Ambivalent Types for Principal Type Inference with GADTs

APLAS 2013, Melbourne

Jacques Garrigue & Didier Rémy
Nagoya University / INRIA
Generalized Algebraic Datatypes

- Algebraic datatypes allowing **different type parameters** for different cases.
- Similar to inductive types of Coq et al.

\[
\text{type } _\text{expr} = \\
| \text{Int} : \text{int} \to \text{int} \, \text{expr} \\
| \text{Add} : (\text{int} \to \text{int} \to \text{int}) \, \text{expr} \\
| \text{App} : (\text{'a} \to \text{'b}) \, \text{expr} \ast \text{'a} \, \text{expr} \to \text{'b} \, \text{expr}
\]

\[
\text{App} (\text{Add}, \text{Int} \, 3) : (\text{int} \to \text{int}) \, \text{expr}
\]

- Able to express **invariants** and **proofs**
- Also provide **existential types**: \( \exists \text{'a}.(\text{'a} \to \text{'b}) \, \text{expr} \ast \text{'a} \, \text{expr} \)
- Available in Haskell since 2005, and in OCaml since 2012. This paper describes OCaml’s approach.
GADTs and pattern-matching

- Matching on a constructor introduces local equations.
- These equations can be used in the body of the case.
- The parameter must be a rigid type variable.
- Existentials introduce fresh rigid type variables.

```ocaml
let rec eval : type a. a expr -> a = function
  | Int n -> n (* a = int *)
  | Add -> (+) (* a = int -> int -> int *)
  | App (f, x) -> eval f (eval x) (* polymorphic recursion *)
                    (* \exists b, f : b -> a \land x : b *)

val eval : 'a expr -> 'a = <fun>

eval (App (App (Add, Int 3), Int 4));;
- : int = 7
```
Type inference

- Providing sound type inference for GADTs is not difficult.
- However, principal type inference for the unrestricted type system is not possible.

We consider a simple setting where the only GADT is `eq`.

```
type (_,_) eq = Eq : ('a,'a) eq (* equality witness *)

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

- What should be the return type?
- Both `int` and `a` are valid choices, and they are not compatible.
- Such a situation is called ambiguous.
Known solution : explicit types

A simple solution is to require that all GADT pattern-matchings be annotated with rigid type annotations (containing only rigid type variables).

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

If we allow some propagation of annotations this doesn’t sound too painful:

```ml
let f : type a. (a,int) eq -> int
    = fun Eq -> 1
```
Weaknesses of explicit types

- Annotating the matching alone is not sufficient:

  ```ml
define g : type a. (a,int) eq -> int =
  let g (type a) x y =
    match (x : (a,int) eq) return int with
    Eq -> if y > 0 then y else 0
  end

  Here the type of `y` is ambiguous too.
  Not only the input and result of pattern-matching must be annotated, but also all free variables.

- Propagation does not always work, but if we try to use known function types as explicit types too, we lose monotonicity:

  ```ml
define f : type a. (a,int) eq -> int =
  let f : type a. (a,int) eq -> int =
    fun x -> succ (match x with Eq -> 1)
  end

  If we replace the type of `succ` by `'a -> int`, which is more general than `int -> int`, this is no longer typable.
Rethinking ambiguity

Compare these two programs:

```ocaml
let f (type a) (x : (a,int) eq) =  
    match x with Eq -> 1 (* a = int *)

let f' (type a) (x : (a,int) eq) =  
    match x with Eq -> true (* a = int *)
```

According to the standard definition of ambiguity, \( f \) is ambiguous, but \( f' \) is not, since there is no equation involving \( \text{bool} \).

This seems strange, as they are very similar.

Is there another definition of ambiguity, which would allow choosing \( f : 'a t \to \text{int} \) over \( f : 'a t \to 'a \)?
Another definition of ambiguity

We redefine ambiguity as leakage of an ambivalent type.

- There is ambivalence if we need to use an equation inside the typing derivation.

  \[
  \text{let } g \text{ (type } a) \text{ (x : (a,int) eq) (y : a) =}
  
  \text{match } x \text{ with Eq } \rightarrow \text{ if } \text{true} \text{ then } y \text{ else } 0
  \]

  The typing rule for \textit{if} mixes \texttt{a} and \texttt{int} into an \texttt{ambivalent type}.

- Ambivalence is propagated to all connected occurrences.

- Type annotations stop its propagation.

- An ambivalent type is leaked if it occurs outside the scope of its equation. It becomes ambiguous. Here, the typing rule for \texttt{match} leaks the result of \texttt{if} outside of the scope of \texttt{a = int}.
Using refined ambiguity

- Still need to annotate the scrutinee, but if we can type a case without using the equation, there is no ambivalence.

```ml
let f (type a) (x : (a,int) eq) =
  match x with Eq -> 1
val f : ('a,int) eq -> int
```

-Leaks can be fixed by local annotations.

```ml
let g (type a) (x : (a,int) eq) (y : a) =
  match x with Eq -> if true then y else (0 : a)
val g : ('a,int) eq -> 'a -> 'a
```

Advantages

- More programs are accepted outright.
- Less pressure for a non-monotonous propagation algorithm.
- Particularly useful if matching appears nested.
Formalizing ambivalence

- The basic idea is simple: replace types by sets of types.

- Formalization is easy for monotypes alone.
  - We just use the same rules for most cases.
  - We can still use a substitutive Let rule for polymorphism.

- Polymorphic types are more difficult.
  - We must track sharing inside them.
  - Needed for polymorphic recursion, etc...
**Set-based formalization** (not in paper)

\[
\begin{align*}
\tau & ::= \ a & \text{rigid variable} \\
& | \ eq(\tau, \tau) & \text{equality witness} \\
& | \ \tau \to \tau & \text{other types} \\
\zeta & ::= \ \text{set of types } \tau \\
P & ::= \ \text{set of rigid variables } a \\
\Gamma & ::= \ \emptyset \ | \ \Gamma, x : \zeta \ | \ \Gamma, a \ | \ \Gamma, \tau \equiv \tau \ \text{contexts}
\end{align*}
\]

For \( \zeta \) to be well-formed under a context \( \Gamma \),

- It must be structurally decomposable:
  \[
  \zeta = P \ | \ \zeta = \{\text{int}\} \cup P \ | \ \zeta = \zeta_1 \to \zeta_2 \cup P \ | \ \zeta = \text{eq}(\zeta_1, \zeta_2) \cup P
  \]
  where \( \zeta_1 \to \zeta_2 = \{\tau_1 \to \tau_2 \mid \tau_1 \in \zeta_1, \tau_2 \in \zeta_2\} \) and \( \text{eq}(\zeta_1, \zeta_2) = \ldots \)

- Its types must be compatible with each other under \( \Gamma \).
  I.e., for any ground instance \( \theta \) of the rigid variables of \( \Gamma \) satisfying its equations, \( \theta(\tau_1) = \theta(\tau_2) \).
Set-based rules

\[
\begin{align*}
\text{Var} & \quad \frac{x : \zeta \in \Gamma}{\Gamma \vdash x : \zeta} \\
\text{App} & \quad \frac{\Gamma \vdash M_1 : \zeta_2 \to \zeta_1 \cup P \quad \Gamma \vdash M_2 : \zeta_2}{\Gamma \vdash M_1 \ M_2 : \zeta_1} \\
\text{Let} & \quad \frac{\Gamma \vdash M_1 : \zeta_1 \quad \Gamma \vdash [M_1/x]M_2 : \zeta}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \zeta} \\
\text{Fun} & \quad \frac{\Gamma, x : \zeta_0 \vdash M_1 : \zeta_1}{\Gamma \vdash \text{fun } x \to M_1 : \zeta_0 \to \zeta_1 \cup P} \\
\text{Ann} & \quad \frac{\Gamma \vdash M : \zeta_1 \quad \tau \in \zeta_1 \cap \zeta_2}{\Gamma \vdash (M : \tau) : \zeta_2} \\
\text{Use} & \quad \frac{\Gamma \vdash M_1 : \{\text{eq}(\tau_1, \tau_2)\} \cup \zeta_1 \quad \Gamma, \tau_1 \equiv \tau_2 \vdash M_2 : \zeta_2}{\Gamma \vdash \text{use } M_1 : \text{eq}(\tau_1, \tau_2) \text{ in } M_2 : \zeta_2}
\end{align*}
\]

All types must be well-formed in their context.
Polymorphism and type inference

- Move to a graph-based approach, to track sharing.

- **Nodes are sets** which may contain a normal type and some rigid variables.

- **Polymorphic types are graphs**, where each node may be polymorphic (*i.e.* allow the addition of rigid variables).
The following specification of ambivalent types should be understood as representing DAGs.

\[ \rho ::= a \mid \zeta \rightarrow \zeta \mid \text{eq}(\zeta, \zeta) \mid \text{int} \]
\[ \psi ::= \epsilon \mid \rho \approx \psi \quad \zeta ::= \psi^\alpha \quad \sigma ::= \forall(\bar{\alpha}) \zeta \]

True variables are empty nodes: \( \epsilon^\alpha \)

Typing contexts contain node descriptions:

\[ \Gamma ::= \emptyset \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \tau_1 \rightarrow \tau_2 \mid \Gamma, \alpha :: \psi \]

Well-formedness ensures coherence: \( \Gamma \vdash \psi^\alpha \) only if \( \alpha :: \psi \in \Gamma \)

Example of type judgment:

\[ a \doteq \text{int}, \alpha :: a \approx \text{int} \vdash \lambda(x)x : \forall(\gamma) (a \approx \text{int}^\alpha \rightarrow a \approx \text{int}^\alpha)^\gamma \]
Substitution

Substitution discards the original contents of a node.

\[
\begin{align*}
[\zeta/\alpha] \psi^\alpha &= \zeta \\
[\zeta/\alpha](\zeta_1 \to \zeta_2)^\gamma &= ([\zeta/\alpha] \zeta_1 \to [\zeta/\alpha] \zeta_2)^\gamma
\end{align*}
\]

A substitution \( \theta \) preserves ambivalence in a type \( \zeta \) if and only if, for any \( \alpha \in \text{dom}(\theta) \) and any node \( \psi^\alpha \) inside \( \zeta \), we have

\[
\theta(\psi) \subseteq [\theta(\psi^\alpha)]
\]

where for any \( \psi^\alpha \), \( [\psi^\alpha] = \psi \). I.e. substitution preserves the structure of types, possibly adding new elements to nodes.

This is similar to structural polymorphism (polymorphic variants).
Graph-based rules

\[
\begin{align*}
\text{Inst} & \quad \Gamma \vdash M : \forall(\alpha) \left[ \psi_0^{\alpha/\alpha} \right] \sigma \\ & \quad \psi_0 \subseteq \psi \\ & \quad \Gamma \vdash \psi^\gamma \\ & \quad \Gamma \vdash M : [\psi^\gamma/\alpha] \sigma \\
\text{Var} & \quad \vdash \Gamma \\ & \quad x : \sigma \in \Gamma \\ & \quad \Gamma \vdash x : \sigma \\
\text{App} & \quad \Gamma \vdash M_1 : ((\zeta_2 \rightarrow \zeta) \approx \psi)^\alpha \\ & \quad \Gamma \vdash M_2 : \zeta_2 \\ & \quad \Gamma \vdash M_1 \ M_2 : \zeta \\
\text{Let} & \quad \Gamma \vdash M_1 : \sigma_1 \\ & \quad \Gamma, x : \sigma_1 \vdash M_2 : \zeta_2 \\ & \quad \Gamma \vdash \text{let} \ x = M_1 \ \text{in} \ M_2 : \zeta_2 \\
\text{Fun} & \quad \Gamma, x : \zeta_0 \vdash M : \zeta \\ & \quad \Gamma \vdash \lambda(x) \ M : \forall(\gamma) \ (\zeta_0 \rightarrow \zeta)^\gamma \\
\text{Ann} & \quad \Gamma \vdash \forall(\text{ftv}(\tau)) \ \tau \\ & \quad \Gamma \vdash (\tau) : \forall(\text{ftv}(\tau)) \ \llbracket \tau \rightarrow \tau \rrbracket \\
\text{Use} & \quad \Gamma \vdash (\text{eq}(\tau_1, \tau_2)) \ M_1 : \zeta_1 \\ & \quad \Gamma, \tau_1 \triangledown \tau_2 \vdash M_2 : \zeta_2 \\ & \quad \Gamma \vdash \text{use} \ M_1 : \text{eq}(\tau_1, \tau_2) \ \text{in} \ M_2 : \zeta_2
\end{align*}
\]
Ambiguity and principality

- **Ambiguity** is a decidable property of typing derivations.

- **Principality** is a property of programs, not directly verifiable.

- Our approach is to reject ambiguous derivations.

- The remaining derivations admit a principal one.

- Our type inference builds the most general and least ambivalent derivation, and fails if it becomes ambiguous.

- By construction, our approach preserves monotonicity.
Comparison with OutsideIn

OutsideIn is a powerful constraint-based type inference algorithm where information is not allowed to leak from GADT cases.

Comparison is difficult:

- GHC 7, up to 7.6.x implements a buggy version of OutsideIn, which accepts some non-principal examples. The bug is fixed in the development version.

- OutsideIn is essentially a constraint propagation strategy, which is somehow orthogonal to ambiguity detection.

- OCaml has some form of propagation, which relies on polymorphism, and is close to syntactic propagation.

- We compare OCaml 4.00 to the development version of GHC 7.
Comparison examples

- OCaml fails (while GHC 7 succeeds)
  
  ```ocaml
  let f : type a. (a, int) eq -> a = fun x ->
  let r = match x with Eq -> 1 in r
  Error: This expression has type int but expected a
  Insufficient propagation.
  ```

- GHC fails (while OCaml succeeds)
  
  ```haskell
  data Eqq a b where EQQ :: Eqq a a
  f :: Eqq a Int -> ()
  f x =
    let z = case x of {EQQ -> True} in ()
  Couldn’t match expected type ‘t0’ with actual type ‘Bool’
  ‘t0’ is untouchable inside the constraints (a ~ Int)
  No external constraint on z.
  ```
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>OCaml</th>
<th>GHC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GADTs</strong></td>
<td>since 2012</td>
<td>since 2005</td>
</tr>
<tr>
<td><strong>Type discipline</strong></td>
<td>ambiguity det.</td>
<td>OutsideIn</td>
</tr>
<tr>
<td><strong>Polymorphic let</strong></td>
<td>√</td>
<td>–</td>
</tr>
<tr>
<td><strong>Inference</strong></td>
<td>unification-based</td>
<td>constraint-based</td>
</tr>
<tr>
<td><strong>Principality</strong></td>
<td>√</td>
<td>(1)</td>
</tr>
<tr>
<td><strong>Monotonicity</strong></td>
<td>√</td>
<td>–</td>
</tr>
<tr>
<td><strong>Exhaustiveness check</strong></td>
<td>√</td>
<td>–</td>
</tr>
<tr>
<td><strong>Type-level functions</strong></td>
<td>–</td>
<td>√</td>
</tr>
</tbody>
</table>

(1) There is no principal type system, but OutsideIn only accepts derivations that are principal in the unrestricted type system.
In the paper

- Full formalization of the polymorphic version of ambiguity detection, using the graph-based approach.

- The inference algorithm and its principality proof are available in the accompanying technical report.

http://gallium.inria.fr/~remy/gadts/