Formalizing OCaml GADT typing in Coq

Jacques Garrigue

Nagoya University
Graduate School of Mathematics

garrigue@math.nagoya-u.ac.jp

Xuanrui Qi

Nagoya University
Graduate School of Mathematics

xuanrui@nagoya-u.jp

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 work, 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 𝜅), 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 fledged 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

1A 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 typabil-
ity\(^2\). This is exactly what happens with \(\beta\)-reduction, as the
substituted term may have a more general type. However, to
make preservation possible, we also need to keep type annota-
tions in other reduction rules, since ambivalent typability
relies on these annotations.

\[
\begin{align*}
(M_1 : \tau_2 \rightarrow \tau_1) M_2 &\quad\longrightarrow\quad (M_1 (M_2 : \tau_2) : \tau_1) \\
(M_1 : r) M_2 &\quad\longrightarrow\quad (M_1 (M_2 : ?) : ?)
\end{align*}
\]

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.

\[
(M_1 : r) M_2 \quad\longrightarrow\quad (M_1 (M_2 : r.\text{dom}) : r.\text{cod})
\]

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

3 Coq formalization

Our Coq formalization is essentially an extension of the for-
malization of structural polymorphism [2] to ambivalent
types [4]. While structural polymorphism distinguished be-
tween types and kinds, the latter being used to represent
polymorphic record or variant types, the ambivalent ap-
proach takes a more radical view. Now types are degenerate,
keeping only type variables \(\alpha\) and all other type construc-
tors are represented by kinds \(\kappa\) (see Fig. 1), which contain
a structural constraint \(\psi\), type variables corresponding to
children of the constructor, and a list \(\vec{r}\) of rigid paths. For
instance, here is the representation of the type \((\beta \rightarrow \gamma) \land a\),
corresponding to the ambivalent type being both the rigid
variable \(a\) and a function type.

\[
\alpha ::= (\rightarrow, \{\text{dom} \mapsto \beta, \text{cod} \mapsto \gamma\})_a, \beta ::= \bullet_{a.\text{dom}}, \gamma ::= \bullet_{a.\text{cod}} \triangleright \alpha
\]

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 \(\Gamma\) mapping term variables to type
schemes.

\[
Q : K ; \Gamma \vdash M : \alpha
\]

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

\[
\psi :: \rightarrow | \text{eq} | \ldots \\
C :: \bullet \rightarrow (\psi, \{I \mapsto \alpha, \ldots\}) \\
\kappa :: C_r \\
r :: \alpha \mid r.l \\
\tau :: \text{r} \rightarrow \text{r} | \text{eq}(\text{r}, \text{r}) \\
Q :: \emptyset | Q, \text{r} = \text{r} \\
K :: \emptyset | K, \alpha :: \kappa \\
\sigma :: \forall \bar{\alpha}.K \rightarrow \alpha \\
\Gamma :: \emptyset | \Gamma, x : \sigma \\
\theta :: [\alpha \mapsto \alpha', \ldots]
\]

\[\psi\] structural constraint

\[C\] node constraint

\[\kappa\] kind

\[r\] rigid variable path

\[\tau\] tree type

\[Q\] equations

\[K\] kinding environment

\[\sigma\] type scheme

\[\Gamma\] typing environment

\[\theta\] substitution

Figure 1. Type related notations

\[
\begin{align*}
\text{VAR} & : Q : K ; K + \Gamma \quad x : \forall \bar{\alpha}.K_0 \triangleright \alpha \in \Gamma \\
& \quad K_0 \vdash K \\
& \quad Q : K ; K + \theta : K \\
Q, K & \vdash x : \theta(\alpha) \\
\text{APP} & : Q, K; \Gamma + M_1 : \alpha \\
& \quad Q, K; \Gamma + M_2 : \alpha_2 \\
& \quad \alpha ::= (\rightarrow, \{\text{dom} \mapsto \alpha_2, \text{cod} \mapsto \alpha_1\})_e \in K \\
& \quad Q : K ; K + M_1 : \alpha_1 \\
& \quad Q ; \text{r}_1 = \text{r}_2; K ; \Gamma + M_2 : \alpha \\
& \quad \text{USE} \quad Q : K ; \Gamma + \text{use} M_1 : \text{eq}(\text{r}_1, \text{r}_2) \text{ in } M_2 : \alpha \\
\text{GC} & : Q, K ; K' \vdash M : \alpha \\
& \quad \text{FV}_K(\Gamma, \alpha) \cap \text{dom}(K') = \emptyset \\
& \quad Q : K ; K + M : \alpha
\end{align*}
\]

Figure 2. Selected typing rules

in Fig. 2.

\[
\begin{align*}
Q, K + \kappa & \quad \text{well-formedness of kinds} \\
Q + K & \quad \forall \alpha : \kappa \in Q, K + \kappa \\
Q, K + \sigma & \quad \text{well-formedness of type schemes} \\
Q, K + \Gamma & \quad \forall \sigma \in \Gamma, Q + K + \sigma \\
K + \alpha : \tau & \quad \text{instantiation of tree types} \\
K + \theta : K' & \quad \text{well-kindness of substitutions}
\end{align*}
\]

All but the well-kindness 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.

\[
\begin{align*}
(\lambda x. M) V &\quad \longrightarrow\quad M[V/x] \\
\text{let } x = V \text{ in } M &\quad \longrightarrow\quad M[V/x] \\
\text{e } V_1 \ldots V_n &\quad \longrightarrow\quad \delta_e(V_1, \ldots, V_n) \\
(M_1 : \tau_2 \rightarrow \tau_1) M_2 &\quad \longrightarrow\quad (M_1 (M_2 : \tau_2) : \tau_1) \\
(M_1 : r) M_2 &\quad \longrightarrow\quad (M_1 (M_2 : r.\text{dom}) : r.\text{cod}) \\
\text{use \text{Refl} : eq(\text{r}_1, \text{r}_2) \text{ in } M} &\quad \longrightarrow\quad M
\end{align*}
\]

\[^2\text{Some type systems do not have monotonicity, and as a result they usually cannot have principality of the type system itself, but only a weaker property, such as principality of some typings.}\]
Formalizing OCaml GADT typing in Coq

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