Formalizing OCaml GADT typing in Coq

Jacques Garrigue
garrigue@math.nagoya-u.ac.jp
Nagoya University
Graduate School of Mathematics

Xuanrui Qi
xuanrui@nagoya-u.jp
Nagoya University
Graduate School of Mathematics

Abstract

The principality of OCaml's GADT type inference relies on the concept of *ambivalence*, which allows to reject programs whose typing is ambiguous. Yet, ambivalence itself requires to use a graph view of types, making inductive reasoning more difficult. In this talk, we present our ongoing work on mechanizing the metatheory of a core language for modern OCaml and formally proving soundness and principality for this core language. Our core language supports structural polymorphism, recursive types, and type-level equality witnesses, which are the defining features of OCaml type inference as of version 4.12. We have now proved subject reduction for a meaningful subset of the reduction rules.[1]

CCS Concepts • **Software and its engineering** → **Semantics**; **Data types and structures**; **Formal software verification**;

Keywords OCaml, formal specification, generalized algebraic datatypes

1 Introduction

Generalized algebraic datatypes (GADTs) have been an OCaml language extension since version 4.00, and is now arguably an integral part of the OCaml type system: prominent projects built in OCaml, such as Coq [6] and Tezos [7], use GADTs in production.

Type inference for GADTs is known to be a difficult problem, in particular when one wants principality. It is also tricky to implement, and OCaml has seen a number of soundness bugs over the years. We intend to address these problems by mechanizing the metatheory [1] of a core language of OCaml, equipped with three defining features of the modern OCaml type system — structural polymorphism, recursive types, and type-level equations — the last of which is equivalent to GADTs.

Formally, OCaml GADTs are based on ambivalent type inference [4], which offers principality. However, the soundness of the ambivalent type system was obtained through a translation into another type system. When trying to prove subject reduction for ambivalent types with type annotations in a more direct way, we actually discovered that it failed, and that this failure came from a lack of principality in the type inference. We have been able to fix this failure by enforcing a well-formedness condition on graph types containing rigid type variables.

We build our work on one of the authors' previous work [2], which contains a formalized metatheory of a variant of ML with structural polymorphism and recursive types; we extend the aforementioned work with rigid (i.e., "existential") type variables a (distinct from flexible type variables α), type-level equations over these rigid type variables, and a notion of ambivalence [4] inside types, which denotes types whose coherence relies on those type equations.

2 The bug, or "why subject reduction"

Early on in our formalization work, we discovered that there was an error in the principality proof for ambivalence type inference, which can be observed in the following OCaml example (even using the `-principal` option).

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

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
```

The resulting type depends on the order in which the equations were used! Since the ambivalent type system does not prioritize between equations according to such order, and the two types are clearly not equivalent, this denotes a loss of principality, which we could track back to a missing assumption about ambivalent unification.

Fortunately, this can be fixed by "inheriting" ambivalence from types to their subparts, i.e. not only the type of `g`, but also that of `g 3` should be ambivalent, and depend on the equation used. This is now done in the upcoming OCaml 4.13, when using `-principal`. In OCaml's implementation this relies on a notion of scoping level, which made the change easy.

Having already seen a hole in the proof, we would like to be more certain of its principality. While adapting the full fledge principality proof for structural polymorphism [2] is a daunting task, proving subject reduction for a well chosen set of rules is already a good hint. While the two notions

[1] A preliminary report on this work was presented at CoqPL'21 [5]. At that point, we had not yet formally proved any property of the formalization. This version adds new concepts, and concrete proofs.

are independent, the proof of principality for ambivalent types relies on the monotonicity of typing, i.e. replacing an assumption by a more general one should not break typability[2]. This is exactly what happens with β-reduction, as the substituted term may have a more general type. However, to make preservation possible, we also need to keep type annotations in other reduction rules, since ambivalent typability relies on these annotations.

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

When r is a rigid variable, we have to find annotations for M_2 and $(M_1 M_2)$. This need itself suggests the solution: we can introduce a notion of *rigid path*, which describes the type corresponding to a subcomponent of a rigid variable.

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

It turns out that seeing such rigid paths as ambivalent, since they are only meaningful in presence of an equation $r = \tau_1 \to \tau_2$, is all we need to fix the formal type system.

3 Coq formalization

Our Coq formalization is essentially an extension of the formalization of structural polymorphism [2] to ambivalent types [4]. While structural polymorphism distinguished between types and kinds, the latter being used to represent polymorphic record or variant types, the ambivalent approach takes a more radical view. Now types are degenerate, keeping only type variables α, and all other type constructors are represented by kinds κ (see Fig. 1)., which contain a structural constraint ψ, type variables corresponding to children of the constructor, and a list $\bar{r}$ of rigid paths. For instance, here is the representation of the type $(\beta \to \gamma) \wedge a$, corresponding to the ambivalent type being both the rigid variable a and a function type.

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

A typing judgment now uses three environments: a set Q of rigid equations, expressed using the classical arborescent view of types, and containing only rigid variables; a kinding environment K mapping flexible type variables to kinds; and a typing environment Γ mapping term variables to type schemes.

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

One can get a glimpse of the extra complexity by considering the auxiliary judgments required to define typing rules

[2]Some type systems do not have monotonicity, and as a result they usually cannot have principality of the type system itself, but only a weaker property, such as principality of some typings.

$$\begin{array}{lcll}\psi & ::= & \to \mid \mathrm{eq} \mid \ldots & \text{structural constraint} \\ C & ::= & \bullet \mid (\psi, \{l \mapsto \alpha, \ldots\}) & \text{node constraint} \\ \kappa & ::= & C_{\bar{r}} & \text{kind} \\ r & ::= & a \mid r.l & \text{rigid variable path} \\ \tau & ::= & r \mid \tau \to \tau \mid \mathrm{eq}(\tau, \tau) & \text{tree type} \\ Q & ::= & \emptyset \mid Q, \tau = \tau & \text{equations} \\ K & ::= & \emptyset \mid K, \alpha :: \kappa & \text{kinding environment} \\ \sigma & ::= & \forall \bar{\alpha}.K \triangleright \alpha & \text{type scheme} \\ \Gamma & ::= & \emptyset \mid \Gamma, x : \sigma & \text{typing environment} \\ \theta & ::= & [\alpha \mapsto \alpha', \ldots] & \text{substitution}\end{array}$$

Figure 1. Type related notations

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

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

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

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

Figure 2. Selected typing rules

in Fig. 2.

$Q; K \vdash \kappa$	well-formedness of kinds
$Q \vdash K$	$\forall \alpha :: \kappa \in K,\ Q; K \vdash \kappa$
$Q; K \vdash \sigma$	well-formedness of type schemes
$Q; K \vdash \Gamma$	$\forall x : \sigma \in \Gamma,\ Q; K \vdash \sigma$
$K \vdash \alpha : \tau$	instantiation of tree types
$K \vdash \theta : K'$	well-kindedness of substitutions

All but the well-kindedness of substitutions were freshly introduced for ambivalent typing.

4 Current status

Our Coq formalization can be found on GitHub [3]. We have now proved preservation for the following reduction rules.

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

The last three rules embody the semantics of ambivalence and GADT typing. Preservation for any of them was not yet proven in our previous report [5], partly because we had not yet properly defined the instantiation of tree types and a number of related notions.

Preservation alone does not entail soundness: we are yet to prove preservation for the construct `(type a)` permitting to turn a rigid type variable into a flexible one, and eventually quantify it, so that progress cannot be proven for full programs. The main difficulty with flexibilization is that terms lack the type information that should be substituted.

From here on, it actually seems simpler to prove soundness through a translation of type derivations into a more classical type system with explicit type instantiation, and no tracking of ambivalence, and prove principality independently of reduction. Both are new lines of work, but we can now be relatively confident that the fundamental pieces are in place.

Acknowledgments

We would like to thank the Tezos Foundation for funding the Certifiable OCaml Type Inference (COCTI) project, of which this work is part.

References

[1] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008.

[2] Jacques Garrigue. A certified implementation of ml with structural polymorphism and recursive types. *Mathematical Structures in Computer Science*, 25, 2015.

[3] Jacques Garrigue and Xuanrui Qi. A Coq formalization of ambivalent types, 2021. https://github.com/COCTI/certint-amb.

[4] Jacques Garrigue and Didier Rémy. Ambivalent types for principal type inference with gadts. In *Programming Languages and Systems (APLAS)*, 2013.

[5] Xuanrui Qi and Jacques Garrigue. Towards a Coq specification for generalized algebraic datatypes in OCaml. In *Presentation at CoqPL'21*, January 2021.

[6] The Coq Development Team. The Coq proof assistant, 2021. https://coq.inria.fr.

[7] The Tezos Team. Tezos, 2021. https://tezos.com/.