Formalizing OCaml GADT typing in Coq

Jacques Garrigue   Xuanrui Qi

Graduate School of Mathematics, Nagoya University

August 26, 2021

OCaml, GADTs and principality

- Principality of GADT inference is known to be difficult.
- OCaml proven to be principal thanks to ambivalent types, which allow to detect ambiguity escaping from a branch [Garrigue & Rémy, APLAS 2013].

```
type (_,_) eq = Refl : ('a,'a) eq;;

let f (type a) (w : (a,int) eq) (x : a) =      (* coherent *)
  let Refl = w in if x > 0 then x else x ;;
val f : ('a, int) eq -> 'a -> 'a
(* Principal for OCaml, rejected by GHC as ambiguous *)

let g (type a) (w : (a,int) eq) (x : a) =     (* ambiguous *)
  let Refl = w in if x > 0 then x else 0 ;;
Error: This instance of int is ambiguous:
       it would escape the scope of its equation
```

Ambivalent types in a nutshell

- Types that rely on GADT equations are represented as ambivalent types, which are a form of intersection types.
- Ambivalent types are only valid when equations are available, but their reliance on equations is implicit.

```
let f (type a) (w : (a,int) eq) (x : a) =
  let Refl = w in (* add the equation a = int *)
  if x > 0        (* this x has ambivalent type a ∧ int *)
  then x else x   (* but these have only type a *)
(* Hence the result is of type a *)
val f : ('a, int) eq -> 'a -> 'a
```

```
let g (type a) (w : (a,int) eq) (x : a) =
  let Refl = w in if x > 0
  then x   (* this x has type a *)
  else 0   (* but 0 has type int *)
(* The result has type a ∧ int, which becomes ambiguous *)
Error: This instance of int is ambiguous
```

Soundness and principality of inference

OCaml and Haskell (GHC) differ in their handling of Unification under GADT equations.

- In Haskell, unification under a GADT equation cannot involve variables from outside (OutsideIn).
- In OCaml, this is allowed as long as the equation is not required for the unification (ambivalence).

Relying on ambivalence

- is sound with respect to in-place unification
  $\Rightarrow$ tracks whether local unifications are valid outside.
- ensures principality of inference
  $\Rightarrow$ alternative types are rejected.

Disambiguation

- Type annotations hide the ambivalence, by separating inner and outer types.
- This solves ambiguities. The following are valid:

```
let g (type a) (w : (a,int) eq) (x : a) =
  let Refl = w in (if x > 0 then x else 0 : a) ;;
val g : ('a, int) eq -> 'a -> 'a

let g (type a) (w : (a,int) eq) (x : a) =
  let Refl = w in (if x > 0 then x else 0 : int) ;;
val g : ('a, int) eq -> 'a -> int
```

OCaml lets you write the annotation outside if your prefer.

But is it really principal?

When looking for reduction rules validating subject reduction, we came upon the following example:

```
let f (type a b) (w1 : (a, b -> b) eq)
                 (w2 : (a, int -> int) eq) (g : a) =
   let Refl = w1 in let Refl = w2 in g 3;;
val f : ('a, 'b -> 'b) eq -> ('a, int -> int) eq -> 'a -> 'b
```

```
let f (type a b) (w1 : (a, b -> b) eq)
                 (w2 : (a, int -> int) eq) (g : a) =
   let Refl = w2 in let Refl = w1 in g 3;;
val f : ('a, 'b -> 'b) eq -> ('a, int -> int) eq -> 'a -> int
```

- Changing the order of equations changes the resulting type.
- Bug in the theory: the ambivalence of `g` is not propagated to the result of the application `g 3`, failing to detect ambiguity.

But is it really principal?

When looking for reduction rules validating subject reduction, we came upon the following example:

```
let f (type a b) (w1 : (a, b -> b) eq)
                 (w2 : (a, int -> int) eq) (g : a) =
   let Refl = w1 in let Refl = w2 in g 3;;
val f : ('a, 'b -> 'b) eq -> ('a, int -> int) eq -> 'a -> 'b
```

```
let f (type a b) (w1 : (a, b -> b) eq)
                 (w2 : (a, int -> int) eq) (g : a) =
   let Refl = w2 in let Refl = w1 in g 3;;
val f : ('a, 'b -> 'b) eq -> ('a, int -> int) eq -> 'a -> int
```

- Changing the order of equations changes the resulting type.
- Bug in the theory: the ambivalence of `g` is not propagated to the result of the application `g 3`, failing to detect ambiguity.

Proving a fix in Coq

- We already proved soundness and principality for another fragment of OCaml, using a graph representation of types [Garrigue 2014, Structural Polymorphism].

$$\overline{\alpha :: \kappa}; \overline{x : \sigma} \vdash M : \alpha$$

  Here κ's are kinds, which describe nodes.
- By enriching the information in kinds with rigid variable paths, we can represent correct ambivalence.
- Principality is hard to prove, but subject reduction is already a good benchmark for a well-behaved type system.

Kinds and environments

- Kinds are constraints on a node, representing the graph structure: $\alpha = (\beta \to \gamma) \wedge a$ translates to

$$\alpha :: (\to, \{dom \mapsto \beta, cod \mapsto \gamma\})_a, \beta :: \bullet_{a.dom}, \gamma :: \bullet_{a.cod} \triangleright \alpha$$

- Grammar

ψ	$::=$	$\to \mid \mathrm{eq} \mid \ldots$	abstract constraint
C	$::=$	$\bullet \mid (\psi, \{l \mapsto \alpha, \ldots\})$	graph constraint
κ	$::=$	$C_{\bar{r}}$	kind
r	$::=$	$a \mid r.l$	rigid variable path
τ	$::=$	$r \mid \tau \to \tau \mid \mathrm{eq}(\tau, \tau)$	tree type
Q	$::=$	$\emptyset \mid Q, \tau = \tau$	equations
K	$::=$	$\emptyset \mid K, \alpha :: \kappa$	kinding environment
σ	$::=$	$\forall \bar{\alpha}.K \triangleright \alpha$	type scheme
Γ	$::=$	$\emptyset \mid \Gamma, x : \sigma$	typing environment
θ	$::=$	$[\alpha \mapsto \alpha', \ldots]$	substitution

Terms and Judgments

- Well-formedness

$$Q; K \vdash \kappa \qquad Q \vdash K \qquad Q; K \vdash \sigma \qquad Q; K \vdash \Gamma \qquad K \vdash \theta : K'$$

- Graph type instance of a tree type: $K \vdash \tau : \alpha$
- Terms

$$\begin{array}{rcll} M & ::= & x \mid c \mid \lambda x.M \mid M\ M \mid \mathsf{let}\ x = M\ \mathsf{in}\ M & \\ & \mid & (M : \tau) & \text{type annotation} \\ & \mid & \mathsf{Refl} & \text{witness introduction} \\ & \mid & \mathsf{type}\ a.M & \text{rigid variable introduction} \\ & \mid & \mathsf{use}\ M : \mathsf{eq}(\tau, \tau)\ \mathsf{in}\ M & \text{witness elimination} \end{array}$$

- Typing judgment

$$Q; K; \Gamma \vdash M : \alpha$$

Typing implies both $Q \vdash K$ and $Q; K \vdash \Gamma$.

Example

```
let f (type a) (w : (a,int) eq) (x : a) =
  let Refl = w in if x > 0 then x else x
```

can be encoded as

$$
\begin{aligned}
f = \;& \text{type } a.\lambda w.\lambda x.\\
& \quad \text{let } x = (x : a) \text{ in}\\
& \quad \text{use } w : \mathrm{eq}(a, \mathrm{int}) \text{ in } \mathit{ifpos}\; x\; x\; x
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{ifpos} : \;& \forall \alpha_1 :: \bullet_{\mathrm{int}}, \beta :: \bullet,\\
& \quad \alpha \;:: (\rightarrow, \{\mathit{dom} \mapsto \alpha_1, \mathit{cod} \mapsto \alpha_2\}),\\
& \quad \alpha_2 :: (\rightarrow, \{\mathit{dom} \mapsto \beta, \mathit{cod} \mapsto \alpha_3\}),\\
& \quad \alpha_3 :: (\rightarrow, \{\mathit{dom} \mapsto \beta, \mathit{cod} \mapsto \beta\}) \triangleright \alpha\\
\simeq \;& \forall \beta. \mathrm{int} \rightarrow \beta \rightarrow \beta \rightarrow \beta
\end{aligned}
$$

Selected typing rules

USE
$$\frac{Q; K; \Gamma \vdash M_1 : \alpha_1 \qquad K \vdash \mathrm{eq}(\tau_1, \tau_2) : \alpha_1 \qquad Q, \tau_1 = \tau_2; K; \Gamma \vdash M_2 : \alpha}{Q; K; \Gamma \vdash \mathrm{use}\ M_1 : \mathrm{eq}(\tau_1, \tau_2)\ \mathrm{in}\ M_2 : \alpha}$$

GC
$$\frac{Q; K, K'; \Gamma \vdash M : \alpha \quad \mathrm{FV}_K(\Gamma, \alpha) \cap \mathrm{dom}(K') = \emptyset}{Q; K; \Gamma \vdash M : \alpha}$$

VAR
$$\frac{Q \vdash K \quad Q; K \vdash \Gamma \quad x : \forall \bar{\alpha}. K_0 \triangleright \alpha \in \Gamma \quad K, K_0 \vdash \theta : K}{Q; K; \Gamma \vdash x : \theta(\alpha)}$$

APP
$$\frac{Q; K; \Gamma \vdash M_1 : \alpha \qquad Q; K; \Gamma \vdash M_2 : \alpha_2 \qquad \alpha :: (\to, \{dom \mapsto \alpha_2, cod \mapsto \alpha_1\})_{\bar{r}} \in K}{Q; K; \Gamma \vdash M_1\ M_2 : \alpha_1}$$

Detecting ambiguity

- Using VAR, APP, and GC, we can show that

$$a = int; K, \beta :: \bullet_a; \Gamma, x : \forall\alpha :: \bullet_a \triangleright \alpha \vdash ifpos\ x\ x\ x : \beta$$

so that we can apply USE.

- On the other hand, a minimal derivation for $g\ 3$ in

$$\text{let } g = (g : a) \text{ in use } w : \mathrm{eq}(a, int \rightarrow int) \text{ in } g\ 3$$

would be

$$a = int \rightarrow int; K, \beta :: \bullet_{int,a.cod}; \Gamma, g : \forall\alpha :: \bullet_a \triangleright \alpha \vdash g\ 3 : \beta$$

which becomes ambiguous when USE removes $a = int \rightarrow int$.

Coq development

- Based on "A certified implementation of ML with structural polymorphism and recursive types" [Garrigue 2014].
- Itself based on Arthur Charguéraud's development, using locally nameless cofinite quantification ("Engineering Metatheory" [Aydemir et al. 2008]).
- Avoided unification in the type system by interpreting Q as the set of its (rigid) unifiers.
- Finished proofs of subject reduction for following rules:

$$
\begin{array}{rcl}
(\lambda x.M)\ V & \longrightarrow & M[V/x] \\
\text{let } x = V \text{ in } M & \longrightarrow & M[V/x] \\
c\ V_1 \ldots V_n & \longrightarrow & \delta_c(V_1, \ldots, V_n) \\
(M_1 : \tau_2 \to \tau_1)\ M_2 & \longrightarrow & (M_1\ (M_2 : \tau_2) : \tau_1) \\
(M_1 : r)\ M_2 & \longrightarrow & (M_1\ (M_2 : r.dom) : r.cod) \\
\text{use } \mathit{Refl} : \mathrm{eq}(\tau_1, \tau_2) \text{ in } M & \longrightarrow & M
\end{array}
$$

Relation to principality

- Subject reduction and principality are independent properties.
- For ML-like type systems, principality is usually the combination of:

  Monotonicity A type derivation is still valid using a stronger Γ (where types are more polymorphic).[1]

  Most General Unifier Unification of types admits a most general solution.

- Existence of MGU relies on the ability to decompose types, which is also exactly what we needed to prove subject reduction for annotated applications.

$$(M_1 : r)\ M_2 \longrightarrow (M_1\ (M_2 : r.dom) : r.cod)$$

[1] OutsideIn does not satisfy monotonicity, and is not strictly principal.

Remaining work

- Prove type soundness
  Simpler to use translation into an explicit type system.
  Some formalization of soundness of GADTs already exists [Ostermann & Jabs, ESOP 2018]
- Prove principality
  This is hard, but a first step is existence of MGU.
- Soundness of type inference
  Another role of ambivalence is to ensure the soundness of inference. It would be interesting to prove it for weaker (non-principal) versions of the type system.

Further applications

- Graph types are also used inside OCaml to enforce the principality of first-class polymorphism and first-class modules.

```
module type Id = sig val id : 'a -> 'a end;;
fun (m : (module Id)) ->
         let module M = (val m) in M.id m;;
- : (module Id) -> (module Id) = <fun>

fun m -> ignore (m : (module Id));
         let module M = (val m) in M.id m;;
Warning: this module unpacking is not principal.
```

- Basic idea: a type is known if it is not shared with Γ.
- Extension should be straightforward.

Other approaches to soundness

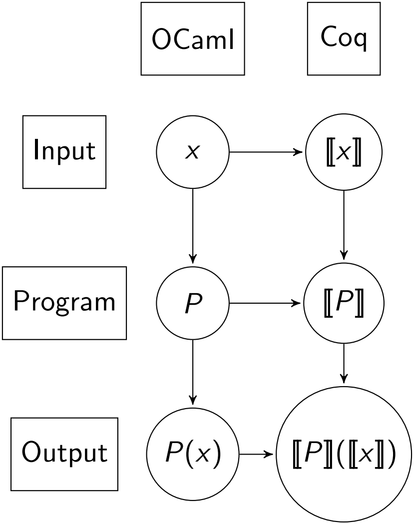
We are also investigating other ways to make OCaml type inference more robust.

Directly by making internal data-structures abstract, and having unification follow precise laws. Ultimately, the type inference algorithm should look like its formal definition. (with Takafumi Saikawa)

Indirectly by translating the type annotated source tree into Gallina programs, and relying on Coq's type soundness.

`https://www.math.nagoya-u.ac.jp/~garrigue/cocti/`

Soundness by translation



If for all $P : \tau \to \tau'$ and $x : \tau$

- P translates to $[\![P]\!]$, and $\vdash [\![P]\!] : [\![\tau \to \tau']\!]$
- x translates to $[\![x]\!]$, and $\vdash [\![x]\!] : [\![\tau]\!]$
- $[\![P]\!]$ applied to $[\![x]\!]$ evaluates to $[\![P(x)]\!]$

then the soundness of Coq's type system implies the soundness of OCaml's evaluation