TYPE INFERENCE

François Pottier

The Programming Languages Mentoring Workshop @ ICFP
August 30, 2015

What is type inference ?

What is the type of this OCaml function ?

```
let f verbose msg = if verbose then msg else ""
```

What is type inference ?

What is the type of this OCaml function ?

```
let f verbose msg = if verbose then msg else ""
```

OCaml infers it :

```
# let f verbose msg = if verbose then msg else "";;
val f : bool -> string -> string = <fun>
```

What is type inference ?

What is the type of this OCaml function ?

```
let f verbose msg = if verbose then msg else ""
```

OCaml infers it :

```
# let f verbose msg = if verbose then msg else "";;
val f : bool -> string -> string = <fun>
```

Type inference is mostly a matter of finding out the obvious.

Where is type inference ?

Everywhere.

Where is type inference ?

Everywhere.

Every typed programming language has some type inference.

- Pascal, C, etc. have a tiny amount
  - the type of every expression is “inferred” bottom-up
- C++ and Java have a bit more
  - C++ has `auto`, `decltype`, inference of template parameters...
  - Java infers type parameters to method calls and **`new`** (slowly... see next)
- Scala has a lot
  - a form of “local type inference”
  - “bidirectional” (bottom-up in places, top-down in others)
- SML, OCaml, Haskell have a lot, too
  - “non-local” (based on unification / constraint solving)
- Haskell, Scala, Coq, Agda infer not just types, but also terms (that is, code) !

An anecdote

Anyone who has ever used the "diamond" in Java 7...

```
List<Integer> xs =
  new Cons<> (1,
  new Cons<> (1,
  new Cons<> (2,
  new Cons<> (3,
  new Cons<> (3,
  new Cons<> (5,
  new Cons<> (6,
  new Cons<> (6,
  new Cons<> (8,
  new Cons<> (9,
  new Cons<> (9,
  new Cons<> (9,
  new Nil<> ()
  )))))))))))); // Tested with javac 1.8.0_05
```

An anecdote

Anyone who has ever used the “diamond” in Java 7...

```
List<Integer> xs =
  new Cons<> (1, //  0.5 seconds
  new Cons<> (1, //  0.5 seconds
  new Cons<> (2, //  0.5 seconds
  new Cons<> (3, //  0.6 seconds
  new Cons<> (3, //  0.7 seconds
  new Cons<> (5, //  0.9 seconds
  new Cons<> (6, //  1.4 seconds
  new Cons<> (6, //  6.0 seconds
  new Cons<> (8, //  6.5 seconds
  new Cons<> (9, // 10.5 seconds
  new Cons<> (9, // 26   seconds
  new Cons<> (9, // 76   seconds
  new Nil<> ()
  ))))))))))));  // Tested with javac 1.8.0_05
```

... may be interested to hear that this feature seems to have exponential cost.

What is type inference good for ?

How does it work ?

Should I do research in type inference ?

Benefits

What does type inference do for us, programmers ? Obviously,

- it reduces verbosity and redundancy,
- giving us static type checking at little syntactic cost.

Benefits

What does type inference do for us, programmers ? Obviously,

- it reduces verbosity and redundancy,
- giving us static type checking at little syntactic cost.

Less obviously,

- it sometimes helps us figure out what we are doing...

Example : sorting

What is the type of `sort` ?

```
let rec sort (xs : 'a list) =
  if xs = [] then
    []
  else
    let pivot = List.hd xs in
    let xs1, xs2 = List.partition (fun x -> x <= pivot) xs in
    sort xs1 @ sort xs2
```

Example : sorting

What is the type of `sort` ?

```
let rec sort (xs : 'a list) =
  if xs = [] then
    []
  else
    let pivot = List.hd xs in
    let xs1, xs2 = List.partition (fun x -> x <= pivot) xs in
    sort xs1 @ sort xs2
```

Oops... This is a lot more general than I thought ! ?

```
val sort : 'a list -> 'b list
```

This function never returns a non-empty list.

Example : searching a binary search tree

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

What is the type of `find` ?

```
let rec find compare x = function
| Empty -> raise Not_found
| Node(l, v, r) ->
    let c = compare x v in
    if c = 0 then v
    else find compare x (if c < 0 then l else r)
```

Example : searching a binary search tree

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

What is the type of `find` ?

```
let rec find compare x = function
| Empty -> raise Not_found
| Node(l, v, r) ->
    let c = compare x v in
    if c = 0 then v
    else find compare x (if c < 0 then l else r)
```

It may well be more general than you expected :

```
val find : ('a -> 'b -> int) -> 'a -> 'b tree -> 'b
```

Good – this allows us to implement lookup in a map using `find`.

Example : groking delimited continuations

This 1989 paper by Danvy and Filinski...

A Functional Abstraction of Typed Contexts

Olivier Danvy & Andrzej Filinski

DIKU – Computer Science Department, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen Ø, Denmark
uucp: danvy@diku.dk & andrzej@diku.dk

Example : groking delimited continuations

This 1989 paper contains typing rules like this :

$$\frac{\rho, \sigma \vdash \mathtt{E} : \sigma, \tau}{\rho, \alpha \vdash \mathtt{reset(E)} : \tau, \alpha}$$

Example : groking delimited continuations

This 1989 paper contains typing rules like this :

$$\frac{\rho, \sigma \vdash \mathtt{E} : \sigma, \tau}{\rho, \alpha \vdash \mathtt{reset(E)} : \tau, \alpha}$$

and this :

$$\frac{[\mathtt{f} \mapsto (\tau/\delta \rightarrow \alpha/\delta)]\rho, \sigma \vdash \mathtt{E} : \sigma, \beta}{\rho, \alpha \vdash \mathtt{shift\ f\ in\ E} : \tau, \beta}$$

Example : groking delimited continuations

This 1989 paper contains typing rules like this :

$$\frac{\rho, \sigma \vdash \mathtt{E} : \sigma, \tau}{\rho, \alpha \vdash \mathtt{reset(E)} : \tau, \alpha}$$

and this :

$$\frac{[\mathtt{f} \mapsto (\tau/\delta \rightarrow \alpha/\delta)]\rho, \sigma \vdash \mathtt{E} : \sigma, \beta}{\rho, \alpha \vdash \mathtt{shift\ f\ in\ E} : \tau, \beta}$$

How does one make sense of these rules ? How does one guess them ?

Example : groking delimited continuations

Well, the semantics of `shift` and `reset` is known...

```
let return x k = k x
let bind c f k = c (fun x -> f x k)
let reset c = return (c (fun x -> x))
let shift f k = f (fun v -> return (k v)) (fun x -> x)
```

...so their types can be inferred.

Example : groking delimited continuations

Let us introduce a little notation :

```
type ('alpha, 'tau, 'beta) komputation =
  ('tau -> 'alpha) -> 'beta

type ('sigma, 'alpha, 'tau, 'beta) funktion =
  'sigma -> ('alpha, 'tau, 'beta) komputation
```

Example : groking delimited continuations

What should be the typing rule for `reset` ? Ask OCaml :

```
# (reset : (_, _, _) komputation -> (_, _, _) komputation);;
- : ('a, 'a, 'b) komputation -> ('c, 'b, 'c) komputation
```

Example : groking delimited continuations

What should be the typing rule for `reset` ? Ask OCaml :

```
# (reset : (_, _, _) komputation -> (_, _, _) komputation);;
- : ('a, 'a, 'b) komputation -> ('c, 'b, 'c) komputation
```

So Danvy and Filinski were right :

$$\frac{\rho, \sigma \vdash \texttt{E} : \sigma, \tau}{\rho, \alpha \vdash \texttt{reset(E)} : \tau, \alpha}$$

(`'a` is σ, `'b` is τ, `'c` is α.)

Example : groking delimited continuations

What should be the typing rule for `shift` ? Ask OCaml :

```
# (shift : ((_, _, _, _) funktion -> (_, _, _) komputation) ->
                                         (_, _, _) komputation);;
- : (('a, 'b, 'c, 'b) funktion -> ('d, 'd, 'e) komputation) ->
                                     ('c, 'a, 'e) komputation
```

Example : groking delimited continuations

What should be the typing rule for `shift` ? Ask OCaml :

```
# (shift : ((_, _, _, _) funktion -> (_, _, _) komputation) ->
                                     (_, _, _) komputation);;
- : (('a, 'b, 'c, 'b) funktion -> ('d, 'd, 'e) komputation) ->
                                  ('c, 'a, 'e) komputation
```

So Danvy and Filinski were right :

$$\frac{[\mathtt{f} \mapsto (\tau/\delta \to \alpha/\delta)]\rho, \sigma \vdash \mathtt{E} : \sigma, \beta}{\rho, \alpha \vdash \mathtt{shift\ f\ in\ E} : \tau, \beta}$$

(`'a` is τ, `'b` is δ, `'c` is α, `'d` is σ, `'e` is β.)

Bottom line

Sometimes, type inference helps us figure out what we are doing.

Drawbacks

In what ways could type inference be a bad thing ?

- Liberally quoting Reynolds (1985), type inference allows us to make code succinct to the point of unintelligibility.
- Reduced redundancy makes it harder for the machine to locate and explain type errors.

Both issues can be mitigated by adding well-chosen type annotations.

What is type inference good for ?

How does it work ?

Should I do research in type inference ?

A look at simple type inference

Let us focus on a simply-typed programming language.

- base types (`int`, `bool`, ...), function types (`int -> bool`, ...), pair types, etc.
- no polymorphism, no subtyping, no nuthin'

Type inference in this setting is particularly simple and powerful.

Simple type inference, very informally

```
let f verbose msg = if verbose then msg else ""
```

Simple type inference, very informally

```
let f verbose msg = if verbose then msg else ""
```

Say `f` has unknown type α.

Simple type inference, very informally

```
let f verbose msg = if verbose then msg else ""
```

Say `f` has unknown type α.
`f` is a function of two arguments.

Simple type inference, very informally

```
let f verbose msg = if verbose then msg else ""
```

Say `f` has unknown type α.
`f` is a function of two arguments. So $\alpha = \alpha_1 \rightarrow \alpha_2 \rightarrow \beta$.

Simple type inference, very informally

```
let f verbose msg = if verbose then msg else ""
```

Say `f` has unknown type α.
`f` is a function of two arguments. So $\alpha = \alpha_1 \rightarrow \alpha_2 \rightarrow \beta$.
`verbose` has type α_1.

Simple type inference, very informally

```
let f verbose msg = if verbose then msg else ""
```

Say `f` has unknown type α.
`f` is a function of two arguments. So $\alpha = \alpha_1 \to \alpha_2 \to \beta$.
`verbose` has type α_1.
`msg` has type α_2.

Simple type inference, very informally

```
let f verbose msg = if verbose then msg else ""
```

Say `f` has unknown type α.
`f` is a function of two arguments. So $\alpha = \alpha_1 \to \alpha_2 \to \beta$.
`verbose` has type α_1.
`msg` has type α_2.
The "`if`" expression must have type β.

Simple type inference, very informally

```
let f verbose msg = if verbose then msg else ""
```

Say `f` has unknown type α.

`f` is a function of two arguments. So $\alpha = \alpha_1 \to \alpha_2 \to \beta$.

`verbose` has type α_1.

`msg` has type α_2.

The “`if`” expression must have type β. So $\alpha_1 =$ `bool`.

Simple type inference, very informally

```
let f verbose msg = if verbose then msg else ""
```

Say `f` has unknown type α.

`f` is a function of two arguments. So $\alpha = \alpha_1 \rightarrow \alpha_2 \rightarrow \beta$.

`verbose` has type α_1.

`msg` has type α_2.

The "`if`" expression must have type β. So $\alpha_1 = \texttt{bool}$.

And $\alpha_2 = \texttt{string} = \beta$.

Simple type inference, very informally

```
let f verbose msg = if verbose then msg else ""
```

Say `f` has unknown type α.

`f` is a function of two arguments. So $\alpha = \alpha_1 \rightarrow \alpha_2 \rightarrow \beta$.

`verbose` has type α_1.

`msg` has type α_2.

The "`if`" expression must have type β. So $\alpha_1 =$ `bool`.
And $\alpha_2 =$ `string` $= \beta$.

Solving these equations reveals that `f` has type `bool -> string -> string`.

A partial history of simple type inference

Let us see how it has been explained / formalized through history...

The 1970s

The 1970s

A Theory of Type Polymorphism in Programming

Robin Milner

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

Milner (1978) invents type inference and ML polymorphism.

He re-discovers, extends, and popularizes an earlier result by Hindley (1969).

Milner's description

Milner publishes a "declarative" presentation, Algorithm W,

Milner’s description

Milner publishes a “declarative” presentation, Algorithm W,

(ii) If f is (de), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$;
let $U = \mathscr{U}(S\rho, \sigma \to \beta)$, β new;
then $T = USR$, and $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

Milner's description

Milner publishes a "declarative" presentation, Algorithm W,

and an "imperative" one, Algorithm J.

(ii) If f is (de), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$;
let $U = \mathscr{U}(S\rho, \sigma \to \beta)$, β new;
then $T = USR$, and $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

Milner's description

Milner publishes a "declarative" presentation, Algorithm W,

and an "imperative" one, Algorithm J.

(ii) If f is (de), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$;
let $U = \mathscr{U}(S\rho, \sigma \rightarrow \beta)$, β new;
then $T = USR$, and $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

(ii) If f is (de) then:

$\rho := \mathscr{J}(\bar{p}, d)$; $\sigma := \mathscr{J}(\bar{p}, e)$;
UNIFY $(\rho, \sigma \rightarrow \beta)$; ($\beta$ new)
$\tau := \beta$

Milner's description

Milner publishes a "declarative" presentation, Algorithm W,

(ii) If f is (de), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$;
let $U = \mathscr{U}(S\rho, \sigma \to \beta)$, β new;
then $T = USR$, and $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

and an "imperative" one, Algorithm J.

(ii) If f is (de) then:

$\rho := \mathscr{J}(\bar{p}, d)$; $\sigma := \mathscr{J}(\bar{p}, e)$;
UNIFY $(\rho, \sigma \to \beta)$; (β new)
$\tau := \beta$

Algorithm J maintains a current substitution in a global variable.

Milner's description

Milner publishes a "declarative" presentation, Algorithm W,

and an "imperative" one, Algorithm J.

Algorithm J maintains a current substitution in a global variable.

Both compose substitutions produced by unification, and create "new" variables as needed.

(ii) If f is (de), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$;
let $U = \mathscr{U}(S\rho, \sigma \to \beta)$, β new;
then $T = USR$, and $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

(ii) If f is (de) then:

$\rho := \mathscr{J}(\bar{p}, d)$; $\sigma := \mathscr{J}(\bar{p}, e)$;
UNIFY $(\rho, \sigma \to \beta)$; (β new)
$\tau := \beta$

Milner's description

Milner publishes a "declarative" presentation, Algorithm W,

and an "imperative" one, Algorithm J.

Algorithm J maintains a current substitution in a global variable.

Both compose substitutions produced by unification, and create "new" variables as needed.

(ii) If f is (de), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$;
let $U = \mathscr{U}(S\rho, \sigma \to \beta)$, β new;
then $T = USR$, and $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

(ii) If f is (de) then:

$\rho := \mathscr{J}(\bar{p}, d)$; $\sigma := \mathscr{J}(\bar{p}, e)$;
UNIFY $(\rho, \sigma \to \beta)$; (β new)
$\tau := \beta$

Milner does not describe UNIFY.

Naive unification (Robinson, 1965) has exponential complexity due to lack of sharing.

The 1980s

The 1980s

A Simple Algorithm and Proof for Type Inference

Mitchell Wand*

College of Computer Science
Northeastern University

Cardelli (1987), Wand (1987) and others formulate type inference as a two-stage process : generating and solving a conjunction of equations.

Case 3. $(A, (\lambda x.M), t)$. Let τ_1 and τ_2 be fresh type variables. Generate the equation $t = \tau_1 \rightarrow \tau_2$ and the subgoal $((A[x \leftarrow \tau_1])_M, M, \tau_2)$.

This leads to a higher-level, more modular presentation, which matches the informal explanation.

The 1990s

The 1990s

$$\frac{\alpha \doteq e \wedge \alpha \doteq e'}{\alpha \doteq e \doteq e'} \rightsquigarrow \quad (\textsc{Fuse}) \qquad \frac{f(\tau_1, \dots \tau_p) \doteq f(\beta_1, \dots \beta_p) \doteq e}{\tau_1 \doteq \beta_1 \wedge \dots \tau_p \doteq \beta_p \wedge f(\beta_1, \dots \beta_p) \doteq e} \rightsquigarrow$$

$$\text{if } f \neq g, \qquad \frac{f(\tau_1, \dots \tau_p) \doteq g(\sigma_1, \dots \sigma_q) \doteq e}{\bot} \rightsquigarrow \quad (\textsc{Fail})$$

$$\text{if } \alpha \in \mathcal{V}(e) \setminus e \setminus \mathcal{V}(\tau) \wedge \tau \notin \mathcal{V}, \qquad \frac{(\alpha \mapsto \tau)(e)}{\exists \alpha \cdot (e \wedge \alpha \doteq \tau)} \rightsquigarrow \quad (\textsc{Generalize})$$

Kirchner & Jouannaud (1990), Rémy (1992) and others push this approach further.

- They explain constraint solving as rewriting.
- They explain sharing by using variables as memory addresses.
- They explain “new” variables as existential quantification.

Constraints

An intermediate language for describing type inference problems.

$$
\begin{aligned}
\tau &::= \alpha \mid \tau \to \tau \mid \dots \\
C &::= \bot \mid \tau = \tau \mid C \wedge C \mid \exists \alpha . C
\end{aligned}
$$

A constraint generator transforms the program into a constraint.

A constraint solver determines whether the constraint is satisfiable (and computes a description of its solutions).

Constraint generation

A function of a type environment Γ, a term t, and a type τ to a constraint.

Defined by cases :

$$
\begin{array}{rcl}
[\![\Gamma \vdash x : \tau]\!] & = & (\Gamma(x) = \tau) \\
[\![\Gamma \vdash \lambda x.u : \tau]\!] & = & \exists \alpha_1 \alpha_2. \left(\begin{array}{l} \tau = \alpha_1 \rightarrow \alpha_2 \wedge \\ [\![\Gamma[x \mapsto \alpha_1] \vdash u : \alpha_2]\!] \end{array} \right) \\
[\![\Gamma \vdash t_1\ t_2 : \tau]\!] & = & \exists \alpha.([\![\Gamma \vdash t_1 : \alpha \rightarrow \tau]\!] \wedge [\![\Gamma \vdash t_2 : \alpha]\!])
\end{array}
$$

Constraint solving as rewriting

Transform the constraint, step by step, obeying a set of rewriting rules.

If :

- every rewriting step preserves the meaning of the constraint,
- every sequence of rewriting steps terminates,
- a constraint that cannot be further rewritten either is $\bot$ or is satisfiable,

then we have a solver, i.e., an algorithm for deciding satisfiability.

Variables as addresses

A new variable α can be introduced to stand for a sub-term τ :

$$\frac{(\alpha \mapsto \tau)(e)}{\exists\, \alpha \cdot\ (e \wedge \alpha \doteq \tau)} \rightsquigarrow \quad (\textsc{Generalize})$$

Think of α as the address of τ in the machine.

Instead of duplicating a whole sub-term, one duplicates its address :

$$\frac{f(\tau_1, \ldots \tau_p) \doteq f(\beta_1, \ldots \beta_p) \doteq e}{\tau_1 \doteq \beta_1 \wedge \ldots \tau_p \doteq \beta_p \wedge f(\beta_1, \ldots \beta_p) \doteq e} \rightsquigarrow \quad (\textsc{Decompose})$$

This accounts for sharing. Robinson's exponential blowup is avoided.

Unification, the right way

Rémy works with multi-equations, equations with more than two members :

$$\frac{\alpha \doteq e \wedge \alpha \doteq e'}{\alpha \doteq e \doteq e'} \quad (\textsc{Fuse})$$

In the machine, one maintains equivalence classes of variables using a union-find data structure.

The occurs-check (which detects cyclic equations) takes place once at the end. (Doing it at every step, like Robinson, would cause a quadratic slowdown.)

This is Huet's quasi-linear-time unification algorithm (1976).

What is type inference good for ?

How does it work ?

Should I do research in type inference ?

A takeaway message

Just as in a compiler, an intermediate language is a useful abstraction.

- think declarative, not imperative
- say what you want computed, not how to compute it
- build a constraint, then "optimize" it step by step until it is solved

The constraint-based approach scales up and handles

- Hindley-Milner polymorphism (Pottier and Rémy, 2005)
- elaboration (Pottier, 2014)
- type classes, OCaml objects, and more.

Is type inference a hot topic?

Not really. Not at this moment.

Is type inference a hot topic ?

Not really. Not at this moment.

At ICFP 2015, 4 out of 35 papers seem directly or indirectly concerned with it :

- 1ML – Core and Modules United (F-ing First-Class Modules) (Rossberg)
- Bounded Refinement Types (Vazou, Bakst, Jhala)
- A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading (Ziliani, Sozeau)
- Practical SMT-Based Type Error Localization (Pavlinovic, King, Wies)

Is type inference a hot topic ?

Not really. Not at this moment.

At ICFP 2015, 4 out of 35 papers seem directly or indirectly concerned with it :

- 1ML – Core and Modules United (F-ing First-Class Modules) (Rossberg)
- Bounded Refinement Types (Vazou, Bakst, Jhala)
- A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading (Ziliani, Sozeau)
- Practical SMT-Based Type Error Localization (Pavlinovic, King, Wies)

Yet, people still get drawn into it by necessity.

- remember, every typed programming language needs some type inference !

What are the open problems ?

Inference for powerful / complex type systems.

- universal and existential types
- dependent types (Ziliani and Sozeau) and refinement types (Vazou et al.)
- linear and affine types
- subtyping
- first-class modules (Rossberg)

Inference for tricky / ugly languages.

- e.g., JavaScript – which was not designed as a typed language, to begin with

Locating and explaining type errors.

- show all locations, or a most likely one ? (Pavlinovic et al.)

Identifying re-usable building blocks for type inference algorithms.

What's the potential impact ?

Type inference makes the difference between an awful language and a great one.

- if you care about language design, you will care about type inference

What are the potential pitfalls ?

Type inference is (often) an undecidable problem.

Type error explanation is (often) an ill-specified problem.

Your algorithm may “work well in practice”,

- but it could be difficult to formally argue that it does,
- hence difficult to publish.

YOU TOO COULD BE SUCKED INTO IT.

GOOD LUCK and HAVE FUN !