} \alpha_1 = \beta_1 \\ \alpha_2 = \beta_2 \\ \beta_1 \times \beta_2 = \alpha \end{array} \right) \end{array} \right)$$

Example, continued

$$\forall a.\left(\begin{array}{l}\mathrm{true}\\ \ldots\\ \forall a_1 a_2 . a = a_1 \times a_2 \Rightarrow a_1 \times a_2 = a\end{array}\right)$$

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

Huh?

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 was easy because enough explicit information was available.

This is not just a matter of computing power. These constraints do not have nice *solved forms*...

Implications of implication, continued

What types does this function admit?

$$Eq :: \forall \alpha. eq\ \alpha\ \alpha$$

```
$cast =$
    $\forall \alpha\beta.\lambda(w : eq\ \alpha\ \beta).\lambda(x : \alpha).$
        $\text{case } w \text{ of}$
            $Eq \longrightarrow (*\ \alpha = \beta\ *)\ x$
```

Implications of implication, continued

Both of these are correct:

$$\forall \alpha\beta.\mathit{eq}\ \alpha\ \beta \rightarrow \alpha \rightarrow \alpha$$
$$\forall \alpha\beta.\mathit{eq}\ \alpha\ \beta \rightarrow \alpha \rightarrow \beta$$

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

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

which indeed subsumes the previous two.

Implications of implication, continued

The constraint

$$\alpha = \beta \Rightarrow \alpha = \gamma$$

cannot be further simplified; it is a solved form.

Introducing implication means that constraints *no longer have most general unifiers*. In other words, the system *no longer has 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 and draws inspiration on work by Peyton Jones, Washburn, and Weirich.

Introducing generalized algebraic data types

Typechecking: MLGI

Simple type inference: MLGX

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.

Data constructor declarations

Every data constructor K is assigned a closed type scheme by a declaration of the form

$$K :: \forall\bar{\alpha}\bar{\beta}.\tau_1 \times \ldots \times \tau_n \rightarrow T\ \bar{\alpha}\ \bar{\tau}$$

$\bar{\alpha}$ are *ordinary* parameters, while $\bar{\tau}$ are *generalized* parameters.

Types

$$\begin{array}{rl} \text{Types} & \tau ::= a \mid \tau \to \tau \mid T\,\bar{\tau}\,\bar{\tau} \\ \text{Type schemes} & \sigma ::= \forall \bar{a}.\tau \\ \text{Simple type annotations} & \theta ::= \exists \bar{\gamma}.\tau \\ \text{Polymorphic type annotations} & \varsigma ::= \exists \bar{\gamma}.\sigma \end{array}$$

Each type annotation binds its own *flexible* type variables $\bar{\gamma}$. It can also have free type variables, which are interpreted as *rigid*...

Expressions

$$
\begin{array}{rl}
\text{Expressions} & e ::= x \mid \lambda(x:\theta).e \mid e\ e \mid \text{let}\, x = e \,\text{in}\, e \mid \mu(x:\varsigma).e \mid \\
& \quad K\, e \ldots e \mid \text{case}\, e\, \text{of}\, \bar{c} \mid \forall\bar{a}.e \mid (e:\theta) \\
\text{Clauses} & c ::= p.e \\
\text{Patterns} & p ::= K\,\bar{\beta}\,\bar{x}
\end{array}
$$

This is Core ML with polymorphic recursion, (generalized) algebraic data types, and type annotations.

Type variables are bound *rigidly* (universally) only. This (nonessential) restriction simplifies the presentation.

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*:

$$\frac{E, \Gamma \vdash e : \tau_1 \qquad E \Vdash \tau_1 = \tau_2}{E, \Gamma \vdash e : \tau_2}$$

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

Specification, continued

These are standard rules:

$$\frac{K \preceq \tau_1 \times \ldots \times \tau_n \rightarrow T\,\bar{\tau}_1\,\bar{\tau}_2 \qquad \forall i \quad E, \Gamma \vdash e_i : \tau_i}{E, \Gamma \vdash K\,e_1 \ldots e_n : T\,\bar{\tau}_1\,\bar{\tau}_2}$$

$$\frac{E, \Gamma \vdash e : \tau_1 \qquad \forall i \quad E, \Gamma \vdash c_i : \tau_1 \rightarrow \tau_2}{E, \Gamma \vdash \mathsf{case}\, e\, \mathsf{of}\, c_1 \ldots c_n : \tau_2}$$

The interesting stuff happens in the rules that deal with *individual clauses*...

Specification, continued

$$\frac{p : T\,\bar{\tau}_1\,\bar{\tau}_2 \vdash (\bar{\beta}, E', \Gamma') \qquad E \wedge E', \Gamma\Gamma' \vdash e : \tau_2 \qquad \bar{\beta} \mathrel{\#} \mathrm{ftv}(E, \Gamma, \tau_2)}{E, \Gamma \vdash p.e : T\,\bar{\tau}_1\,\bar{\tau}_2 \rightarrow \tau_2}$$

Inside each clause, *new (abstract) type variables*, *new type equations*, and *new environment entries* appear.

They are found by confronting the type $T\,\bar{\tau}_1\,\bar{\tau}_2$ of the scrutinee with the pattern p...

Specification, continued

Simple, but subtle.

$$\frac{K \preceq \forall\bar\beta.\tau_1 \times \ldots \times \tau_n \rightarrow T\,\bar\tau_1\,\bar\tau \qquad \bar\beta \;\#\; \ldots}{K\,\bar\beta\,x_1 \ldots x_n : T\,\bar\tau_1\,\bar\tau_2 \vdash (\bar\beta, \bar\tau_2 = \bar\tau, (x_1 : \tau_1; \ldots; x_n : \tau_n))}$$

Confronting the generalized type parameters ($\bar\tau$ versus $\bar\tau_2$) gives rise to new equations.

Instantiating the ordinary type parameters ($\bar\alpha$ versus $\bar\tau_1$) allows determining $\tau_1, \ldots, \tau_n$, as in ordinary ML.

Results and non-results

Theorem (Soundness for MLGI)

Well-typed MLGI programs do not go wrong.

As explained earlier, MLGI does *not* have principal types.

Introducing generalized algebraic data types

Typechecking: MLGI

Simple type inference: MLGX

Shape inference

MLGX

Let's require sufficiently many type annotations to ensure that E is *known* at all times and is *rigid*. Let's also make all type conversions *explicit*.

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

Specification

$$\frac{E, \Gamma \vdash (e : \theta) : \tau_1 \qquad \forall i \quad E, \Gamma \vdash (p_i : \theta).e_i : \tau_1 \longrightarrow \tau_2}{E, \Gamma \vdash \mathsf{case}\,(e : \theta)\;\mathsf{of}\; p_1.e_1 \ldots p_n.e_n : \tau_2}$$

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:

$$\frac{p : T\,\bar{\tau}_1\,\bar{\tau}'_2 \vdash (\bar{\beta}, E', \Gamma') \qquad E \wedge E', \Gamma\Gamma' \vdash e : \tau_2 \qquad \bar{\beta} \mathrel{\#} \ldots \qquad \bar{\gamma} \mathrel{\#} \ldots}{E, \Gamma \vdash (p : \exists\bar{\gamma}.T \;\star\; \bar{\tau}'_2).e : T\,\bar{\tau}_1 \star \rightarrow \tau_2}$$

The generalized type parameters *taken from the annotation* are used to determine E'. *No guessing* is involved. The weaker the annotation, the weaker E'.

($\star$ stands for a type that is discarded.)

(See analogous rule in MLGI.)

Specification, continued

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

$$\frac{E, \Gamma \vdash e : \tau_1 \qquad \kappa \preceq (\tau_1 \triangleright \tau_2) \qquad E \Vdash \kappa}{E, \Gamma \vdash (e : \kappa) : \tau_2}$$

The syntax of coercions is

$$\kappa ::= \exists \bar{\gamma}.(\tau \triangleright \tau)$$

E validates the coercion $\exists \bar{\gamma}.(\tau_1 \triangleright \tau_2)$ iff

$$E \Vdash \forall \bar{\gamma}.\tau_1 = \tau_2$$

holds.

Soundness and completeness

Theorem (Soundness for MLGX)

If $E, \Gamma \vdash e : \sigma$ holds in MLGX, then it holds in MLGI as well.

Theorem (Completeness with assistance for MLGX)

If $E, \Gamma \vdash e : \sigma$ holds in MLGI, then there exists an annotated version e' of e such that $E, \Gamma \vdash e' : \sigma$ holds in MLGX.

Type inference for MLGX

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

- compute E at all program points and *check* that every explicit coercion is valid;
- forget about E and follow the *standard* reduction to constraint solving. A coercion $\exists\bar{\gamma}.(\tau_1 \triangleright \tau_2)$ behaves just like a constant of type $\forall\bar{\gamma}.\tau_1 \rightarrow \tau_2$. *No implication constraints* are involved, so we recover *principal types*.

Details are omitted.

Programming in MLGX

In MLGX, *eval* is written:

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

This is nice, but *redundant*...

MLGX is modest

In short, MLGX marries *type inference* for Hindley and Milner's type system with *typechecking* for generalized algebraic data types.

In order to reduce the annotation burden, we again turn to *local shape inference*...

Introducing generalized algebraic data types

Typechecking: MLGI

Simple type inference: MLGX

Shape inference

Shapes

Shapes are defined by

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

The *flexible* type variables $\bar{\gamma}$ are bound within τ.

Shapes, continued

Flexible type variables are interpreted as standing for *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.

Shapes, continued

Shapes can have *free* type variables; these are interpreted as *rigid*.

For instance, the shape

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

describes a pair whose first component has type a, where the rigid type variable a was introduced by the programmer, and whose second component has unknown type.

Basic operations

$$\begin{array}{lcl}
\bot & = & \gamma.\gamma \\
(\bar\gamma_1.\tau_1) \longrightarrow (\bar\gamma_2.\tau_2) & = & \bar\gamma_1\bar\gamma_2.\tau_1 \longrightarrow \tau_2 \\
\mathcal{D}(\bot) & = & \bot \\
\mathcal{D}(\bar\gamma.\tau_1 \longrightarrow \star) & = & \bar\gamma.\tau_1 \\
\mathcal{C}(\bot) & = & \bot \\
\mathcal{C}(\bar\gamma.\star \longrightarrow \tau_2) & = & \bar\gamma.\tau_2
\end{array}$$

$\bot$ is the *uninformative* shape.

Out of two shapes, one forms an *arrow* shape. Conversely, out of an arrow shape, one projects *domain* and *codomain*.

Ordering shapes

Shapes are equipped with a standard *instantiation* ordering.

For instance,

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

Ordering shapes, continued

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

For instance,

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

(Recall that $\mathrm{int} \rightarrow \bot$ stands for $\gamma.\mathrm{int} \rightarrow \gamma$.)

This use of unification is *local*.

Normalizing shapes

If an expression is found to have both shape *a* and shape $\gamma.\beta_1 \rightarrow \gamma$, then shape inference should *fail*, because these do not have an upper bound...

... unless some equation in E proves that these shapes really are *compatible*. For instance,

$$a = \beta_1 \rightarrow \beta_2$$

is such an equation.

Normalizing shapes, continued

If we take E into account, then the shapes α and $\beta_1 \longrightarrow \beta_2$ become logically interchangeable.

But the latter is *more informative*...

... so we choose to always *normalize* the former into the latter.

Normalizing shapes, continued

Normalization is performed, roughly speaking, by viewing E as a rewrite system, and preferring structured types to type variables.

An *arbitrary* choice is made when an equation involves two type variables. Ouch! This is bad—but normalization is desirable.

We write $s\lfloor_E$ for the normalized version of s with respect to E.

Pruning shapes

Recall this problematic code snippet:

$$\begin{array}{l} \mathit{cast} = \\ \quad \forall \alpha\beta.\lambda(w : \mathit{eq}\ \alpha\ \beta).\lambda(x : \alpha). \\ \qquad \mathit{case}\ w\ \mathit{of} \\ \qquad\quad \mathit{Eq} \longrightarrow (*\ \alpha = \beta\ *)\ x \end{array}$$

Shape inference will probably infer that x has shape α, which is correct, *up to the equation $\alpha = \beta$*.

Can it rightfully infer that the case construct also has type α? *No*, because the equation $\alpha = \beta$ is no longer available. The correct shape could be β, and the two are incompatible.

Pruning shapes, continued

To ensure that we only infer *correct* shapes, we *prune* unreliable information when exiting a case construct.

Thus, we infer $\bot$ instead of making an arbitrary decision.

This allows us to later prove a *soundness* theorem.

One could also choose to ignore this issue...

Algorithm W

Here is a shape inference algorithm inspired by Peyton Jones *et al.*'s. "wobbly types" paper.

It is *bidirectional*:

$$
\begin{array}{ll}
E, \Gamma \vdash e \uparrow s \rightsquigarrow e' & \text{synthesis, } s \text{ is } \textit{inferred} \\
E, \Gamma \vdash e \downarrow s \rightsquigarrow e' & \text{checking, } s \text{ is } \textit{provided}
\end{array}
$$

Γ maps variables to shapes.

An invariant is that s is normalized with respect to E.

Algorithm W, continued

The transformed term e' is identical to e, except

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

Algorithm W, continued

Variable, synthesis mode:

$$\frac{(x : s) \in \Gamma}{E, \Gamma \vdash x \uparrow s \downharpoonleft_E \rightsquigarrow (x \downarrow_E s)}$$

The inferred shape is the *normalized* version of the shape in Γ.

This normalization step is reflected in the transformed expression by inserting an explicit *coercion*...

Algorithm W, continued

By definition,

$$(e \downarrow_E \bar{\gamma}.\tau)$$

is sugar for

$$(e : \exists\bar{\gamma}.(\tau \triangleright \tau\lfloor_E\,))$$

Inserting a coercion amounts to *explicitly telling MLGX* about the equations that we are exploiting.

Algorithm W, continued

An instance of the rule is

$$\frac{(x : a) \in \Gamma}{a = \mathsf{int}, \Gamma \vdash x \uparrow \mathsf{int} \leadsto (x : (a \rhd \mathsf{int}))}$$

That is, if x is known to have shape a and if the equation $a = \mathsf{int}$ is locally available, then we infer that x has shape int and insert the corresponding coercion.

Algorithm W, continued

Application, synthesis mode:

$$\frac{E, \Gamma \vdash e_1 \uparrow \mathfrak{s} \leadsto e_1' \qquad E, \Gamma \vdash e_2 \downarrow \mathcal{D}(\mathfrak{s}) \leadsto e_2'}{E, \Gamma \vdash e_1\ e_2 \uparrow \mathcal{C}(\mathfrak{s}) \leadsto e_1'\ e_2'}$$

Simple stuff. *Arbitrary* choices of modes in the premises.

Algorithm W, continued

Explicit type annotation, synthesis mode:

$$\frac{E, \Gamma \vdash e \downarrow \theta\lfloor_E \rightsquigarrow e'}{E, \Gamma \vdash (e : \theta) \uparrow \theta\lfloor_E \rightsquigarrow (e' : \theta\lfloor_E)}$$

The mode *changes* from synthesis to checking.

The type annotation is *normalized* in the transformed expression.

Algorithm W, continued

Clause, checking mode:

$$\frac{p : T\,\bar{\tau}_1\,\bar{\tau}_2 \vdash (\bar{\beta}, E', \Gamma') \qquad E \wedge E', \Gamma(\bar{\gamma}.\Gamma') \vdash e \downarrow s \downharpoonleft_{E \wedge E'} \leadsto e' \qquad \bar{\beta} \mathrel{\#} \ldots \qquad \bar{\gamma} \mathrel{\#} \ldots}{E, \Gamma \vdash (p : \bar{\gamma}.T\,\bar{\tau}_1\,\bar{\tau}_2).e \downarrow s \leadsto p.(e' \upharpoonleft_{E \wedge E'} s)}$$

The environment is extended with new bindings of variables to *shapes*.

The expected shape s is *normalized* with respect to the *new theory* $E \wedge E'$.

This normalization step is reflected by *inserting a reverse coercion*...

Programming in MLGX

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

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

Soundness

Theorem (Soundness for Algorithm W)

Let $E, \Gamma \vdash e : \sigma$ hold in MLGI. Let $E \Vdash \Gamma' \preceq \Gamma$ hold. Then,

1. *If $E, \Gamma' \vdash e \uparrow s \rightsquigarrow e'$ holds in W, then $E \Vdash s \preceq \sigma$ holds and $E, \Gamma \vdash e' : \sigma$ holds in MLGI.*
2. *… (analogous statement for checking mode)*

The inferred shape s is a *sound approximation* of the true shape σ, and the changes made to the program *do not in principle break it*.

Still, there is *no guarantee* that the transformed program is well-typed in MLGX!

Are we happy?

MLGX seems simple but *robust*.

There is no doubt that Algorithm W can be improved (and that is done in the paper), but any shape inference algorithm is bound to be *ad hoc*.

Selected References

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.

Part IV

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 definition 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, it is wise to modify the *constraint generation* process so as to take advantage of user-provided *hints*—typically, mandatory type annotations.

Local type inference

If the necessary hints are so numerous that they become a burden, a *local type 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.