Adding GADTs to OCaml: the direct approach

Jacques Garrigue[*] Jacques Le Normand[†]

1 Abstract

Generalized Algebraic Datatypes, or GADTs, extend algebraic datatypes by allowing an explicit relation between type parameters and case analysis. They have useful applications, among others for encoding invariants of data structures, or providing tagless data representations.

We have implemented them in OCaml, by directly modifying the type inference engine. We discuss the technical choices involved, and the properties expected to hold. In particular, we have worked on two aspects of inference: principality, which holds only by requiring some derivations to be minimal, and exhaustiveness of pattern-matching, which requires a new notion of incompatibility.

2 Introduction

GADTs, which can be seen as a limited version of the inductive types available in some dependent type theories, are a useful addition to programming language type systems. Their applications are two-fold: they allow to write proofs inside programs, and to use these proofs to reduce dynamic checks, allowing for instance tagless data representations.

Haskell has been the first mainstream functional programming language to integrate GADTs. Their implementation has already gone through two iterations, first using wobbly types [3], and now in GHC 7 as part of the OutsideIn inference framework [5]. Programming languages of the ML family have been slower to adopt these new friends. For OCaml, Pottier and Régis-Gianas [4] have proposed a stratified approach, where type inference for GADTs is separated from the bulk of the language. We have chosen to go for a more direct approach, directly integrating GADT type inference inside the core engine, closer in spirit to the original wobbly types. While the interaction with other parts of the language may be subtle, this avoids duplicating an already complex inference system.

Documentation for this extension is available at the following URL: `https://sites.google.com/site/ocamlgadt/`. It shall be included in the main OCaml version before the workshop.

3 Examples

We first present some basic examples demonstrating GADTs in OCaml.

[*]Nagoya University, Graduate School of Mathematics
[†]Lexifi, Paris

Type definitions reuse the original syntax for algebraic datatypes, extending it with inductive cases, including a distinct return type for each constructor, in the style of Coq or Haskell. Since type parameters are not used for inductive cases, we allow to replace them with underscores.

```
type _ expr =
  | Int : int -> int expr
  | Add : (int -> int -> int) expr
  | App : ('a -> 'b) expr * 'a expr -> 'b expr
```

Return types must be instances of the defined type. Here `Int` forces its parameter to be `int`, `Add` to be `int -> int -> int`. Type variables which do not appear in the return type are handled as existential types in pattern matching.

Functions using GADTs require some amount of type annotations. The easiest approach is to give the function's signature, using a new syntax where type variables are replaced by abstract type parameters. This annotation also supports polymorphic recursion.

```
let rec eval : type t. t expr -> t = function
  | Int n -> n                        (* t = int *)
  | Add -> (+)          (* t = int -> int -> int *)
  | App (f, x) -> eval f (eval x)
          (* ∃u. f : (u -> t) expr ∧ x : u expr *)
val eval : 'a expr -> 'a
```

Pattern matching allows to exploit GADT equations, which we have indicated in comments. Thanks to them we are able to give the type `t` to each case.

An interesting application of GADTs is the ability to work with tagless data structures. Here is an example where we define a singleton type, and use it to analyze arbitrary datastructures as long as we can represent their type.

```
type _ ty =
  | Tint : int ty
  | Tbool : bool ty
  | Tpair : 'a ty * 'b ty -> ('a * 'b) ty

let rec print : type a. a ty -> a -> string =
  fun t d ->
  match t, d with
  | Tint, n -> string_of_int n
  | Tbool, true -> "true"
  | Tbool, false -> "false"
  | Tpair (ta, tb), (a, b) ->
      "(" ^ print ta a ^ ", " ^ print tb b ^ ")"
val print : 'a ty -> 'a -> string
```

Inside pattern matching, type inference progresses from left to right, allowing subsequent patterns to benefit from type equations generated in the previous ones. Here this means that `d` has type `int` on the first line, `bool` on the 2nd and 3rd line, and `t * u` on the 4th line, with `t` and `u` fresh abstract types such that `ta : t ty` and `tb : u ty`, and we can pattern-match on it accordingly.

4 Basic type inference

Sound type inference for GADTs is actually easy. We just need to extend the typing of pattern matching, so that GADT constructor matching adds type equations to the typing environment. In order to make this task easier, the type variables introduced in type annotations in the above examples are actually abstract type parameters, which were introduced in OCaml 3.12 [1]. One can later associate an equation to such abstract type parameters by adding a manifest type to their definition, turning them into type abbreviations. Correct scoping of existential types also comes for free from the implementation of local modules.

After exiting a branch, one must forget those equations. Again this is done trivially by going back to the original type environment. The only remaining difficulty is that we need to check the return type for each case before forgetting these local equations. This is done by propagating the type information provided by type annotations backward. This kind of backward propagation of expected type information was already done to some extent by OCaml's type checker.

5 Variables and type system issues

While type inference is conceptually easy, there were still a number of issues proper to OCaml.

A first one is the handling of type variables. OCaml already has two kinds of type variables used in type annotations. Contrary to Standard ML or Haskell, usual type variables in OCaml are just unification variables, and they scope over the whole function. Explicitly scoped universal type variables were introduced for first-class polymorphism, but they are limited to single type annotations. Neither of them fit well with GADTs. Rather than create a third category of type variables, we have chosen to use abstract type parameters, with the advantages described above. Interestingly, the type annotations in examples above are just syntactic sugar. Namely, the definition for `print` translates to:

```
let rec print : 'a. 'a ty -> 'a -> string =
  fun (type a) ->      (* introduce a fresh abstract type *)
      ((fun t d -> ...) : a ty -> a -> string)
```

Another technical issue was that OCaml's unification chooses to share not only variables but internal type nodes too. This is clearly incompatible with the presence of local type annotations, since this sharing could be invalidated when we leave a branch. Fortunately, the algorithm could easily be modified to disable this structure sharing when the type environment contains local equations. Unfortunately, at this point objects and polymorphic variants require this sharing, and some combinations of them and GADTs are not allowed.

A more subtle point concerns variance annotations. A simple answer is to disallow variance for instantiated type parameters.

6 Exhaustiveness and incompatibility

GADTs do also have an impact on the exhaustiveness check for pattern matching. Namely, type information often makes some cases impossible, and we would expect the checker to take this into account. We have implemented such a refinement by extending OCaml's original exhaustiveness checker. Rather than returning only the first counter example found, we go on collecting them for all cases, and then check whether some of these counter examples can be eliminated. While this strategy is safe (we never forget a missing case), it is not complete, as we may sometimes fail to detect some impossibility. A more exact check would cause a combinatorial explosion, which we have carefully avoided.

Another subtle issue concerns what we should see as impossible. Due to ML's abstract types, some types that are not unifiable during pattern matching may have been unifiable in some other module, meaning that it would be unsound to prune all untypable branches. We solve this by defining an incompatibility relation, which only relates provably distinct types. Some case may be neither incompatible nor typable, but we choose to just ignore those non-unifiable types, going on typing the case without adding extra equations. This ensures that we can always write a complete pattern matching without using wild cards.

7 Completeness and principality

Type inference for GADTs is known to be incomplete with respect to a naive type system. Building on our experience with first-class polymorphism [2], we choose to distinguish between safe type information, which comes to some program point straight from a type annotation, and unsafe information, which would reach it through unification in some sibling expression. We can track this by observing whether a type's structure is shared with the context or not. While pattern matching inference is algorithmic, we only give to it as input the unshared part of the type of its scrutinee. Similarly, for the return type and external type variables, we only communicate unshared parts from outside the pattern matching. Reciprocally, all information from inside the pattern matching needs to be canonicalized (by expanding all local type abbreviations) before being allowed to escape.

Practically this approach combined with the above exhaustiveness check seems to be expressive enough, as we are able to type all examples from [6] annotating only type signatures on functions, and omitting all unreachable cases.

The resulting algorithm is expected to be complete with respect to a specification in the style of [2], and the type inferred be principal when we restrict some type derivations to be minimal. We are still investigating some ways to improves the properties of type inference, by avoiding the need for canonicalization, and restoring the environment weakening lemma, which appears to fail currently for all GADT inference systems.

References

[1] A. Frisch and J. Garrigue. First-class modules and composable signatures in Objective Caml 3.12. In *Workshop on ML*, Baltimore, MD, Sept. 2010.

[2] J. Garrigue and D. Rémy. Extending ML with semi-explicit higher order polymorphism. *Information and Computation*, 155:134–171, Dec. 1999.

[3] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types,. Tech. Report MS-CIS-05-26, University of Pennsylvania, July 2004.

[4] F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 232–244, Charleston, South Carolina, Jan. 2006.

[5] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *Proc. International Conference on Functional Programming*, 2009.

[6] T. Sheard and N. Linger. Programming in Omega. In *2nd Central European Functional Programming School*, volume 5161 of *Springer LNCS*, pages 158–227, 2007.