Hindley-Milner elaboration in applicative style

François Pottier

This pearl presents

This pearl presents

a (shamefully) simple *solution*

This pearl presents

a (shamefully) simple *solution*

to a *problem* that has (gently) troubled me for ten years

This pearl presents

a (shamefully) simple *solution*

to a *problem* that has (gently) troubled me for ten years

and whose *story* begins even longer ago.

Part I

A STORY

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 ML *polymorphism* and *type inference*.

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 $U = \mathscr{U}(S\rho, \sigma \to \beta)$, 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 E.

(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*,

and an imperative one, *Algorithm J*.

Algorithm J maintains a *"current substitution"* in a global variable E.

(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)
$\tau := \beta$

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

The 1980s

The 1980s

A Simple Algorithm and Proof for Type Inference

Mitchell Wand*

College of Computer Science
Northeastern University

Cardelli, 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 $((A[x \leftarrow \tau_1])_M, M, \tau_2)$.

Benefits

Higher-level thinking:

instead of *substitutions* and *composition*,
equations and *conjunction*.

Greater modularity:

constraints and constraint solving as a *library*,
constraint generation performed by the *user*.

Limitations

New variables still created via a *global side effect*.

Polymorphic type inference *not supported*.

> Algorithm J must *solve* the constraints produced so far
> (it looks up E) before it can *produce* more constraints.

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 explain "new" variables as *existential quantification* and constraint solving as *rewriting*.

A necessary step on the road towards explaining *polymorphic* inference.

The 2000s

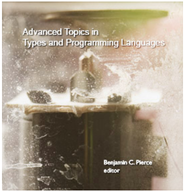

inria
INVENTEURS DU MONDE NUMÉRIQUE
CENTRE DE RECHERCHE
PARIS
ROCQUENCOURT

The 2000s

Constraint Abstractions

Jörgen Gustavsson and Josef Svenningsson

Chalmers University of Technology and Göteborg University
{gustavss,josefs}@cs.chalmers.se



Following Gustavsson and Svenningsson (2001), Didier Rémy and F.P. (2005) explain polymorphic type inference using *constraint abstractions*.

Constraints

Constraints offer a syntax for describing type inference problems.

$$
\begin{array}{rcll}
\tau & ::= & \alpha \mid \tau \rightarrow \tau \mid \ldots & \\
C & ::= & \text{false} \mid \text{true} \mid C \wedge C \mid \tau = \tau \mid \exists \alpha . C & \text{(unification)} \\
 & & \mid \text{let } x = \lambda \alpha . C \text{ in } C & \textit{(abstraction)} \\
 & & \mid x\ \tau & \textit{(application)}
\end{array}
$$

The meaning of let-constraints is given by the law:

$$
\begin{array}{rl}
 & \text{let } x = \lambda \alpha . C_1 \text{ in } C_2 \\
\equiv & \exists \alpha . C_1 \wedge [\lambda \alpha . C_1 / x] C_2
\end{array}
$$

Constraint generation

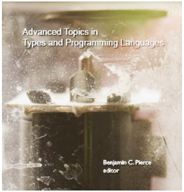
A pure function of a term t and a type τ to a constraint $[\![t : \tau]\!]$.

$$
\begin{aligned}
[\![x : \tau]\!] &= x\ \tau \\
[\![\lambda x.u : \tau]\!] &= \exists \alpha_1 \alpha_2. \left(\begin{array}{l} \tau = \alpha_1 \to \alpha_2 \wedge \\ \mathsf{let}\ x = \lambda \alpha.(\alpha = \alpha_1)\ \mathsf{in}\ [\![u : \alpha_2]\!] \end{array} \right) \\
[\![t_1\ t_2 : \tau]\!] &= \exists \alpha.([\![t_1 : \alpha \to \tau]\!] \wedge [\![t_2 : \alpha]\!]) \\
[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : \tau]\!] &= \mathsf{let}\ x = \lambda \alpha.[\![t_1 : \alpha]\!]\ \mathsf{in}\ [\![t_2 : \tau]\!]
\end{aligned}
$$

Constraint solving

On paper, every constraint can be *rewritten* step by step to either false or a solved form.

The imperative implementation, based on Huet's unification algorithm and Rémy's ranks, is *efficient* (McAllester, 2003).



Library (OCaml)

Abstract syntax for constraints:

```
type variable
val fresh: variable structure option -> variable
type rawco =
| CTrue
| CConj  of rawco * rawco
| CEq    of variable * variable
| CExist of variable * rawco
| ...
```

Combinators that build constraints:

```
val truth: rawco
val (^&) : rawco -> rawco -> rawco
val (--) : variable -> variable -> rawco
val exist: (variable -> rawco) -> rawco
...
```

User (OCaml)

The user defines constraint generation:

```
let rec hastype (t : ML.term) (w : variable) : rawco
= match t with
  | ...
  | ML.Abs (x, u) ->
      exist (fun v1 ->
        exist (fun v2 ->
          w --- arrow v1 v2 ^&
          def x v1 (hastype u v2)
        )
      )
  | ...

let iswelltyped (t : ML.term) : rawco
= exist (fun w -> hastype t w)
```

Part II

A PROBLEM

A problem

Submitting a *closed* ML term to the generator ...

```
let b = if x = y then
... else ... in ...
```

A problem

Submitting a *closed* ML term to the generator ...	`let b = if x = y then ... else ... in ...`
yields a *closed* constraint ...	$\exists\alpha.(\alpha = \mathsf{bool} \wedge \exists\beta\gamma.(\ldots))$

A problem

Submitting a *closed* ML term to the generator ...

```
let b = if x = y then
...  else ...  in ...
```

yields a *closed* constraint ...

$\exists\alpha.(\alpha = \text{bool} \wedge \exists\beta\gamma.(\ldots))$

which the solver rewrites to ...

A problem

Submitting a *closed* ML term to the generator ...	`let b = if x = y then ... else ... in ...`
yields a *closed* constraint ...	$\exists\alpha.(\alpha = \text{bool} \wedge \exists\beta\gamma.(\ldots))$
which the solver rewrites to ...	either false, or true.

A problem (OCaml)

The API offered by the library is *too simple:*

```
val solve:          rawco -> bool
```

(Ignoring type error diagnostics.)

A problem (OCaml)

The API offered by the library is *too simple:*

```
val solve:          rawco -> bool
```

(Ignoring type error diagnostics.) The user has defined:

```
val iswelltyped: ML.term -> rawco
```

A problem (OCaml)

The API offered by the library is *too simple:*

```
val solve:          rawco -> bool
```

(Ignoring type error diagnostics.) The user has defined:

```
val iswelltyped: ML.term -> rawco
```

There is no way of obtaining, say:

```
val elaborate:    ML.term -> F.term
```

which would be the front-end of a *type-directed* compiler.

Question

Can one perform *elaboration*

without compromising the *modularity* and *elegance*

of the constraint-based approach?

Part III

A SOLUTION

A low-level solution

The generator could produce a pair of

a *constraint* and

a *template for an elaborated term*,

sharing mutable placeholders for evidence,

so that, after the constraint is *solved*,

the template can be *"solidified"* into an elaborated term.

Library, low-level (OCaml)

Constraints already contain mutable placeholders for evidence:

```
... | CExist of variable * rawco | ...
```

More placeholders (not shown) required to deal with polymorphism.

Let the library offer a type decoder, which can be invoked *after* solving:

```
type decoder = variable -> ty
val new_decoder: unit -> decoder
...
```

User (OCaml)

The user could write:

```
val hastype:
  ML.term -> variable -> rawco * F.template
val solidify:
  F.template -> F.term
```

where:

- the constraint and the template *share* variables,
- `solidify` uses a type decoder to replace these variables with types.

Why I not am happy with stopping here

This approach is in three stages: generation, solving, solidification.
Each user construct is dealt with *twice,* in stages 1 and 3.

This approach *exposes evidence* to the user.
Evidence is mutable and involves names and binders.

One needs an intermediate representation `F.template`,
or one must pollute `F.term`.

A wish

Even though stages 1 and 3 must be *executed* separately,

the user would prefer to *describe* them in a unified manner.

A dream

If the user could somehow (magically?)

> *construct* the constraint, and "simultaneously"
>
> *query* the solver for the *final* (decoded) witness for a variable

then she would be able to perform elaboration in one swoop:

```
val elaborate: ML.term -> F.term
```

and evidence would not need to be exposed.

The idea

Give the user the *illusion* that she can use the solver in this manner.

Give her a DSL to express *computations* that:

emit constraints *and*

read their solutions.

The idea

Give the user the *illusion* that she can use the solver in this manner.

Give her a DSL to express *computations* that:

> emit constraints *and*
>
> read their solutions.

It turns out that this DSL is just

> our good old *constraints*,
>
> extended with a `map` combinator.

Library, high-level (OCaml)

Solving/evaluating a constraint *produces a result*.

```
type 'a co
val solve: 'a co -> 'a

val pure: 'a -> 'a co
val (^&): 'a co -> 'b co -> ('a * 'b) co
val map: ('a -> 'b) -> 'a co -> 'b co

val (--): variable -> variable -> unit co
val exist: (variable -> 'a co) -> (ty * 'a) co
...
```

E.g., evaluating $\exists\alpha.C$ yields a *pair* of a decoded type (the *witness* for α) and the value of C.

Library, high-level (OCaml)

This is implemented *on top of* the earlier, low-level library.

```
type env =
  decoder
type 'a co =
  rawco * (env -> 'a)
```

A constraint/computation is a pair of

- a raw *constraint,* which contains mutable evidence;
- a *continuation,* which reads this evidence *after* the solver has run.

Library, high-level (OCaml)

The implementation is quasi-trivial.

```
let exist f =
  let v = fresh None in
  let rc, k = f v in
  CExist (v, rc),
  fun env ->
    let decode = env in
    (decode v, k env)
```

User (OCaml)

The user defines inference/elaboration in *one* inductive function:

```
let rec hastype t w : F.term co
= match t with
  | ...
  | ML.Abs (x, u) ->
      exist (fun v1 ->
        exist (fun v2 ->
          w --- arrow v1 v2 ^&
          def x v1 (hastype u v2)
        )
      ) <$$> fun (ty1, (ty2, ((), u'))) ->
      F.Abs (x, ty1, u')
  | ...
```

The (final, decoded) type `ty1` of `x` seems to be *magically* available.

Remarks

Elaboration from ML to System F in the paper (and online).

The type `'a co` forms an *applicative functor*, not a monad.

Part IV

Conclusion

Conclusion

- a simple idea, really
- just icing on the cake
- modularity, elegance, performance
- usable in other settings? e.g. higher-order pattern unification?

Thank you

`http://gallium.inria.fr/~fpottier/inferno/`

No mutable state was exposed in the making of this library.