Tracing Ambiguity in GADT Type Inference

ML Workshop 2012, Copenhagen

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.

```ocaml
type _ expr =
  | Int : int -> int expr
  | Add : (int -> int -> int) expr
  | App : ('a -> 'b) expr * 'a expr -> 'b expr

App (Add, Int 3) : (int -> int) expr
```

- Able to express invariants and proofs
- Also provide existential types: \( \exists 'a. ('a -> 'b) expr * 'a expr \)
- Now available in OCaml 4.00
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 *)
  (* ∃b, f : b -> a ∧ x : b *)

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

eval (App (App (Add, Int 3), Int 4));;
- : int = 7
```
Rigid type variables and recursion

OCaml has two ways of requesting polymorphism:

- Use **locally abstract types** that behave as rigid type variables.
- Use **universal type variables**, for polymorphic recursion.

The syntax `type a. a expr -> a` combines them:

```ocaml
let rec eval : type a. a expr -> a = ...
```

is syntactic sugar for

```ocaml
let rec eval : 'a. 'a expr -> 'a =
  fun (type a) -> (... : a expr -> a)
```

In this talk we do not deal with recursion, so we will not use this syntactic sugar much.
This talk

- Difficulty of GADT type inference
- Traditional approach using explicit types
- Our approach: refined ambiguity detection
- How it compares with GHC’s OutsideIn
GADTs and type inference

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

```haskell
let f (type a) (x : a t) =
  match x with Int -> 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).

\[
\text{let } f \ (\text{type } a) \ x = \\
\quad \text{match } (x : a \ t) \text{ return } a \text{ with } \text{Int } \rightarrow 1
\]

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

\[
\text{let } f : \text{type } a. \ a \ t \rightarrow a \text{ = function Int } \rightarrow 1
\]
Weaknesses of explicit types

– Is it really sufficient?

   let g (type a) x y =  
     match (x : a t) return a with  
       Int -> if y > 0 then y else 0  

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.

– Simple syntactic propagation is too weak

   let f : type a. a t -> a = fun x ->  
     let r = match x with Int -> 1  
     in r  

If we want to propagate backward the type of \( r \), we need a  
stronger approach, like GHC’s OutsideIn.
Rethinking ambiguity

Compare these two programs:

```ocaml
let f (type a) (x : a t) =  (* a = int *)
  match x with Int -> 1

let f' (type a) (x : a t) =  (* a = int *)
  match x with Int -> true
```

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, such that \( f : 'a t \rightarrow \text{int} \) would not be rejected, but \( f : 'a t \rightarrow 'a \) would?
Another definition of ambiguity

We redefine ambiguity as \textit{leakage of an ambivalent type}.

- A type is \textit{ambivalent} if we need to use an equation inside the typing derivation.

\begin{verbatim}
let g (type a) (x : a t) (y : a) =
  match x with Int -> if true then y else 0
\end{verbatim}

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

- Ambivalence is \textit{propagated} to all connected occurrences.

- Type annotations \textit{stop its propagation}.

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

- If we can type a case without using the equation, there is no ambivalence, so there is no ambiguity.

```ocaml
let f (type a) (x : a t) = match x with Int -> 1
val f : 'a t -> int
```

- Leaks can be fixed by inner or outer annotations.

```ocaml
let g (type a) (x : a t) y =
  match x with Int -> if true then y else (0 : a)
val g : 'a t -> 'a -> 'a
```

- If a variable is used with ambivalent types, we can annotate its binding occurrence to prevent leaks.

```ocaml
let g (type a) (x : a t) (y : a) =
  match x with Int -> if y > 0 then y else (0 : a)
val g : 'a t -> 'a -> 'a
```
Ambiguity and principality

- **Ambiguity** is now 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 (conjecture).
- Our type inference builds the *most general* and *least ambivalent* derivation, and fails if it becomes ambiguous.
Advantages of refined ambiguity

– Compared to explicit types.
  ◦ Non-ambiguous types don’t need annotations.
  ◦ More programs are accepted outright.
  ◦ Less pressure for a clever propagation algorithm.
  ◦ Particularly useful if there are many local definitions.

– Compared to using type normalization [ML2011].
  ◦ Inferred types are more predictable.
  ◦ Leakage of ambivalent types precisely captures incompatible sharing, which was the cause of unsoundness combining GADTs and objects/polymorphic variants.
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 implements a relaxed version of OutsideIn.

- 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 GHC 7.
Comparison with GHC 7

- OCaml fails (while GHC 7 succeeds)

```ocaml
let f : type a. a t -> a = fun x ->
  let r = match x with Int -> 1 in r
  Error: This expression has type int but expected a
```

Insufficient propagation.

- GHC fails (while Ocaml succeeds)

```haskell
data T a where Int :: T Int

f :: T a -> ()
f x =
  let z = case x of {Int -> 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.
```
Strength and weakness of OutsideIn

- Constraint propagation is so strong that sometimes no annotation at all is needed.

```haskell
data R a where
  R1 :: R Int
  R2 :: a -> R a

test25 R1 = 1
test25 (R2 x) = x
-- test25 :: R t -> t
```

- To allow upward propagation, `let` is not implicitly generalized.

```haskell
test26 =
  let id x = x in (id "a", id True)
-- Fails
```
### Comparison

<table>
<thead>
<tr>
<th></th>
<th>OCaml</th>
<th>GHC</th>
</tr>
</thead>
<tbody>
<tr>
<td>GADTs</td>
<td>since 4.00</td>
<td>since 2005</td>
</tr>
<tr>
<td>Type discipline</td>
<td>ambiguity det.</td>
<td>OutsideIn + norm. ?</td>
</tr>
<tr>
<td>Polymorphic let</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Inference</td>
<td>unification-based</td>
<td>constraint-based</td>
</tr>
<tr>
<td>Principality</td>
<td>maybe</td>
<td>– (1)</td>
</tr>
<tr>
<td>Exhaustiveness check</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Type-level functions</td>
<td>–</td>
<td>✓</td>
</tr>
</tbody>
</table>

(1) OutsideIn itself only accepts derivations that are principal in the unrestricted type system.
This non-principal example for OutsideIn [JFP’11] is accepted:

```haskell
data V a where
  V1 :: Int -> V Bool
  V2 :: V a

test7 (V1 n) _ = n > 0
test7 V2 r = r
-- test7 :: V t -> Bool -> Bool
```

Here is an even stranger case:

```haskell
data T a where Int :: T Int

test14 (x::T a) (y::a) =
  case x of Int -> y
-- test14 :: T a -> a -> Int
```

Some kind of type normalization seems to be going on...
Combining ambiguity with OutsideIn

- Ambiguity detection could help GHC ?!

- In a final phase GHC allows constraints to leak from cases.

- One could restrict this final resolution to non-ambiguous types.
  
  - `test7` would be accepted as is.
  
  - `test14` would have the more natural type

    ```haskell
    test14 (x::T a) (y::a) =
    case x of Int -> y
    -- test14 :: T a -> a -> a
    ```

  - Inferred types would probably be principal with the restriction to non-ambiguous derivations.
Concluding remarks

Still working on the formalization.

- Graph-based approach vs. set-based approach.

Available in OCaml 4.00.

- See the Language extensions section of the reference manual.

- Examples: http://caml.inria.fr/cgi-bin/viewvc.cgi/ocaml/trunk/testsuite/tests/typing-gadts/

- Ambiguity detection is always active (required for soundness), but use `ocaml -principal` for “principal” propagation (this may slow down typing).
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.

- Using polymorphic types introduces a difficulty.
  - We must track (and copy) sharing inside them.
  - Needed for polymorphic recursion, etc.
  - Can be done simply by seeing types as graphs.
Set-based formalization

\[ \begin{align*}
\tau & ::= \alpha & \text{flexible variable} \\
& \quad | \varphi & \text{rigid variable} \\
& \quad | \tau \to \tau & \text{other types} \\
T & ::= \text{set of types } \tau \\
P & ::= \text{set of rigid variables } \varphi \\
\Gamma & ::= \emptyset \quad | \quad \Gamma, x : T \quad | \quad \Gamma, \varphi \quad | \quad \Gamma, \varphi \simeq \tau \quad \text{contexts}
\end{align*} \]

For \( T \) to be coherent under a context \( \Gamma \),

- It must be structurally decomposable:
  \[ T = \{ \alpha \} \text{ or } T = P \text{ or } T = T_1 \to T_2 \cup P \text{ or } T = (T_1)t \cup P \text{ or } \ldots \]

- Its types must be compatible with each other under \( \Gamma \).
  \[ \Gamma \vdash \tau_1 \simeq \tau_2 \text{ is the congruence closure of the equations of } \Gamma. \]
Basic inference rules

\[
\begin{align*}
\text{Var} & \quad x : T \in \Gamma \quad \Rightarrow \quad \Gamma \vdash x : T \\
\text{App} & \quad \Gamma \vdash a_1 : T_2 \to T_1 \cup P \quad \Gamma \vdash a_2 : T_2 \quad \Rightarrow \quad \Gamma \vdash a_1 \ a_2 : T_1 \\
\text{Let} & \quad \Gamma \vdash a_1 : T_1 \quad \Gamma \vdash [a_1/x]a_2 : T \\
& \quad \Rightarrow \quad \Gamma \vdash \text{let } x = a_1 \text{ in } a_2 : T \\
\text{Fun} & \quad \Gamma, x : T_0 \vdash a_1 : T_1 \\
& \quad \Rightarrow \quad \Gamma \vdash \text{fun } x \to a_1 : T_0 \to T_1 \cup P \\
\text{Ann} & \quad \Gamma \vdash a : T_1 \quad \tau \in T_1 \cap T_2 \\
& \quad \Rightarrow \quad \Gamma \vdash (a : \tau) : T_2 \\
\text{Match} & \quad \Gamma \vdash a_1 : T_1 \quad (\varphi)t \in T_1 \quad C : (\tau)t \\
& \quad \Gamma, \varphi \simeq \tau \vdash a_2 : T \\
& \quad \Rightarrow \quad \Gamma \vdash \text{match } a_1 : (\varphi)t \text{ with } C \to a_2 : T
\end{align*}
\]

All types must be coherent.
Type inference

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

- Nodes are the pair of a normal type node and a set of rigid variables.

- Infer polymorphic types as graphs, where each node may be polymorphic (i.e. allow the addition of rigid variables).

- In OCaml’s type inference algorithm, the extra types are actually held in a separate data structure, so that the modifications to the algorithm were minimal.