Strong static typing and advanced functional programming

Francesco Zappa Nardelli

Moscova Project — INRIA Rocquencourt

BISS 2005, Bertinoro, Italy — March 14-18, 2005

Around 1949...

"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realised that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

Sir Maurice Wilkes (1913 -)

...in 2005?

How to ensure that a system behaves correctly with respect to some specification (implicit or explicit)?

Answer: *formal methods*.

- powerful: Hoare logic, modal logics, denotational semantics...
- lightweight: model checkers, run-time monitoring, *type systems*...

What type systems are good for

- Detecting errors;
- enforcing abstraction;
- documentation;
- efficiency;
- guarantee (to some extent) *language safety*.

Language safety

A safe language is one that protects its own abstractions.

	Statically checked	Dynamically checked
Safe	ML, Haskell, Java, ...	Lisp, Scheme, Postscript, Perl,...
Unsafe	C, C++, ...	

This course

"Formal methods will never have a significant impact
until they can be used by people that don't understand them."

attributed to Tom Melham

Type systems can be used by people that don't understand them!

But we are computer scientists: the objective of this course is to

understand (a subset of) the Objective Caml type system.

Outline

1. mini-ML

   syntax, big-step reduction semantics, monomorphic types, the Curry-Howard correspondence, polymorphic types, run-time errors, small-step reduction semantics, safety

2. Type inference

   the W algorithm, constraint-based type inference, HM(X)

3. Simple extensions of mini-ML

   tuples, sums, recursive types, algebraic types

4. Imperative programming

   references, exceptions

5. Extensible records

   row polymorphism, constraint-based unification, a simple object system, subtyping

6. The OCaml module system

The syntax of mini-ML

Expressions:

$$
\begin{array}{rcll}
a & ::= & x & \text{identifier} \\
 & | & c & \text{constant} \\
 & | & op & \text{primitive unary operator} \\
 & | & \mathtt{fun}\ x \rightarrow a & \text{function abstraction} \\
 & | & a\ a & \text{function application} \\
 & | & (a, a) & \text{pair construction} \\
 & | & \mathtt{fst}\ a & \text{left projection} \\
 & | & \mathtt{snd}\ a & \text{right projection} \\
 & | & \mathtt{let}\ x = a\ \mathtt{in}\ a & \text{local binding}
\end{array}
$$

Constants: $0, 1, 2, \ldots \quad \mathtt{true}, \mathtt{false}, \quad \mathtt{"foo"}, \ldots$

Operators: $+, -, \ldots \quad \mathtt{\&\&}, \mathtt{not}, \ldots \quad \mathtt{\hat{}}, \ldots \quad \mathtt{fix}, \ldots$

α-conversion and substitution

Free variables:

$$
\begin{aligned}
\mathcal{L}(x) &= \{x\} \\
\mathcal{L}(c) = \mathcal{L}(op) &= \emptyset \\
\mathcal{L}(\texttt{fun } x \to a) &= \mathcal{L}(a) \setminus \{x\} \\
\mathcal{L}(a_1\ a_2) = \mathcal{L}((a_1, a_2)) &= \mathcal{L}(a_1) \cup \mathcal{L}(a_2) \\
\mathcal{L}(\texttt{fst } a) = \mathcal{L}(\texttt{snd } a) &= \mathcal{L}(a) \\
\mathcal{L}(\texttt{let } x = a_1 \texttt{ in } a_2) &= \mathcal{L}(a_1) \cup (\mathcal{L}(a_2) \setminus \{x\})
\end{aligned}
$$

We identify α-equivalent terms (that is, we consider terms up-to renaming of bound variables).

α-conversion and substitution [2]

Capture avoding substitution:

$$
\begin{array}{rll}
x[x \leftarrow a'] & = a' & \\
y[x \leftarrow a'] & = y & \text{if } y \neq x \\
c[x \leftarrow a'] & = c & \\
op[x \leftarrow a'] & = op & \\
(\texttt{fun}\ y \to a)[x \leftarrow a'] & = \texttt{fun}\ y \to (a[x \leftarrow a']) & \\
 & \quad \text{if } y \neq x \text{ and } y \notin \mathcal{L}(a') & \\
(a_1\ a_2)[x \leftarrow a'] & = a_1[x \leftarrow a']\ a_2[x \leftarrow a'] & \\
(\texttt{let}\ y = a_1\ \texttt{in}\ a_2)[x \leftarrow a'] & = \texttt{let}\ y = a_1[x \leftarrow a']\ \texttt{in}\ a_2[x \leftarrow a'] & \\
 & \quad \text{if } y \neq x \text{ and } y \notin \mathcal{L}(a') &
\end{array}
$$

Reduction rules (big step semantics)

Values: $v ::= \texttt{fun}\ x \rightarrow a$ functions

$\mid c$ constants

$\mid op$ un-applied operators

$\mid (v_1, v_2)$ pairs of values

The big-step semantics relates expressions and values: $(a, v) \in \twoheadrightarrow$ is written $a \twoheadrightarrow v$.

$$c \twoheadrightarrow c \qquad op \twoheadrightarrow op \qquad (\texttt{fun}\ x \rightarrow a) \twoheadrightarrow (\texttt{fun}\ x \rightarrow a)$$

$$\frac{a_1 \twoheadrightarrow (\texttt{fun}\ x \rightarrow a) \qquad a_2 \twoheadrightarrow v_2 \qquad a[x \leftarrow v_2] \twoheadrightarrow v}{a_1\ a_2 \twoheadrightarrow v} \qquad \frac{a_1 \twoheadrightarrow v_1 \qquad a_2 \twoheadrightarrow v_2}{(a_1, a_2) \twoheadrightarrow (v_1, v_2)}$$

$$\frac{a \twoheadrightarrow (v_1, v_2)}{\texttt{fst}\ a \twoheadrightarrow v_1 \qquad \texttt{snd}\ a \twoheadrightarrow v_2} \qquad \frac{a_1 \twoheadrightarrow v_1 \qquad a_2[x \leftarrow v_1] \twoheadrightarrow v}{(\texttt{let}\ x = a_1\ \texttt{in}\ a_2) \twoheadrightarrow v}$$

Reduction rules [2]

Examples of reduction rules of operators:

$$\frac{a_1 \twoheadrightarrow + \qquad a_2 \twoheadrightarrow (n_1, n_2)}{a_1\ a_2 \twoheadrightarrow n_1 + n_2}$$

$$\frac{a \twoheadrightarrow (\texttt{fun}\ f \to a_1) \qquad a_1[f \leftarrow \texttt{fix}\ (\texttt{fun}\ f \to a_1)] \twoheadrightarrow v}{\texttt{fix}\ a \twoheadrightarrow v}$$

Big-step vs. small-step semantics

The big-step semantics associates a *value* v, or $\texttt{err}$[1], to all expressions a:

$$a \twoheadrightarrow v$$

but does not give any information on the intermediary steps of the computation.

The *small-step semantics* relates two expressions:

$$a \rightarrow a_1 \ .$$

A computation is a sequence of steps:

$$a \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \not\rightarrow \ .$$

[1] the rules for `err` are missing in the slides.

Reduction rules (small-step semantics)

Axioms:

$$\begin{array}{rcll} (\texttt{fun}\ x \to a)\ v & \xrightarrow{\varepsilon} & a[x \leftarrow v] & (\beta) \\ (\texttt{let}\ x = v\ \texttt{in}\ a) & \xrightarrow{\varepsilon} & a[x \leftarrow v] & (\textsf{\textit{let}}) \\ \texttt{fst}\ (v_1, v_2) & \xrightarrow{\varepsilon} & v_1 & (\textsf{\textit{fst}}) \\ \texttt{snd}\ (v_1, v_2) & \xrightarrow{\varepsilon} & v_2 & (\textsf{\textit{snd}}) \end{array}$$

Delta rules:

$$\begin{array}{rcll} +\ (n_1, n_2) & \xrightarrow{\varepsilon} & n_1 + n_2 & (\delta_+) \\ \texttt{fix}\ (\texttt{fun}\ x \to a) & \xrightarrow{\varepsilon} & a[x \leftarrow \texttt{fix}\ (\texttt{fun}\ x \to a)] & (\delta_{\textsf{fix}}) \end{array}$$

Reduction under an evaluation context:

$$\frac{a \xrightarrow{\varepsilon} a'}{E[a] \to E[a']}\ (\text{context})$$

Evaluation contexts

Evaluation contexts:

$E ::= [\,]$	head
$\mid E\ a$	left of an application
$\mid v\ E$	right of an application
$\mid \texttt{let}\ x = E\ \texttt{in}\ a$	left of a `let`
$\mid (E, a)$	left of a pair
$\mid (v, E)$	right of a pair
$\mid \texttt{fst}\ E$	argument of a projection
$\mid \texttt{snd}\ E$	

Types

Types: $\tau ::= T$ basic types (`int`, `bool`, etc)

$\mid \alpha$ type variable

$\mid \tau_1 \to \tau_2$ type of functions from τ_1 into τ_2

$\mid \tau_1 \times \tau_2$ type of pairs of τ_1 and τ_2

Type relation: (Γ, a, τ), traditionally written

$$\Gamma \vdash a : \tau$$

Type environment

Γ is a type environment: a partial function between variables and types, that associates a type $\Gamma(x)$ to all free identifiers of a.

- $\emptyset$ denotes the empty evironment,
- $\Gamma, x : \tau$ denotes the environment that associates τ to x, and $\Gamma(y)$ to all other identifiers y.

Monomorphic types

$$\Gamma \vdash x : \Gamma(x) \ \ (\text{var}) \qquad \Gamma \vdash c : TC(c) \ \ (\text{const}) \qquad \Gamma \vdash op : TC(op) \ \ (\text{op})$$

$$\frac{\Gamma; x : \tau_1 \vdash a : \tau_2}{\Gamma \vdash (\texttt{fun}\ x \to a) : \tau_1 \to \tau_2} \ (\text{fun}) \qquad \frac{\Gamma \vdash a_1 : \tau' \to \tau \qquad \Gamma \vdash a_2 : \tau'}{\Gamma \vdash a_1\ a_2 : \tau} \ (\text{app})$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash (a_1, a_2) : \tau_1 \times \tau_2} \ (\text{pair}) \qquad \frac{\Gamma \vdash a : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ a : \tau_1 \qquad \Gamma \vdash \texttt{snd}\ a : \tau_2} \ (\text{proj})$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma; x : \tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2} \ (\text{let})$$

A type derivation

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{x : \texttt{int} \vdash x : \texttt{int} \qquad x : \texttt{int} \vdash 1 : \texttt{int}}{x : \texttt{int} \vdash (x, 1) : \texttt{int} \times \texttt{int}}
\quad
x : \texttt{int} \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int}
}{x : \texttt{int} \vdash +(x, 1) : \texttt{int}}
}{\emptyset \vdash \texttt{fun}\ x \to +(x, 1) : \texttt{int} \to \texttt{int}}
\qquad
\dfrac{
f : \texttt{int} \to \texttt{int} \vdash f : \texttt{int} \to \texttt{int} \quad f : \texttt{int} \to \texttt{int} \vdash 2 : \texttt{int}
}{f : \texttt{int} \to \texttt{int} \vdash f\ 2 : \texttt{int}}
}{\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \to +(x, 1)\ \texttt{in}\ f\ 2 : \texttt{int}}
$$

Where are we?

- mini-ML (lambda-calculus + pairs + `let`)
- monomorphic type system

Until here, the `let` construct is syntactic sugar:

$$\texttt{let}\ x = a_1\ \texttt{in}\ a_2$$

is equivalent to

$$(\texttt{fun}\ x \rightarrow a_2)\ a_1\ .$$

Some type judgments

Valid judgments:

$$\emptyset \vdash \texttt{fun}\ x \to x : \alpha \to \alpha$$
$$\emptyset \vdash \texttt{fun}\ x \to x : \texttt{bool} \to \texttt{bool}$$

Not valid judgements:

$$\emptyset \vdash \texttt{fun}\ x \to +(x, 1) : \texttt{int}$$
$$\emptyset \vdash \texttt{fun}\ x \to +(x, 1) : \alpha \to \texttt{int}$$

Expressions not typable (there is no Γ and no τ such that $\Gamma \vdash a : \tau$).

$$1\ 2$$
$$\texttt{fun}\ f \to f\ f$$
$$\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$$

Type scheme

Type schemes are a compact and finite representation for all the types that can be given to a polymorphic expression.

Type scheme: $\sigma ::= \forall \alpha_1, \ldots, \alpha_n.\, \tau$

Example: in $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$, we can associate the type schema $\forall \alpha . \alpha \to \alpha$ to the identifier f.

If the set of quantified variables is empty, we write τ instead of $\forall .\, \tau$.

Free variables, again

Free variables of a type scheme:

$$
\begin{array}{rcl}
\mathcal{L}(T) & = & \emptyset \\
\mathcal{L}(\alpha) & = & \{\alpha\} \\
\mathcal{L}(\tau_1 \to \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\tau_1 \times \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\forall \alpha_1, \ldots, \alpha_n . \tau) & = & \mathcal{L}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}
\end{array}
$$

We identify type schemas up-to α-conversion.

Free variables of a type environment:

$$
\mathcal{L}(\Gamma) = \bigcup_{x \in \mathrm{Dom}(\Gamma)} \mathcal{L}(\Gamma(x))
$$

Instantiation of a type schema

Intuition: $\forall\alpha.\,\alpha \to \alpha$ can be seen as the set of types $\{\tau \to \tau \mid \tau \text{ is a type}\}$.

Formally:

$$\forall\alpha_1 \ldots \alpha_n.\,\tau' \leq \tau \text{ iff there exist } \tau_1, \ldots, \tau_n \text{ s.t. } \tau = \tau'[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n]$$

Examples:

$\texttt{int} \to \texttt{int}$ and $\texttt{bool} \to \texttt{bool}$ are instances of $\forall\alpha.\,\alpha \to \alpha$.

The type $\texttt{int} \to \texttt{bool}$ is not.

$\forall.\tau' \leq \tau$ is equivalent to $\tau = \tau'$.

Polymorphic types à la ML

$$\frac{\Gamma(x) \leq \tau}{\Gamma \vdash x : \tau} \text{ (var-inst)} \qquad \frac{TC(c) \leq \tau}{\Gamma \vdash c : \tau} \text{ (const-inst)} \qquad \frac{TC(op) \leq \tau}{\Gamma \vdash op : \tau} \text{ (op-inst)}$$

$$\frac{\Gamma; x : \tau_1 \vdash a : \tau_2}{\Gamma \vdash (\texttt{fun } x \rightarrow a) : \tau_1 \rightarrow \tau_2} \text{ (fun)} \qquad \frac{\Gamma \vdash a_1 : \tau' \rightarrow \tau \qquad \Gamma \vdash a_2 : \tau'}{\Gamma \vdash a_1\ a_2 : \tau} \text{ (app)}$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash (a_1, a_2) : \tau_1 \times \tau_2} \text{ (pair)} \qquad \frac{\Gamma \vdash a : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst } a : \tau_1 \qquad \Gamma \vdash \texttt{snd } a : \tau_2} \text{ (proj)}$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma; x : Gen(\tau_1, \Gamma) \vdash a_2 : \tau_2}{\Gamma \vdash (\texttt{let } x = a_1 \texttt{ in } a_2) : \tau_2} \text{ (let-gen)}$$

$$Gen(\tau_1, \Gamma) = \forall \alpha_1, \ldots, \alpha_n .\, \tau_1 \quad \text{where} \quad \{\alpha_1, \ldots, \alpha_n\} = \mathcal{L}(\tau_1) \setminus \mathcal{L}(\Gamma)$$

Example

$$\dfrac{\dfrac{\dfrac{\alpha \le \alpha}{x : \alpha \vdash x : \alpha}}{\emptyset \vdash \texttt{fun}\ x \to x : \alpha \to \alpha} \qquad \dfrac{\dfrac{\forall\alpha.\,\alpha\to\alpha \le \texttt{int}\to\texttt{int}}{f : \forall\alpha.\,\alpha\to\alpha \vdash f : \texttt{int}\to\texttt{int}} \quad f : \forall\alpha.\,\alpha\to\alpha \vdash 1 : \texttt{int}}{f : \forall\alpha.\,\alpha\to\alpha \vdash f\ 1 : \texttt{int}}}{\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ f\ 1 : \texttt{int}}$$

Another example

$$\dfrac{\dfrac{\dfrac{\alpha \leq \alpha}{x : \alpha; z : \beta \vdash x : \alpha}}{x : \alpha \vdash \texttt{fun}\ z \to x : \beta \to \alpha} \qquad \dfrac{\forall \beta.\beta \to \alpha \leq \gamma \to \alpha}{x : \alpha; y : \forall \beta.\beta \to \alpha \vdash y : \gamma \to \alpha}}{\dfrac{x : \alpha \vdash \texttt{let}\ y = \texttt{fun}\ z \to x\ \texttt{in}\ y : \gamma \to \alpha}{\emptyset \vdash \texttt{fun}\ x \to \texttt{let}\ y = \texttt{fun}\ z \to x\ \texttt{in}\ y : \alpha \to \gamma \to \alpha}}$$

The side condition on the (let-gen) rule

Consider the valid type judgement

$$z : \alpha \vdash z : \alpha$$

An unrestricted (let-gen) rule allows us to derive

$$z : \alpha \vdash \mathtt{let}\ x = z\ \mathtt{in}\ x : \forall \alpha.\, \alpha$$

Then, by (var-inst), we can also deduce

$$z : \alpha \vdash \mathtt{let}\ x = z\ \mathtt{in}\ x : \beta$$

By (fun), we derive

$$\emptyset \vdash \mathtt{fun}\ z \rightarrow \mathtt{let}\ x = z\ \mathtt{in}\ x : \alpha \rightarrow \beta$$

which does not make any sense (eg, the unrestricted (let-gen) rule is *unsound*).

Some facts

Proposition 1. [Judgements are stable under substitution on types] *Let φ be a substitution. If $\Gamma \vdash a : \tau$, then $\varphi(\Gamma) \vdash a : \varphi(\tau)$.*

Proposition 2. [Elimination of unused hypothesis] *If for all identifier free in a, the hypothesis in $\Gamma_1(x)$ and $\Gamma_2(x)$ coincide, then $\Gamma_1 \vdash a : \tau$ is equivalent to $\Gamma_2 \vdash a : \tau$.*

Proposition 3. [Judgements are stable under strenghtening of hypothesis] *If Γ and Γ' have the same domain and if for all $x \in \mathrm{Dom}(\Gamma)$ it holds $\Gamma'(x) \leq \Gamma(x)$, then $\Gamma \vdash a : \tau$ implies $\Gamma' \vdash a : \tau$.*

Safety

Given a term a, one of the following holds:

1. a reduces in a finite number of steps to a value v:

$$a \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \rightarrow v$$

2. a reduces forever:

$$a \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \rightarrow \ldots$$

3. a reduces to a *stuck expression* (run-time error):

$$a \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \not\rightarrow$$

Claim: if a is well-typed, case 3. cannot occur.

The relation “less typable of”

We say that a_1 is *less typable of* a_2, denoted $a_1 \sqsubseteq a_2$, if

$$\text{for all } \Gamma \text{ and } \tau,\ (\Gamma \vdash a_1 : \tau) \text{ implies } (\Gamma \vdash a_2 : \tau)$$

Proposition 4. [Congruence of $\sqsubseteq$] *For all context C, $a_1 \sqsubseteq a_2$ implies $C[a_1] \sqsubseteq C[a_2]$.*

Proof: Let Γ and τ such that $\Gamma \vdash C[a_1] : \tau$. Show that $\Gamma \vdash C[a_2] : \tau$ by structural induction on the context C. $\square$

Judgements and substitutions

Proposition 5. [Substitution Lemma] *Suppose that*

$$\begin{aligned} \Gamma &\vdash a' : \tau' \\ \Gamma; x : \forall \alpha_1 \ldots \alpha_n . \tau' &\vdash a : \tau \end{aligned}$$

where $\alpha_1, \ldots, \alpha_n$ are not free in Γ. Then,

$$\Gamma \vdash a[x \leftarrow a'] : \tau$$

Proof: By induction on the structure of the expression a. □

Hypothesis on the operators

Let `cast` be an operator of type $\forall \alpha, \beta.\ \alpha \to \beta$, which reduces as `cast` $v \xrightarrow{\varepsilon} v$. The type system is then unsound!

We made some hypothesis on the operators:

H0 Fo all operators op, $TC(op)$ is of the form $\forall \vec{\alpha}.\tau \to \tau'$. For all constant c, $TC(c)$ is a base type T.

H1 If $a \xrightarrow{\varepsilon} a'$ by a δ-rule, then $a \sqsubseteq a'$.

H2 If $\emptyset \vdash op\ v : \tau$, then there is an expression a' such that $op\ v \xrightarrow{\varepsilon} a'$ by a δ-rule.

Safety, at last

Proposition 6. *If* $a \xrightarrow{\varepsilon} a'$*, then* $a \sqsubseteq a'$*.*

Proof: Case analysis on the reduction rule used. □

Proposition 7. [Subject reduction] *If* $a \rightarrow a'$*, then* $a \sqsubseteq a'$*.*

Proof: Follows from congruence of $\sqsubseteq$. □

Proposition 8. [Progression Lemma] *If* $\emptyset \vdash a : \tau$*, either* a *is a value or there exists an expression* a' *such that* $a \rightarrow a'$*.*

Proof: Induction on the structure of a. □

Theorem 1. [Safety] *If* $\emptyset \vdash a : \tau$ *and* $a \rightarrow^\star a' \not\rightarrow$*, then* a' *is a value.*

Type inference

(Pure) verification : all subexpressions have been explicitely annotated:

```
fun (x : int) → (
  let y : int = (+ : int×int→int) ((x : int, 1 : int) : int×int) in
  y : int
) : int
```

Declaration of the types of identifiers and propagation of types : the programmer specifies the types of the parameters of functions and of local variables:

```
fun (x : int) → let y : int = + (x, 1) in y
```

Inference of all types :

```
fun x → let y = + (x, 1) in y
```

Monomorphic type inference

Two phases:

1. Starting from the source program, we build a system of equations between types that characterizes all its possible typings;

2. We solve this system of equations: if there are no solutions the program is badly typed, otherwise we obtain a *principal* solution. This gives us a *principal type* of the program.

Equation generation

- If a is a variable x, then $C(a) = \{\alpha_a \stackrel{?}{=} \alpha_x\}$.
- If a is a constant c, then $C(a) = \{\alpha_a \stackrel{?}{=} TC(c)\}$.
- If a is an operator op, then $C(a) = \{\alpha_a \stackrel{?}{=} TC(op)\}$.
- If a is a function `fun` $x \to b$, then $C(a) = \{\alpha_a \stackrel{?}{=} \alpha_x \to \alpha_b\} \cup C(b)$.
- If a is an application $b\ c$, then $C(a) = \{\alpha_b \stackrel{?}{=} \alpha_c \to \alpha_a\} \cup C(b) \cup C(c)$.
- if a is a pair (b, c), then $C(a) = \{\alpha_a \stackrel{?}{=} \alpha_b \times \alpha_c\} \cup C(b) \cup C(c)$.
- If a is a projection `fst` b, then $C(a) = \{\alpha_a \times \beta_a \stackrel{?}{=} \alpha_b\} \cup C(b)$.
- If a is a projection `snd` b, then $C(a) = \{\beta_a \times \alpha_a \stackrel{?}{=} \alpha_b\} \cup C(b)$.
- If a is `let` $x = b$ `in` c, then $C(a) = \{\alpha_x \stackrel{?}{=} \alpha_b;\ \alpha_a \stackrel{?}{=} \alpha_c\} \cup C(b) \cup C(c)$.

Example

Consider the program:

$$a = (\underbrace{\texttt{fun}\ x \rightarrow \underbrace{\texttt{fun}\ y \rightarrow \underbrace{1}_{d}}_{c}}_{b})\ \underbrace{\texttt{true}}_{e}$$

We have:

$$\begin{aligned} C(a) = \{\ & \alpha_b \stackrel{?}{=} \alpha_e \rightarrow \alpha_a; \\ & \alpha_b \stackrel{?}{=} \alpha_x \rightarrow \alpha_c; \\ & \alpha_c \stackrel{?}{=} \alpha_y \rightarrow \alpha_d; \\ & \alpha_d \stackrel{?}{=} \texttt{int}; \\ & \alpha_e \stackrel{?}{=} \texttt{bool}\} \end{aligned}$$

Typings and equations

A *solution* of a set of equations $C(a)$ is a substitution φ such that for all equations $\tau_1 \stackrel{?}{=} \tau_2$ in $C(a)$, it holds $\varphi(\tau_1) = \varphi(\tau_2)$.

Proposition 9. [Correction of equations] *If φ is a solution of $C(a)$, then $\Gamma \vdash a : \varphi(\alpha_a)$ where Γ is the type environment $\{x : \varphi(\alpha_x) \mid x \in \mathcal{L}(a)\}$.*

Proposition 10. [Completeness of equations] *Let a be an expression. If there exists an environment Γ and a type τ such that $\Gamma \vdash a : \tau$, then the system of equations $C(a)$ has a solution φ such that $\varphi(\alpha_a) = \tau$ and $\varphi(\alpha_x) = \Gamma(x)$ for all $x \in \mathcal{L}(a)$.*

Equation solution

$$
\begin{array}{rcl}
\texttt{mgu}(\emptyset) & = & id \\
\texttt{mgu}(\{\alpha \stackrel{?}{=} \alpha\} \cup C) & = & \texttt{mgu}(C) \\
\texttt{mgu}(\{\alpha \stackrel{?}{=} \tau\} \cup C) & = & \texttt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ if } \alpha \text{ not free in } \tau \\
\texttt{mgu}(\{\tau \stackrel{?}{=} \alpha\} \cup C) & = & \texttt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ if } \alpha \text{ not free in } \tau \\
\texttt{mgu}(\{\tau_1 \rightarrow \tau_2 \stackrel{?}{=} \tau_1' \rightarrow \tau_2'\} \cup C) & = & \texttt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_1'; \tau_2 \stackrel{?}{=} \tau_2'\} \cup C) \\
\texttt{mgu}(\{\tau_1 \times \tau_2 \stackrel{?}{=} \tau_1' \times \tau_2'\} \cup C) & = & \texttt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_1'; \tau_2 \stackrel{?}{=} \tau_2'\} \cup C)
\end{array}
$$

In all other cases, $\texttt{mgu}(C)$ fails.

Back to the last example

The principal solution is

$$
\begin{array}{ll}
\alpha_x \leftarrow \texttt{bool} & \alpha_e \leftarrow \texttt{bool} \\
\alpha_a \leftarrow \alpha_y \rightarrow \texttt{int} & \alpha_c \leftarrow \alpha_y \rightarrow \texttt{int} \\
\alpha_d \leftarrow \texttt{int} &
\end{array}
$$

All other solutions can be obtained by replacing α_y by an arbitrary type.

Properties of $\mathtt{mgu}$

1. Correct: if $\mathtt{mgu}(C) = \varphi$, then φ is a solution of C.

2. Completeness: if C has a solution ψ, then $\mathtt{mgu}(C)$ does not fail, and returns a solution φ such that $\varphi \leq \psi$.

The inequality $\varphi \leq \psi$ holds iff there exists a substitution θ such that $\psi = \varphi \circ \theta$.

The algorithm I

Input: an expression a.

Output: a typing Γ, τ, or fails.

Algorithm: compute $\varphi = \mathtt{mgu}(C(a))$ (failure possible). If success, return the environment $\{x : \varphi(\alpha_x) \mid x \in \mathcal{L}(a)\}$ and the type $\varphi(\alpha_a)$.

Proposition 11. [Properties of the algorithm I]

1. *Correction: if $I(a) = (\Gamma, \tau)$, then $\Gamma \vdash a : \tau$.*

2. *Principality: if $\Gamma' \vdash a : \tau'$, then $I(a)$ succeeds and returns a typing (Γ, τ) more general than (Γ', τ'), that is, there exists a substitution θ such that $\tau' = \theta(\tau)$ and $\Gamma'(x) = \theta(\Gamma(x))$ for all $x \in \mathcal{L}(a)$.*

Polymorphic type inference

If we apply the algorithm I to

$$\texttt{let}\ f = \texttt{fun}\ x \rightarrow x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$$

the algorithm will fail, because it will end up with an equation

$$\texttt{int} \stackrel{?}{=} \texttt{bool}$$

that does not have a solution.

The algorithm W

Damas, Milner, 1982

Input: a type environment Γ and one expression a.

Output: the inferred type τ, or fails.

We give an algorithmic presentation that relies on a...

State: a current substitution φ and an infinite set of variables V.

Definition:

$$
\begin{array}{lcl}
\texttt{fresh} & = & \texttt{do}\, \alpha \in V \\
 & & \texttt{do}\, V \leftarrow V \setminus \{\alpha\} \\
 & & \texttt{return}\, \alpha \\
W(\Gamma \vdash x) & = & \texttt{let}\, \forall \alpha_1 \ldots \alpha_n . \tau = \Gamma(x) \\
 & & \texttt{do}\, \beta_1, \ldots, \beta_n = \texttt{fresh}, \ldots, \texttt{fresh} \\
 & & \texttt{return}\, \tau[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n] \\
W(\Gamma \vdash \texttt{fun}\ x \rightarrow a_1) & = & \texttt{do}\, \alpha = \texttt{fresh} \\
 & & \texttt{do}\, \tau_1 = W(\Gamma; x : \alpha \vdash a_1) \\
 & & \texttt{return}\, \alpha \rightarrow \tau_1 \\
W(\Gamma \vdash a_1\, a_2) & = & \texttt{do}\, \tau_1 = W(\Gamma \vdash a_1) \\
 & & \texttt{do}\, \tau_2 = W(\Gamma \vdash a_2) \\
 & & \texttt{do}\, \alpha = \texttt{fresh} \\
 & & \texttt{do}\, \varphi \leftarrow \texttt{mgu}(\varphi(\tau_1) \stackrel{?}{=} \varphi(\tau_2 \rightarrow \alpha)) \circ \varphi \\
 & & \texttt{return}\, \alpha
\end{array}
$$

$$
\begin{array}{lcl}
W(\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2) & = & \texttt{do}\, \tau_1 = W(\Gamma \vdash a_1) \\
& & \texttt{let}\, \sigma = Gen(\varphi(\tau_1), \varphi(\Gamma)) \\
& & \texttt{return}\, W(\Gamma; x : \sigma \vdash a_2) \\
W(\Gamma \vdash (a_1, a_2)) & = & \texttt{do}\, \tau_1 = W(\Gamma \vdash a_1) \\
& & \texttt{do}\, \tau_2 = W(\Gamma \vdash a_2) \\
& & \texttt{return}\, \tau_1 \times \tau_2 \\
W(\Gamma \vdash \texttt{fst}\ a) & = & \texttt{do}\, \tau = W(\Gamma \vdash a) \\
& & \texttt{do}\, \alpha_1, \alpha_2 = \texttt{fresh}, \texttt{fresh} \\
& & \texttt{do}\, \varphi \leftarrow \texttt{mgu}(\varphi(\tau) \stackrel{?}{=} \alpha_1 \times \alpha_2) \circ \varphi \\
& & \texttt{return}\, \alpha_1 \\
W(\Gamma \vdash \texttt{snd}\ a) & = & \texttt{do}\, \tau = W(\Gamma \vdash a) \\
& & \texttt{do}\, \alpha_1, \alpha_2 = \texttt{fresh}, \texttt{fresh} \\
& & \texttt{do}\, \varphi \leftarrow \texttt{mgu}(\varphi(\tau) \stackrel{?}{=} \alpha_1 \times \alpha_2) \circ \varphi \\
& & \texttt{return}\, \alpha_2
\end{array}
$$

Example

Let $a = \texttt{fun } x \rightarrow +(x, 1)$.

$$\begin{array}{rcl}
W(x : \alpha \vdash +, id, \mathcal{V} \setminus \{\alpha\}) & = & (\texttt{int} \times \texttt{int} \rightarrow \texttt{int}, id, \mathcal{V} \setminus \{\alpha\}) \\
W(x : \alpha \vdash x, id, \mathcal{V} \setminus \{\alpha\}) & = & (\alpha, id, \mathcal{V} \setminus \{\alpha\}) \\
W(x : \alpha \vdash 1, id, \mathcal{V} \setminus \{\alpha\}) & = & (\texttt{int}, id, \mathcal{V} \setminus \{\alpha\}) \\
W(x : \alpha \vdash (x, 1), id, \mathcal{V} \setminus \{\alpha\}) & = & (\alpha \times \texttt{int}, id, \mathcal{V} \setminus \{\alpha\}) \\
W(x : \alpha \vdash +(x, 1), id, \mathcal{V} \setminus \{\alpha\}) & = & (\beta, [\alpha \leftarrow \texttt{int}, \beta \leftarrow \texttt{int}], \mathcal{V} \setminus \{\alpha, \beta\}) \\
W(\emptyset \vdash a, id, \mathcal{V}) & = & (\alpha \rightarrow \beta, [\alpha \leftarrow \texttt{int}, \beta \leftarrow \texttt{int}], \mathcal{V} \setminus \{\alpha, \beta\})
\end{array}$$

(`mgu` is used to unify $\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$ with $\alpha \times \texttt{int} \rightarrow \beta$).

Properties of the algorithm W

Invariant: φ is of the form $\mathtt{mgu}(C)$ for some C; no variable in V is free in C or in Γ.

Theorem 2. [Correctness] *If $W(\Gamma \vdash a)$ terminates in the state (φ, V) and returns τ, then $\varphi(\Gamma) \vdash a : \varphi(\tau)$ is derivable.*

Theorem 3. [Completeness and principality] *Let Γ be a type environment; let (φ_0, V_0) the initial state of the algorithm such that the invariant is satisfied. Suppose we have θ_0 and τ_0 such that $\theta_0\varphi_0(\Gamma) \vdash a : \tau_0 \quad (J_0)$. Then, the execution of $W(\Gamma \vdash a)$ succeeds: let (φ_1, V_1) be the final state and τ_1 the returned type. As the algorithm is correct, we have $\varphi_1(\Gamma) \vdash a : \varphi_1(\tau_1) \quad (J_1)$. Then it exists a substitution θ_1 such that $\theta_0\varphi_0$ and $\theta_1\varphi_1$ coincide out of V_0 and $\tau_0 = \theta_1\varphi_1(\tau_1)$.*

A better approach?

The proof of completeness remained folklore for many years...

The complexity lies in the fact that the algorithm W interleaves the phases of equation generation and equation solving.

Question: Is it possible to separate these two phases (as we did for the algorithm I)?

Answer: Yes! With the system *HM(X)* by Odersky, Sulzmann, and Wehr. Also studied by Skalka and Pottier (and by others).

Polymorphics typing of ML by expansion of `let`

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma \vdash a_2[x \leftarrow a_1] : \tau}{\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}\ \text{(let-subst)}$$

Example:

$$\emptyset \vdash \texttt{let f} = \texttt{fun}\ x \rightarrow x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}$$

because $\emptyset \vdash \texttt{fun}\ x \rightarrow x : \alpha \rightarrow \alpha$, and because $\emptyset \vdash ((\texttt{fun}\ x \rightarrow x)\ 1, (\texttt{fun}\ x \rightarrow x)\ \texttt{true}) : \texttt{int} \times \texttt{bool}$.

We are back to the monomorphic type system...

`let` expansion

Theorem 4. [Expansion of `let`] *The judgement* $\Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau$ *is derivable in polymorphic mini-ML if and only if there exists* τ_1 *such that* $\Gamma \vdash a_1 : \tau_1$ *and* $\Gamma \vdash a_2[x \leftarrow a_1] : \tau$*.*

But...

- explosion of the size of the term;
- very hard to give informative error messages;
- when we add references, the above is no longer true.