Ambivalent Types for Principal Type Inference with GADTs

Didier Rémy

(Joint work with Jacques Garrigue)

IFIP WG2.8, Aussois, October 2013

GADTs

Similar to inductive types of Coq *et al*.

```
type  α exp =
  | Int : int → int exp
  | Add : (int → int → int) exp
  | App : (α → β) exp * α exp → β exp

App (Add, Int 3) : (int → int) exp
```

Enable to express invariants and proofs.

Also provide existential types:

$$\begin{array}{rcl} \mathrm{App} & : & \forall \alpha\beta.\ \big((\alpha \to \beta)\ \mathit{exp} \times \alpha\ \mathit{exp}\big) \to \beta\ \mathit{exp} \\ & \approx & \forall \beta.\ \big(\exists \alpha.\ (\alpha \to \beta)\ \mathit{exp} \times \alpha\ \mathit{exp}\big) \to \beta\ \mathit{exp} \end{array}$$

Available in Haskell for many years, in OCaml since last year.

This presents the solution now in use in OCaml.

Type checking of GADTs is easy

Matching on a constructor introduces local equations.

These equations are visible in the body of the case

```
let rec eval (type a) (x : a exp) : a =
  match x with
  | Int n          → n
  | Add                        → (+)
  | App (f, x)                       →
        eval (f) (eval x)
```

This is the source program.

Type checking of GADTs is easy

Matching on a constructor introduces local equations.

These equations are visible in the body of the case

```
let rec eval (type a) (x : a exp) : a =
  match x with
  | Int n ⟨a = int⟩ → n
  | Add             → (+)
  | App (f, x)      →
        eval (f) (eval x)
```

An equation is introduced when we enter the branch.

Type checking of GADTs is easy

Matching on a constructor introduces local equations.

These equations are visible in the body of the case

```
let rec eval (type a) (x : a exp) : a =
  match x with
  | Int n ⟨a = int⟩ → n : int ≈ a
  | Add             → (+)
  | App (f, x)            →
        eval (f) (eval x)
```

Variable n has type n which, by the equation, is equal to type a.

Type checking of GADTs is easy

Matching on a constructor introduces local equations.

These equations are visible in the body of the case

```
let rec eval (type a) (x : a exp) : a =
  match x with
  | Int n           → n
  | Add ⟨a = int → int → int⟩ → (+) : int → int → int ≈ a
  | App (f, x)                         →
        eval (f) (eval x)
```

Similarly for the other branches.

Type checking of GADTs is easy

Matching on a constructor introduces local equations.

These equations are visible in the body of the case

```
let rec eval (type a) (x : a exp) : a =
  match x with
  | Int n          → n
  | Add                          → (+)
  | App (f, x) ⟨∃β, f : β → a ∧ x : β⟩ →
        eval (f : (β → a) exp) (eval x : β exp) : a exp
```

Similarly for the other branches.

But type inference difficult...

Matching on a constructor introduces local equations.

These equations are visible in the body of the case

```
let rec eval (type a) (x : a exp)      =
  match x with
  | Int n          → n
  | Add                    → (+)
  | App (f, x)                    →
       eval (f) (eval x)
```

If the return type of the match is not given, what should it be?

But type inference difficult...

Matching on a constructor introduces local equations.

These equations are visible in the body of the case

```
let rec eval (type a) (x : a exp)      =
  match x with
  | Int n          → n : int
  | Add                     → (+)
  | App (f, x)                     →
        eval (f) (eval x)
```

If the return type of the match is not given, what should it be?

- *int* in the first branch,

But type inference difficult...

Matching on a constructor introduces local equations.

These equations are visible in the body of the case

```
let rec eval (type a) (x : a exp)      =
  match x with
  | Int n          → n : int
  | Add                        → (+) : int → int → int
  | App (f, x)                         →
       eval (f) (eval x)
```

If the return type of the match is not given, what should it be?

- *int* in the first branch, but it will later clash with $int \to int \to int$.

But type inference difficult...

Matching on a constructor introduces local equations.

These equations are visible in the body of the case

```
let rec eval (type a) (x : a exp)       =
  match x with
  | Int n ⟨a = int⟩ → n
  | Add                      → (+)
  | App (f, x)                     →
        eval (f) (eval x)
```

If the return type of the match is not given, what should it be?

- Use the equation $a = int$ in the branch, but ...

But type inference difficult...

Matching on a constructor introduces local equations.

These equations are visible in the body of the case

```
let rec eval (type a) (x : a exp)      =
  match x with
  | Int n ⟨a = int⟩ → n : int ≈ a ∨ n : int ?
  | Add                         → (+)
  | App (f, x)                        →
        eval (f) (eval x)
```

If the return type of the match is not given, what should it be?

- Use the equation $a = int$ in the branch, but ...
- a or int, equivalent inside the branch,

But type inference difficult...

Matching on a constructor introduces local equations.

These equations are visible in the body of the case

```
let rec eval (type a) (x : a exp)      =
  match x with
  | Int n          → n : int ≈ a ∨ n : int ?   Ambiguous !
  | Add                     → (+)
  | App (f, x)                     →
       eval (f) (eval x)
```

If the return type of the match is not given, what should it be?

- Use the equation $a = int$ in the branch, but ...
- *a* or *int*, equivalent inside the branch,
- become incompatible outside. Returning one or the other are two incompatible solutions. This is called an ambiguity and is rejected.

Easy solution: annotate, everywhere

Our running GADT:

```
type (_,_) eq = Eq : (α,α) eq
```

Give the type of the scrutinee and of the result (making up syntax).

```
let f (type a) x =
  match x : (a, int) eq return a with Eq → 1
```

Easy solution: annotate, everywhere

Our running GADT:

```
type (_,_) eq = Eq : (α,α) eq
```

Give the type of the scrutinee and of the result (making up syntax).

```
let f (type a) x =
  match x : (a, int) eq return a with Eq → 1
```

That is not enough. All free variables must also be annotated:

```
let g (type a) x y  =
  match x : (a, int) eq return a with Eq →
        if y > 0 then y else 1
```

Easy solution: annotate, everywhere

Our running GADT:

```
type (_,_) eq = Eq : (α,α) eq
```

Give the type of the scrutinee and of the result (making up syntax).

```
let f (type a) x =
  match x : (a, int) eq return a with Eq → 1
```

That is not enough. All free variables must also be annotated:

```
let g (type a) x (y : a)  =
  match x : (a, int) eq return a with Eq →
        if y > 0 then y else 1
```

Adding simple type propagation mechanism, we can just write:

```
let f (type a) (x : (a, int) eq) (y : a) : a =
  match x with Eq → if y > 0 then y else 1
```

Advanced solutions: propagate, agressively

Simple *syntactic* propagation is too weak

```
let f (type a) (x : (a, int) eq) : a =
          match x with Eq → 1
```

— OK

Advanced solutions: propagate, agressively

Simple *syntactic* propagation is too weak

```
let f (type a) (x : (a, int) eq) : a =
  let r = match x with Eq → 1 in r
```

— FAILS

Advanced solutions: propagate, agressively

Simple *syntactic* propagation is too weak

Statified type inference (Y. Regis-Gianas and F, Pottier)

Propagate known type information aggressively (iteration process).
Then, proceed as in the explicit version.

Advanced solutions: propagate, agressively

Simple *syntactic* propagation is too weak

Statified type inference (Y. Regis-Gianas and F, Pottier)

Propagate known type information aggressively (iteration process).
Then, proceed as in the explicit version.

OutsideIn (GHC) (T. Schrijvers, SPJ, D. Vytiniotis, M. Sulzmann)

Propagate information flowing from the context into the branch.
But not conversely.

Our solution: rethink ambiguity

We redefine ambiguity as leakage of an ambivalent type.

- An ambivalent is one that allows the use of an equation

  $\texttt{let g (}\textit{type a}\texttt{) (x : }(a, \mathit{int})\ \mathit{eq}\texttt{) (y : }a\texttt{) =}$
  $\quad\texttt{match x with Eq } \langle a = \mathit{int} \rangle \rightarrow$
  $\quad\quad\texttt{... (if true then y else 0 : } a \approx \mathit{int}\texttt{) ...}$

  To type the conditional we must use the equation $a = \mathit{int}$ to convert a into int, so we give the conditional the ambivalent type $a \approx \texttt{int}$.
- Ambivalence is attached to types and propagated to all connected occurences.
- A type annotation fixes a particular type and removes ambivalence.
- An ambivalent type is leaked if it cannot be proved equal under the equations in scope. It is then rejected as ambiguous.

Our solution, on examples

Small variations on the same program:

$$\texttt{let } \mathtt{f}_0 \ (\mathit{type}\ a)\ (\mathtt{x} : (a, \mathit{int})\ \mathit{eq})\ (\mathtt{y} : a) =$$
$$\quad \texttt{match x with Eq}\ \langle a = \mathit{int} \rangle \rightarrow$$
$$\qquad \texttt{true} \quad : \mathit{bool}$$

—without using the equation

In practice

- When no equation is used, there is no ambivalence, nor ambiguities.

Our solution, on examples

Small variations on the same program:

$$\texttt{let } \texttt{f}_1 \ (\mathit{type}\ a)\ (\texttt{x} : (a, \mathit{int})\ \mathit{eq})\ (\texttt{y} : a) =$$
$$\quad \texttt{match x with Eq } \langle a = \mathit{int} \rangle \rightarrow$$
$$\qquad \texttt{1} \quad : \mathit{int}$$

—without using the equation

In practice

- When no equation is used, there is no ambivalence, nor ambiguities.

Our solution, on examples

Small variations on the same program:

$$
\begin{array}{l}
\texttt{let } \texttt{f}_2\ (\mathit{type}\ a)\ (\texttt{x} : (a, \mathit{int})\ \mathit{eq})\ (\texttt{y} : a) = \\
\quad \texttt{match x with Eq}\ \langle a = \mathit{int} \rangle \rightarrow \\
\qquad \texttt{y} > 0 \qquad : \mathit{bool}
\end{array}
$$

—the type of y is $a \approx \mathit{int}$, but not visible in the result

In practice

- When no equation is used, there is no ambivalence, nor ambiguities.
- A type that depends on the use of an equation is ambivalent.
- Only types that leaks out are ambiguous and rejected.

Our solution, on examples

Small variations on the same program:

$\texttt{let f}_2\ (\textit{type a})\ (\texttt{x} : (a, \textit{int})\ \textit{eq})\ (\texttt{y} : a) =$
$\quad \texttt{match x with Eq}\ \langle a = \textit{int} \rangle \rightarrow$
$\qquad \texttt{if y} > 0\ \texttt{then y else}\ 0 \qquad : a \approx \textit{int} \quad \textit{FAILS}$

—the conditional had type $a \approx int$, which leaks in the result

In practice

- When no equation is used, there is no ambivalence, nor ambiguities.
- A type that depends on the use of an equation is ambivalent.
- Only types that leaks out are ambiguous and rejected.

Our solution, on examples

Small variations on the same program:

$$
\begin{array}{l}
\texttt{let f}_2\ \texttt{(}\mathit{type}\ a\texttt{)}\ \texttt{(x : }(a, \mathit{int})\ \mathit{eq}\texttt{)}\ (\texttt{y} : a) \texttt{ =} \\
\quad \texttt{match x with Eq}\ \boxed{\langle a = \mathit{int}\rangle} \rightarrow \\
\qquad (\texttt{if y} > \texttt{0 then y else 0} : \texttt{a}) \qquad \boxed{: a}
\end{array}
$$

—the conditional has type $a \approx int$, which does not leak in the result

In practice

- When no equation is used, there is no ambivalence, nor ambiguities.
- A type that depends on the use of an equation is ambivalent.
- Only types that leaks out are ambiguous and rejected.
- Inner or outer annotations can be used to prevent leakage

Our solution, on examples

Small variations on the same program:

$$
\begin{array}{l}
\texttt{let } \mathtt{f}_2 \ (\mathit{type}\ a)\ (\mathtt{x} : (a, \mathit{int})\ \mathit{eq})\ \ y : a = \\
\quad \texttt{match x with Eq } \langle a = \mathit{int} \rangle \rightarrow \\
\qquad (\texttt{if } (y : a) > 0 \texttt{ then } (y : a) \texttt{ else } 0) \qquad : a
\end{array}
$$

—the conditional has type $a \approx int$, which does not leak in the result

In practice

- When no equation is used, there is no ambivalence, nor ambiguities.
- A type that depends on the use of an equation is ambivalent.
- Only types that leaks out are ambiguous and rejected.
- Inner or outer annotations can be used to prevent leakage

Ambiguity and principality

- Ambiguity is now an intrinsic property of typing derivations (while it was a property of programs).
- Principality is a property of programs.
- Our approach amounts to reject ambiguous derivations.
- The remaining derivations admit a principal one.
- Our type inference builds the most general and least ambivalent derivation, and fails when the only derivations are ambiguous.

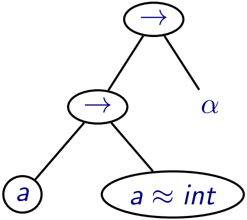
Advantages of refined ambiguity

- Non-ambiguous types don't need annotations.
- Hence, more programs are accepted outright.
- Less pressure for a clever propagation algorithm.
- Particularly useful when there are many local definitions.

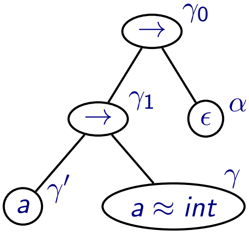
Formalizing ambivalence

- Intuitively, we replace types by sets of equivalent types
- However, we must carefully keep sharing in types so that introducing ambivalence commutes with unification.
- For that, we label every node with a variable, and
- we enforce node descriptions with the same label to be equal.

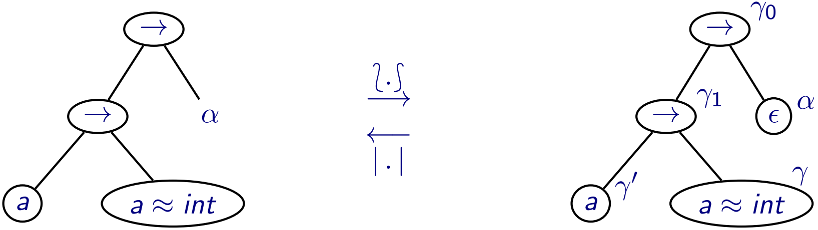
Formalizing ambivalence



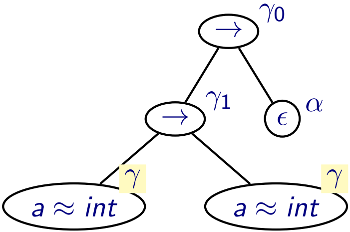
becomes



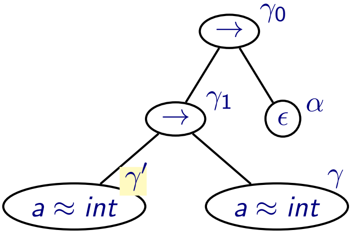
Formalizing ambivalence



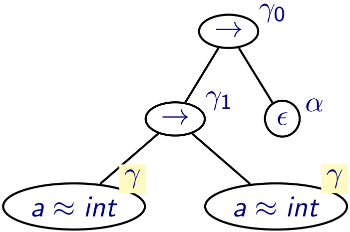
Formalizing ambivalence



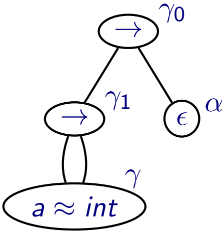
differs from



Formalizing ambivalence

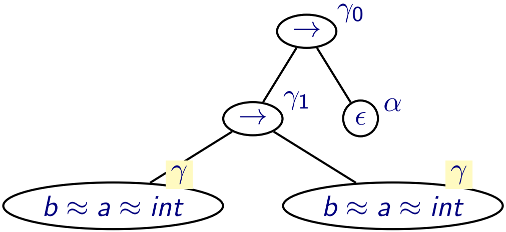


same as

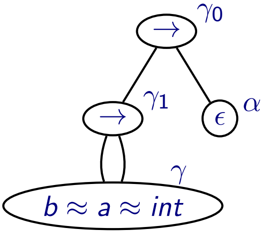


An ambivalent type may still be replaced by a more ambivalent one, e.g. node γ may be replaced by $b \approx a \approx int$

Formalizing ambivalence



or



After subtituting ($b \approx a \approx int$) for γ

Formalizing ambivalence

Fits perfectly with first-order unification

- Solving unification problems may only request equalities of the form $a_1 = \ldots a_n = \tau$ where a_i's are rigid variables.
- Unification *naturally* finds a type with the least ambivalence.
- When exiting a branch, we need only check that the requested ambivalence is implied by the equations remaining in the context.
- (The context can also be organized by decomposing equations into atomic forms $a_1 = \ldots a_n = \tau$, but this is only for efficiency issues.)

ML-style type inference works as usual

Formalization

Types

$$\begin{array}{llll}
\zeta & ::= & \psi^{\alpha} & \text{Types} \\
\rho & ::= & \mathsf{a} \mid \zeta \to \zeta \mid \mathsf{eq}(\zeta, \zeta) \mid \mathtt{int} & \text{Raw types} \\
\psi & ::= & \epsilon \mid \rho \approx \psi & \text{Sets of raw types} \\
\sigma & ::= & \forall(\bar{\alpha})\, \zeta & \text{Type schemes} \\
\tau & ::= & \alpha \mid \tau \to \tau \mid \mathit{int} & \text{Simple types}
\end{array}$$

The erasure of a type ζ is a simple type $|\tau|$ (definition obvious).

Conversely, $\wr\tau\int$ is the type most general type ζ such that τ is ζ.

Typing contexts

As usual + node descriptions $\alpha :: \psi$

$$\Gamma ::= \emptyset \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \tau_1 \doteq \tau_2 \mid \Gamma, \alpha :: \psi$$

Well-formedness

Ensures that at most one of element of ψ is not a rigid variable.

Ensures coherence: $\Gamma \vdash \psi^\alpha$ only if $\alpha :: \psi \in \Gamma$.

Finally, equalities in ψ should follow from equations in Γ.

Typing judgments (Example)

$$\alpha :: \mathtt{int} \vdash \lambda(x)\, x : \forall(\gamma)\, (\mathtt{int}^\alpha \to \mathtt{int}^\alpha)^\gamma$$

Substitution

Substitution discards the original contents of a node.

$$[\zeta/\alpha]\psi^\alpha = \zeta \qquad [\zeta/\alpha](\zeta_1 \to \zeta_2)^\gamma = ([\zeta/\alpha]\zeta_1 \to [\zeta/\alpha]\zeta_2)^\gamma$$

For example, $[\psi^\alpha/\alpha]\zeta$ is a type in which all nodes labelled α are ψ.

A substitution θ preserves ambivalence in a type ζ if and only if, for any $\alpha \in \mathsf{dom}(\theta)$ and any node ψ^α inside ζ, we have

$$\theta(\psi) \subseteq \psi_1 \quad \text{where} \quad {\psi_1}^\alpha = \theta(\psi^\alpha)$$

Typing rules (enforcing sharing) ▶▶

GEN
$$\frac{\Gamma, \alpha :: \psi \vdash M : \sigma}{\Gamma \vdash M : \forall(\alpha)\ \sigma}$$

INST
$$\frac{\Gamma \vdash M : \forall(\alpha)\ [\psi_0^\alpha/\alpha]\sigma \qquad \psi_0 \subseteq \psi \qquad \Gamma \vdash \psi^\gamma}{\Gamma \vdash M : [\psi^\gamma/\alpha]\sigma}$$

VAR
$$\frac{\vdash \Gamma \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

NEW
$$\frac{\Gamma, a, \alpha :: a \vdash M : \sigma \qquad \Gamma \vdash \forall(\alpha)\ [\epsilon^\alpha/\alpha]\sigma}{\Gamma \vdash \nu(a)M : \forall(\alpha)\ [\epsilon^\alpha/\alpha]\sigma}$$

FUN
$$\frac{\Gamma, x : \zeta_0 \vdash M : \zeta}{\Gamma \vdash \lambda(x)\ M : \forall(\gamma)\ (\zeta_0 \to \zeta)^\gamma}$$

APP
$$\frac{\Gamma \vdash M_1 : ((\zeta_2 \to \zeta) \approx \psi)^\alpha \quad \Gamma \vdash M_2 : \zeta_2}{\Gamma \vdash M_1\ M_2 : \zeta}$$

LET
$$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \zeta_2}{\Gamma \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \zeta_2}$$

ANN
$$\frac{\Gamma \vdash \forall(\mathsf{ftv}(\tau))\ \tau}{\Gamma \vdash (\tau) : \forall(\mathsf{ftv}(\tau))\ \wr\tau \to \tau\int}$$

EQ
$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Eq} : \forall(\alpha, \gamma)\ \mathsf{eq}(\alpha, \alpha)^\gamma}$$

MATCH
$$\frac{\Gamma \vdash (\mathsf{eq}(\tau_1, \tau_2))\ M_1 : \zeta_1 \qquad \Gamma, \tau_1 \doteq \tau_2 \vdash M_2 : \zeta_2}{\Gamma \vdash \mathsf{match}\ M_1 : \mathsf{eq}(\tau_1, \tau_2)\ \mathsf{with}\ \mathsf{Eq}\ \to M_2 : \zeta_2}$$

Typing rules (enforcing sharing)

Inst

$$\frac{\Gamma \vdash M : \forall(\alpha)\ [\psi_0^\alpha/\alpha]\sigma \qquad \psi_0 \subseteq \psi \qquad \Gamma \vdash \psi^\gamma}{\Gamma \vdash M : [\psi^\gamma/\alpha]\sigma}$$

Var

$$\frac{\vdash \Gamma \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

New

$$\frac{\Gamma, a, \alpha :: a \vdash M : \sigma \qquad \Gamma \vdash \forall(\alpha)\ [\epsilon^\alpha/\alpha]\sigma}{\Gamma \vdash \nu(a) M : \forall(\alpha)\ [\epsilon^\alpha/\alpha]\sigma}$$

Fun

$$\frac{\Gamma, x : \zeta_0 \vdash M : \zeta}{\Gamma \vdash \lambda(x)\ M : \forall(\gamma)\ (\zeta_0 \to \zeta)^\gamma}$$

App

$$\frac{\Gamma \vdash M_1 : ((\zeta_2 \to \zeta) \approx \psi)^\alpha \quad \Gamma \vdash M_2 : \zeta_2}{\Gamma \vdash M_1\ M_2 : \zeta}$$

Let

$$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \zeta_2}{\Gamma \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \zeta_2}$$

Ann

$$\frac{\Gamma \vdash \forall(\mathsf{ftv}(\tau))\ \tau}{\Gamma \vdash (\tau) : \forall(\mathsf{ftv}(\tau))\ \wr \tau \to \tau \int}$$

Eq

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Eq} : \forall(\alpha, \gamma)\ \mathsf{eq}(\alpha, \alpha)^\gamma}$$

Match

$$\frac{\Gamma \vdash (\mathsf{eq}(\tau_1, \tau_2))\ M_1 : \zeta_1 \qquad \Gamma, \tau_1 \doteq \tau_2 \vdash M_2 : \zeta_2}{\Gamma \vdash \mathsf{match}\ M_1 : \mathsf{eq}(\tau_1, \tau_2)\ \mathsf{with}\ \mathsf{Eq}\ \to M_2 : \zeta_2}$$

Typing rules (enforcing sharing) ▶▶

Gen

$$\Gamma, \alpha :: \psi \vdash M : \sigma$$

Inst

$$\Gamma \vdash M : \forall(\alpha)\ [\psi_0^\alpha/\alpha]\sigma \qquad \psi_0 \subseteq \psi \qquad \Gamma \vdash \psi^\gamma$$

New

$$\frac{\Gamma,\ a, \alpha :: a \vdash M : \sigma \qquad \Gamma \vdash \forall(\alpha)\ [\epsilon^\alpha/\alpha]\sigma}{\Gamma \vdash \nu(a)M : \forall(\alpha)\ [\epsilon^\alpha/\alpha]\sigma}$$

Fun

$$\frac{\Gamma, x : \zeta_0 \vdash M : \zeta}{\Gamma \vdash \lambda(x)\ M : \forall(\gamma)\ (\zeta_0 \to \zeta)^\gamma}$$

App

$$\frac{\Gamma \vdash M_1 : ((\zeta_2 \to \zeta) \approx \psi)^\alpha \quad \Gamma \vdash M_2 : \zeta_2}{\Gamma \vdash M_1\ M_2 : \zeta}$$

Let

$$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \zeta_2}{\Gamma \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \zeta_2}$$

Ann

$$\frac{\Gamma \vdash \forall(\mathsf{ftv}(\tau))\ \tau}{\Gamma \vdash (\tau) : \forall(\mathsf{ftv}(\tau))\ \wr\tau \to \tau\int}$$

Eq

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Eq} : \forall(\alpha, \gamma)\ \mathsf{eq}(\alpha, \alpha)^\gamma}$$

Match

$$\frac{\Gamma \vdash (\mathsf{eq}(\tau_1, \tau_2))\ M_1 : \zeta_1 \qquad \Gamma, \tau_1 \doteq \tau_2 \vdash M_2 : \zeta_2}{\Gamma \vdash \mathsf{match}\ M_1 : \mathsf{eq}(\tau_1, \tau_2)\ \mathsf{with}\ \mathsf{Eq}\ \to M_2 : \zeta_2}$$

Typing rules (enforcing sharing)

Gen

$$\frac{\Gamma, \alpha :: \psi \vdash M : \sigma}{\Gamma \vdash M : \forall(\alpha)\ \sigma}$$

Inst

$$\frac{\Gamma \vdash M : \forall(\alpha)\ [\psi_0^\alpha / \alpha]\sigma \qquad \psi_0 \subseteq \psi \qquad \Gamma \vdash \psi^\gamma}{\Gamma \vdash M : [\psi^\gamma / \alpha]\sigma}$$

Var

$$\vdash \Gamma \qquad x : \sigma \in \Gamma$$

Γ ...

New

$$\Gamma, a, \alpha :: a \vdash M : \sigma \qquad \Gamma \vdash \forall(\alpha)\ [\epsilon^\alpha / \alpha]\sigma$$

App

$$\frac{\Gamma \vdash M_1 : ((\zeta_2 \to \zeta) \approx \psi)^\alpha \quad \Gamma \vdash M_2 : \zeta_2}{\Gamma \vdash M_1\ M_2 : \zeta}$$

Fun

Γ, x ...

$\Gamma \vdash \lambda(x)$ ...

Let

$$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \zeta_2}{\Gamma \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \zeta_2}$$

Ann

$$\frac{\Gamma \vdash \forall(\mathsf{ftv}(\tau))\ \tau}{\Gamma \vdash (\tau) : \forall(\mathsf{ftv}(\tau))\ \wr \tau \to \tau \wr}$$

Eq

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Eq} : \forall(\alpha, \gamma)\ \mathsf{eq}(\alpha, \alpha)^\gamma}$$

Match

$$\frac{\Gamma \vdash (\mathsf{eq}(\tau_1, \tau_2))\ M_1 : \zeta_1 \qquad \Gamma, \tau_1 \doteq \tau_2 \vdash M_2 : \zeta_2}{\Gamma \vdash \mathsf{match}\ M_1 : \mathsf{eq}(\tau_1, \tau_2)\ \mathsf{with}\ \mathsf{Eq} \ \to M_2 : \zeta_2}$$

Typing rules (enforcing sharing) ▶▶

Gen
$$\frac{\Gamma, \alpha :: \psi \vdash M : \sigma}{\Gamma \vdash M : \forall(\alpha)\ \sigma}$$

Inst
$$\frac{\Gamma \vdash M : \forall(\alpha)\ [\psi_0^\alpha/\alpha]\sigma \qquad \psi_0 \subseteq \psi \qquad \Gamma \vdash \psi^\gamma}{\Gamma \vdash M : [\psi^\gamma/\alpha]\sigma}$$

Var
$$\frac{\vdash \Gamma \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

New
$$\frac{\Gamma, a, \alpha :: a \vdash M : \sigma \qquad \Gamma \vdash \forall(\alpha)\ [\epsilon^\alpha/\alpha]\sigma}{\Gamma \vdash \nu(a) M : \forall(\alpha)\ [\epsilon^\alpha/\alpha]\sigma}$$

Fun
$$\frac{\Gamma, x : \zeta_0 \vdash M : \zeta}{\Gamma \vdash \lambda(x)\, M : \forall(\gamma)\ (\zeta_0 \to \ldots}$$

App
$$\frac{\Gamma \vdash M_1 : ((\zeta_2 \to \zeta) \approx \psi)^\alpha \quad \Gamma \vdash M_2 : \zeta_2}{\ldots}$$

Let
$$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma \ldots}{\Gamma \vdash \text{let } x = M_1 \text{ in } \ldots}$$

Ann
$$\frac{\Gamma \vdash \forall(\mathrm{ftv}(\tau))\ \tau}{\Gamma \vdash (\tau) : \forall(\mathrm{ftv}(\tau))\ \wr \tau \to \tau \int}$$

Eq
$$\frac{\vdash \Gamma}{\Gamma \vdash \mathrm{Eq} : \forall(\alpha, \gamma)\ \mathrm{eq}(\alpha, \alpha)^\gamma}$$

Match
$$\frac{\Gamma \vdash (\mathrm{eq}(\tau_1, \tau_2))\, M_1 : \zeta_1 \qquad \Gamma, \tau_1 \doteq \tau_2 \vdash M_2 : \zeta_2}{\Gamma \vdash \text{match } M_1 : \mathrm{eq}(\tau_1, \tau_2) \text{ with Eq } \to M_2 : \zeta_2}$$

Typing rules (enforcing sharing)

GEN
$$\frac{\Gamma, \alpha :: \psi \vdash M : \sigma}{\Gamma \vdash M : \forall(\alpha)\ \sigma}$$

INST
$$\frac{\Gamma \vdash M : \forall(\alpha)\ [\psi_0^\alpha/\alpha]\sigma \qquad \psi_0 \subseteq \psi \qquad \Gamma \vdash \psi^\gamma}{\Gamma \vdash M : [\psi^\gamma/\alpha]\sigma}$$

VAR
$$\frac{\vdash \Gamma \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

NEW
$$\frac{\Gamma, a, \alpha :: a \vdash M : \sigma \qquad \Gamma \vdash \forall(\alpha)\ [\epsilon^\alpha/\alpha]\sigma}{\Gamma \vdash \nu(a) M : \forall(\alpha)\ [\epsilon^\alpha/\alpha]\sigma}$$

FUN
$$\frac{\Gamma, x : \zeta_0 \vdash M : \zeta}{\Gamma \vdash \lambda(x)\ M : \forall(\gamma)\ (\zeta_0 \to \zeta)^\gamma}$$

APP
$$\frac{\Gamma \vdash M_1 : ((\zeta_2 \to \zeta) \approx \psi)^\alpha \quad \Gamma \vdash M_2 : \zeta_2}{\Gamma \vdash M_1\ M_2 : \zeta}$$

LET
$$\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \zeta_2$$

ANN
$$\Gamma \vdash \forall(\mathrm{ftv}(\tau))\ \tau$$

MATCH
$$\frac{\Gamma \vdash (\mathrm{eq}(\tau_1, \tau_2))\ M_1 : \zeta_1 \qquad \Gamma, \tau_1 \doteq \tau_2 \vdash M_2 : \zeta_2}{\Gamma \vdash \mathrm{match}\ M_1 : \mathrm{eq}(\tau_1, \tau_2)\ \mathrm{with}\ \mathrm{Eq}\ \to M_2 : \zeta_2}$$

Principal solutions to typing problems

Addapting the setting to the framework

- Because of sharing, one cannot blindly substitute typing judgments.
- To preserve well-formedness, a subtitution θ must also register new node descriptions in a typing context Δ, which must be inserted at proper places in Γ.

Formally

- A typing problem is a skeleton $\Gamma \triangleright M : \zeta$ where $\Gamma \vdash M : \zeta$ may not hold
- A solution is a pair (Δ, θ) such that $\theta(\Gamma) \mid \Delta \vdash M : \theta(\zeta)$ holds where $\theta(\Gamma) \mid \Delta$ inserts Δ at proper positions in $\theta(\Gamma)$.

Typing problems have principal solutions

Theorem

Any solvable typing problem has a most general solution.

No cheating

Having principal solutions is not wired into the typing rules.

This contrasts with OutsideIn (or PolyML) where:

- some typing problems that do not have principal solutions are detected and rejected. . . (because some typing rules say so.)
- so that typing problems that have a solution have a principal one.

Robustness

- This is not to blame OutsideIn or PolyML
  but to emphasize the robustness of our approach...
- Type inference is just based on first-order unification, as in ML.

Monotonicity of typings

Setting

Let $\Gamma \vdash \sigma' \prec \sigma$ be the instantiation relation: *i.e.* any monomorphic instance of σ well-formed in Γ is also a monomorphic instance of σ'.

We extend this relation point-wise to typing contexts: $\Gamma' \prec \Gamma$.

Typing judgments are monotonic

Strengthening the type of a free variable preserves well-typedness:

$$\text{if } \Gamma \vdash M : \zeta \text{ and } \vdash \Gamma' \prec \Gamma, \text{ then } \Gamma' \vdash M : \zeta$$

Monotonicity holds in ML but not in OutsideIn

- This property is used in the proof of principality.
- This is interesting because it increases modularity and predictability.

(Using inferred types as annotations to restrict types breaks monotonicity.)

Comparison with GHC

GHC uses OutsideIn which is a powerful constraint-based type inference algorithm where type information cannot leak out of GADT branches.

Comparison in the large is difficult

- GHC 7 implemented a relaxed version of OutsideIn untill recently (or still does...).
  Will users be happy with the more restrictive version?
- OCaml has some form of propagation, close to syntactic propagation, but using local polymorphism.
- OutsideIn is essentially a constraint propagation strategy, which is largely orthogonal to tracing ambivalence.

Comparison with OutsideIn

OCaml may fail while GHC succeeds

```
let f (type a) (x : (a, int) eq) : a =
  let r = match x with Eq → 1 in r
```

Insufficient propagation.

GHC fails while Ocaml succeeds

```
let f (type a) (x : (a, int) eq) : unit =
  let z = match x with Eq → 1 in ()
```

No outside constraint on `z`,
which is ambiguous in GHC, but not in OCaml as it is not ambivalent

Comparison with OutsideIn (More)

Constraint propagation of OutsideIn is strong

So that sometimes no annotation at all is needed:

```
type a t = R1 : int t | R2 : a → a t
function x → match x with R1 → 1 | R2 x → x
(* - : R t → t *)
```

local let bindings are not *implicitly* generalized

To allow upward propagation,

```
let id x = x in (id "a", id True)
(* -- Fails *)
```

Sometimes forcing λ-lifting and moving local definitions further from their use, which is not great for program maintainance.

(I.e. there is a real cost to monomorphic let.)

Comparison with OutsideIn

System	Ambivalence	OutsideIn
Inference	unification-based	constraint-based
Principality	$\surd$	$\surd$(†)
Monotonicity	$\surd$	$-$
Polymorphic `let`	$\surd$	$-$

(†) Only accepts derivations that are principal.

Let-bindings should be generalized!

In OCaml!

- Jacques Garrigue conducted the experiment in OCaml
- Similar number of files to be changed
- Changes might be harder in OCaml
- Types tend to be larger and harder to infer mentally.
  More uses of structural types, perhaps due to the use of objects and variant types.

OCaml also relies on local polymorphism for

- First-class polymorphism
- Object types
- Propagation of type annotations that complements ambivalent types.

Combining ambivalence and OustsideIn

Interest

Both could help one another to have simultaneously

- fewer ambiguities
- more aggressive propagation

Feasability

The two approaches are mostly orthogonal

- In a final phase GHC checks that constraints do not leak out from branches.
- One could restrict this check to ambivalent types.
- Requires some instrumentation of the type structure to track ambivalent types.

Other applications of this idea?

Should work for GADTs type inference in MLF...

Beyong type inference for GADTs?

I don't know.