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 \textit{solution}

to a \textit{problem} that has (gently) troubled me for ten years

and whose \textit{story} begins even longer ago.
Part I

A STORY
The 1970s

Milner (1978) invents ML polymorphism and type inference.
The 1970s

Milner (1978) invents ML *polymorphism* and *type inference*.
Milner’s description

Milner’s description

Milner publishes a declarative presentation, \textit{Algorithm W},

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

\begin{itemize}
\item let $(R, d_p) = \mathcal{W}(\bar{p}, d)$, and $(S, e_o) = \mathcal{W}(\bar{R}\bar{p}, e)$;
\item let $U = \mathcal{U}(S\rho, \sigma \rightarrow \beta), \beta \text{ new};$
\item then $T = USR$, and $f = U(((Sd)\bar{e})_\beta)$.
\end{itemize}
Milner’s description


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

\[
\begin{align*}
&\text{let } (R, d_p) = \mathcal{W}(\bar{p}, d), \text{ and } (S, e_o) = \mathcal{W}(R\bar{p}, e); \\
&\text{let } U = \mathcal{U}(S\rho, \sigma \rightarrow \beta), \beta \text{ new}; \\
&\text{then } T = \text{USR}, \text{ and } f = U((S\bar{d})e)_\beta.
\end{align*}
\]
Milner’s description


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

\[
\begin{align*}
& \text{let } (R, d) = \text{W}(\tilde{p}, d), \text{ and } (S, e) = \text{W}(R\tilde{p}, e); \\
& \text{let } U = \text{USR}(S, \sigma \rightarrow \beta), \beta \text{ new}; \\
& \text{then } T = USR, \text{ and } f = U(((Sd)e)_{\beta}).
\end{align*}
\]

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

\[
\begin{align*}
& \rho := \text{J}(\tilde{p}, d); \sigma := \text{J}(\tilde{p}, e); \\
& \text{UNIFY } (\rho, \sigma \rightarrow \beta); (\beta \text{ new}) \\
& \tau := \beta
\end{align*}
\]
Milner’s description

Milner publishes a declarative presentation, \textit{Algorithm W}, and an imperative one, \textit{Algorithm J}.

Algorithm J maintains a \textit{“current substitution”} in a global variable $E$.

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

\begin{align*}
&\text{let } (R, d') = \mathcal{W}(\bar{p}, d), \text{ and } (S, \bar{e}_o) = \mathcal{W}(R\bar{p}, e); \\
&\text{let } U = \mathcal{U}(S\rho, \sigma \rightarrow \beta), \beta \text{ new}; \\
&\text{then } T = USR, \text{ and } f = U(((S\bar{d})\bar{e}_o)).
\end{align*}

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

\begin{align*}
\rho &:= \mathcal{J}(\bar{p}, d); \sigma := \mathcal{J}(\bar{p}, e); \\
\text{UNIFY } (\rho, \sigma \rightarrow \beta); \beta \text{ new} \\
\tau &:= \beta
\end{align*}
Milner’s description


Algorithm J maintains a “current substitution” in a global variable $E$.

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

let $(R, d_n) = \mathcal{W}(\bar{p}, d)$, and $(S, \bar{e}_o) = \mathcal{W}(R\bar{p}, e)$;

let $U = \mathcal{U}(S\rho, \sigma \rightarrow \beta)$, $\beta$ new;

then $T = USR$, and $f = U(((S\bar{d})\bar{e})_\beta)$.

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

$\rho := \mathcal{J}(\bar{p}, d)$; $\sigma := \mathcal{J}(\bar{p}, e)$;

UNIFY $(\rho, \sigma \rightarrow \beta)$; ($\beta$ new)

$\tau := \beta$

Both compose substitutions produced by unification, and create “new” variables as needed.
The 1980s

Cardelli, Wand (1987) and others formulate type inference as a two-stage process: generating and solving a conjunction of equations.
The 1980s

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 \(\tau_1\) and \(\tau_2\) 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

Kirchner & Jouannaud (1990), Remy (1992) and others explain "new" variables as existential quantification and constraint solving as rewriting. A necessary step on the road towards explaining polymorphic inference.
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

The 2000s

Constraint Abstractions

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

Constraints

Constraints offer a syntax for describing type inference problems.

\[
\tau ::= \alpha \mid \tau \rightarrow \tau \mid \ldots \\
C ::= \text{false} \mid \text{true} \mid C \land C \mid \tau =\tau \mid \exists \alpha. C \quad \text{(unification)} \\
\mid \text{let } x = \lambda \alpha. C \text{ in } C \quad \text{(abstraction)} \\
\mid x \tau \quad \text{(application)}
\]

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

\[
\text{let } x = \lambda \alpha. C_1 \text{ in } C_2 \\
\equiv \exists \alpha. C_1 \land [\lambda \alpha. C_1 / x] C_2
\]
A pure function of a term $t$ and a type $\tau$ to a constraint $\llbracket t : \tau \rrbracket$.

\[
\llbracket x : \tau \rrbracket = x \, \tau
\]
\[
\llbracket \lambda x. u : \tau \rrbracket = \exists \alpha_1 \alpha_2. \left( \begin{array}{c}
\tau = \alpha_1 \to \alpha_2 \land \\
\text{let } x = \lambda \alpha.(\alpha = \alpha_1) \text{ in } \llbracket u : \alpha_2 \rrbracket
\end{array} \right)
\]
\[
\llbracket t_1 \, t_2 : \tau \rrbracket = \exists \alpha. (\llbracket t_1 : \alpha \to \tau \rrbracket \land \llbracket t_2 : \alpha \rrbracket)
\]
\[
\llbracket \text{let } x = t_1 \text{ in } t_2 : \tau \rrbracket = \text{let } x = \lambda \alpha. \llbracket t_1 : \alpha \rrbracket \text{ in } \llbracket t_2 : \tau \rrbracket
\]
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).
Abstract syntax for constraints:

```ocaml
type variable
val fresh : variable structure option \rightarrow variable
type rawco =
  | CTrue
  | CConj of rawco * rawco
  | CEq of variable * variable
  | CExist of variable * rawco
  | ...
```

Combinators that build constraints:

```ocaml
val truth : rawco
val (^&) : rawco \rightarrow rawco \rightarrow rawco
val (--) : variable \rightarrow variable \rightarrow rawco
val exist : (variable \rightarrow rawco) \rightarrow rawco
...
```
The user defines constraint generation:

```ocaml
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 iswelltypetd (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 ...

yields a *closed* constraint ...

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

∃α.(α = bool ∧ ∃βγ.(...))
```
A problem

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

yields a *closed* constraint ...

which the solver rewrites to ...

let \( b = \) if \( x = y \) then ...

... else ...

in ...

\( \exists \alpha. (\alpha = \text{bool} \land \exists \beta \gamma. (...) \)
A problem

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

yields a *closed* constraint ...

which the solver rewrites to ...

let \( b = \text{if} \ x = y \ \text{then} \ ... \ \text{else} \ ... \ \text{in} \ ...

\exists \alpha. (\alpha = \text{bool} \land \exists \beta \gamma. (...) )

either false, or true.
A problem (OCaml)

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

```ocaml
define solve : rawco -> bool =

(Ignoring type error diagnostics.)
```
The API offered by the library is *too simple*:

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

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

```ocaml
val iswelltyped : ML.term -> rawco
```
A problem (OCaml)

The API offered by the library is *too simple*:
```ocaml
val solve : rawco -> bool
```
(Ignoring type error diagnostics.) The user has defined:
```ocaml
val iswelltyped : ML.term -> rawco
```
There is no way of obtaining, say:
```ocaml
val elaborate : ML.term -> F.term
```
which would be the front-end of a *type-directed* compiler.
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.
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:

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

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:

```plaintext
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.
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 $\alpha$) and the value of $C$. 
This is implemented *on top of* the earlier, low-level library.

```ocaml
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.

```ocaml
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)
```
The user defines inference/elaboration in one inductive function:

```ocaml
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.