Towards a Coq Specification for Generalized Algebraic Datatypes in OCaml

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

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

Abstract

Generalized algebraic datatypes have been a part of the OCaml type system, but there has been no formal specification of them. In this talk, we present our ongoing work on mechanizing the metatheory of a core language for modern OCaml and formally proving the soundness of 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 4.11.

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

Keywords OCaml, formal specification, generalized algebraic datatypes

ACM Reference Format:
Xuanrui Qi and Jacques Garrigue. 2021. Towards a Coq Specification for Generalized Algebraic Datatypes in OCaml. In *Proceedings of CoqPL '21: The 7th International Workshop on Coq for Programming Languages (CoqPL '21).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [7] and Tezos [8], 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 [5], 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 [3], which contains a formalized metatheory of a variant of ML with structural polymorphism and recursive types; we extend the aforementioned work with type-level equations and a notion of ambivalence [5] in types, which serves to distinguish rigid (i.e., "existential") type variables a, which are constrained by type equations and thus could not be freely instantiated, and flexible variables α, which are type variables in the usual sense, and which is necessary during type inference.

2 Basic setup

Following [3], we begin with a slightly unusual setup. Types are trivial in our setup, consisting only of variables:

$$\xi ::= \alpha.$$

In our Coq development, we use the locally nameless encoding [2] for variable bindings, and a type is represented as:

$$\begin{array}{l}\texttt{Inductive}\ typ : \texttt{Set} := \\ \mid typ\_bvar : \mathbb{N} \to typ \\ \mid typ\_fvar : var \to typ.\end{array}$$

The kinds are more unusual, and contain all the interesting information. A kind in our formalization represents all the constraints on a type:

$$\begin{array}{lcl}\kappa & ::= & \bullet_{\bar{r}} \mid (C, \{l_1 : \xi_1, \ldots, l_n : \xi_n\})_{\bar{r}} \\ r & ::= & a \mid r.l\end{array}$$

where $\bar{r}$ is a list of rigid paths which should all be equal to the type described by this kind.

The tuple part of a kind is actually represented in our Coq development as a record of four entries (two of the entries serve to make sure that ψ is well-formed):

CoqPL '21, Jan 19, 2021, Copenhagen, Denmark
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
Record ckind : Set := Kind {
  kind_cstr : Cstr.cstr;
  kind_valid : Cstr.valid kind_cstr;
  kind_rel : list (Cstr.attr × typ);
  kind_coherent : coherent kind_cstr kind_rel }.
```

We refer the reader to [3] for more on this constraint-based type system.

3 Formalizing ambivalence

In our work, we make a distinction between *simple types* and *simple kinds*, which are types and kinds that actually occur in the source program, and *ambivalent types* (and kinds), which are the types and kinds that appear internally in our type system. Simple types are akin to the usual ML types:

$$\tau ::= \alpha \mid a \mid \tau \to \tau \mid \tau = \tau.$$

Simple kinds are just the kinds that use simple instead of ambivalent types.

One important difference between simple and ambivalent types is that simple types are just trees as one would usually think. However, since types could be recursive and thus contain circular references, the ambivalent types are actually graphs. A simple type can, however, be converted to an ambivalent type by pointing the recursive variable references to the correct node, which we define in a function `graph_of_tree_type` (see [6], `ML_SP_Definitions.v` for details).

Besides the basic expression forms already present in [3], we add four new kinds of terms coming from [5]:

$$M ::= \ldots \mid \texttt{Eq} \mid \texttt{use}\ M_1 : \tau_1\ \texttt{in}\ M_2 \mid \nu(a)M \mid (\tau)$$

We also modify the rules relating to the well-formedness of kinds and types according to [5], since adding ambivalent type variables also adds new constraints to the types and kinds. The missing well-formedness condition we mentioned in introduction can be described as follows.

Wf-Kind-Attrs

$$\frac{\forall r \in \bar{r} \qquad \Delta \vdash r \mapsto C \\ \forall (l, \alpha) \in R \text{ s.t. } C \vdash \texttt{unique}(l) \qquad K(\alpha) = \kappa_{\bar{r}'} \qquad r.l \in \bar{r}'}{K; \Delta \vdash (C, R)_{\bar{r}}}$$

Here, K is a kinding environment, mapping type variables to their kinds, Δ is a set of equations on simple types, and $\Delta \vdash a.l_1 \ldots l_n \mapsto C$ means that for any solution θ of the equations in Δ, the type constructor at position indicated by the path $l_1 \ldots l_n$ in $\theta(a)$ should correspond to the constraint C. We require that all unique attributes for the constraint C, corresponding to the arguments of this type constructor, do propagate the rigid paths to the types they point to. There is an ongoing OCaml PR [4] that enforces the same condition using internal levels.

Our typing rules basically follow [5], except that we adapt the rules to our new typing discipline. This gives us judgments of the form

$$\Delta; K; \Gamma \vdash t : \alpha$$

where Δ and K are respectively an equation set and a kinding environment, and Γ is a typing environment mapping term variables to type schemes, that may contain variables in K. For the sake of simplicity, we do not present the rules here; the interesting reader may consult our Coq development ([6], `ML_SP_Definitions.v`).

4 Coq development

Currently, we are working on proving the soundness of our type system by showing progress and subject reduction. Since we have changed many definitions in the original development [3] (the file `ML_SP_Definitions.v` has doubled to reach about 900 lines), many proofs need to be modified. At the time of this submission, the file `ML_SP_Infrastructure.v`, which has almost been completely adapted to the new type system, has seen about 400 extra lines. However, we expect many more lines added to `ML_SP_Soundness.v`, which contains the proofs of the main soundness theorems. One reason the proofs become more complicated is that, in the aforementioned typing judgement, we have now well-formedness conditions of the form $\Delta \vdash K$ and $\Delta; K \vdash \Gamma$, i.e. there are dependencies between the different enviroments, whereas their well-formedness could be checked independently before.

As we have just mentioned, our Coq development differs from typical proof projects in Coq, in that instead of starting from scratch, we base our development on a project that was mostly dormant for about 10 years. Thus, maintaining our proof scripts become an important engineering task. There has been few accounts of maintaining Coq projects over long time spans, and we intend to discuss the proof engineering aspects of our project as an interesting case study.

Acknowledgments

We would like to thank the Tezos Foundation for funding our Certifiable OCaml Type Inference (COCTI) project, which supports our work.

References

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

[2] Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49 (2012). Issue 3.

[3] Jacques Garrigue. 2015. A Certified Implementation of ML with Structural Polymorphism and Recursive Types. *Mathematical Structures in Computer Science* 25 (2015). Issue 4.

[4] Jacques Garrigue. 2020. OCaml PR #9815. https://github.com/ocaml/ocaml/pull/9815
[5] Jacques Garrigue and Didier Rémy. 2013. Ambivalent Types for Principal Type Inference with GADTs. In *Programming Languages and Systems (APLAS 2013)*.
[6] Xuanrui Qi and Jacques Garrigue. 2020. A Coq Formalization of Ambivalent Types. https://github.com/COCTI/certint-amb
[7] The Coq Development Team. 2020. The Coq Proof Assistant. https://coq.inria.fr
[8] The Tezos Team. 2020. Tezos. https://tezos.com/