Towards a Coq Specification for Generalized Algebraic Datatypes in OCaml

Jacques Garrigue Xuanrui Qi

Graduate School of Mathematics, Nagoya University

January 19, 2021

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 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            (* can infer result *)

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
```

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

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

- Terms

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

- Typing judgment

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

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

Selected typing rules

Use
$$\frac{Q; K; \Gamma \vdash M_1 : \theta(\alpha_0) \quad Q, \tau_1 = \tau_2; K; \Gamma \vdash M_2 : \alpha \qquad [\![\mathrm{eq}(\tau_1, \tau_2)]\!] = \forall \bar{\alpha}. K_0 \triangleright \alpha_0 \quad K_0 \vdash \theta : K}{Q; K; \Gamma \vdash \mathrm{use}\ M_1 : \mathrm{eq}(\tau_1, \tau_2)\ \mathrm{in}\ M_2 : \alpha}$$

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

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 substitution lemmas, but "interesting" cases of subject reduction remain.

$$
\begin{array}{rcl}
(M_1 : \tau_2 \to \tau_1)\ M_2 & \longrightarrow & (M_1\ (M_2 : \tau_2) : \tau_1) \\
(M_1 : r)\ M_2 & \longrightarrow & (M_1\ (M_2 : r.dom) : r.cod)
\end{array}
$$

Challenges and benefits of a legacy codebase

- The codebase is old, but updating was not too difficult.
- Freshness of variables relies on automation; easily broken but not hard to fix.
- Many interactions mean many new lemmas, and longer proofs.
- No need to revise the overall structure of proofs.
  Experienced no major technical stumbling block.
- Locally nameless quantification still seems a good fit.
- Question: would it be better to switch to a standard decision procedure for set disjointness?