Tracing ambiguity in GADT type inference

Jacques Garrigue* Didier Rémy†

1 Abstract

GADTs, short for *Generalized Algebraic DataTypes*, extend usual algebraic datatypes with a form of dependent typing that has many useful applications, but raises serious issues for type inference. Pattern matching on GADTs introduces type equalities with limited scopes, which are a source of ambiguities that may destroy principal types—and must be resolved by type annotations.

By tracing ambiguities in types, we may tighten the definition of ambiguities and confine them, so as to request fewer type annotations. Now in use in OCaml 4.00, this solution also lifts some restriction on object types and polymorphic types that appeared in a previous implementation of GADTs in OCaml.

2 Introduction

GADTs, short for *Generalized Algebraic DataTypes*, extend usual algebraic datatypes with a form of dependent typing by enable type refinements in pattern-matching branches. They can express many useful invariants of data-structures, provide safer typing, and allow for more polymorphism. They have already been available in some Haskell implementations (in particular GHC) for many years and now appear as a natural addition to strongly typed functional programming languages.

However, this is by no means trivial. In their presence, full type inference is in general undecidable, even in the restricted setting of ML-style polymorphism. Moreover, many well-typed programs lack a most general type. A solution to both problems is to require explicit type annotations. Unfortunately, while it is relatively easy to design a sound typing algorithm for a language with GADTs, it is surprisingly difficult to keep principal types without requesting full type annotations on every case analysis.

While GHC 7 does not exactly require full *annotations*, it follows a similar strategy, called *OutsideIn* [3], which requires full *type information* to be inferred from the external context, for every pattern-matching.

Surprisingly, GADTs may not play well with other features of the language: in our first implementation of GADT in OCaml [1], we had to restrict the use of object types and polymorphic variants in combination with GADTs, to prevent local equations from breaking the invariant that the same row variable may only appear in two record types that are equal.

From those limitations, we realized that *ambiguity* is actually central to the question of GADT type inference and could be considered independently of principality. Local type equations introduced inside a pattern-matching branch are the source of ambiguities: they allow implicit type conversions, *i.e.* several inter-convertible forms for types that are irrelevant in the scope of the equation, but become nonconvertible—hence ambiguous—when leaving the branch as the equation can no longer be used.

Detection and rejection of ambiguous programs is a preliminary to type inference. Our definition of non-ambiguity allows us to restrict the set of valid typings. We conjecture that among the valid typings there is always a principal one (*i.e.* subsuming all of them), which our inference algorithm finds. Besides, this restriction ensures that types related by local equations do not leak outside of their scopes, recovering the internal invariants needed by OCaml.

3 Refining ambiguity

The informal definition of ambiguity is actually so general that it may just encompass too many cases. Consider the following program.

```
type _ t = Int : int t

let f (type a) (x : a t) =
  match x with Int -> 1
```

Type `t` is a GADT with one index parameter (denoted by the single underscore), and a single case `Int`, for which the index is the type `int`.

In the definition of `f`, we first introduce an explicit universal variable `a`, called a *skolem* variable, treated in a special way in OCaml as it can be refined by GADT pattern matching. By constraining the type of `x` to be `a t`, we are able to refine `a` when pattern-matching `x` against the constructor `Int`: the equation $\mathtt{a} = \mathtt{int}$ becomes available in the corresponding branch, that is while typechecking the expression `1`, which can be assigned either type `a` or `int`. As a result, `f` can be given either type $\alpha\ \mathtt{t} \to \mathtt{int}$ or $\alpha\ \mathtt{t} \to \alpha$. This seems to fulfill the definition of ambiguity, and it should be rejected.

But should we really reject it? Consider the following slight variations in the definition of `f`:

```
let f' (type a) (x : a t) =
  match x with Int -> true

let g (type a) (x : a t) (y : a) =
  match x with Int -> (y > 0)
```

In `f'`, we just return `true`, which has the type `bool`, unrelated to the equation. In `g`, we actually use the equation to

*Nagoya University, Graduate School of Mathematics
†INRIA, Rocquencourt

turn y into an int but eventually return a boolean. Clearly both cases are non-ambiguous. But how do they differ from the original f? The only reason we have deemed f to be ambiguous is that 1 could potentially have type a by using the equation. However, nothing forces us to use this equation, and if we do not use it the only possible type is int. It looks even more innocuous than g, where we need indirectly the equation to infer the type of the body.

So, what would be a really ambiguous type? We obtain one by mixing a's and int's in the returned value.

```
let g' (type a) (x : a t) (y : a) =
  match x with Int -> if y > 0 then y else 0
```

Here, the then branch has type a while the else branch has type int, so choosing either one would be ambiguous.

How can we capture this refined notion of ambiguity? The idea is to track whether such mixed types are escaping from their scope. An intuitive way to see whether this is the case is not to allow the expression to have either type but force it to have the ambiguous type a or int, *i.e.* in a way, the intersection type $\mathtt{a} \wedge \mathtt{int}$, which we may just represent as the set of types $\{\mathtt{a}, \mathtt{int}\}$.

A set of types must still be *coherent*, *i.e.* all the types it contains must be equivalent under the equations available in the current scope. However, a set of types may suddenly become incoherent when leaving a scope, which is what we call an *ambiguity*.

We allow type annotations in the source program as a way to avoid ambiguities. Intuitively, in an expression $(e : \tau)$, both the inner e and the outer $(e : \tau)$ have sets of types T_1 and T_2 that may differ, but such that τ be included in both. This way, ambiguity doesn't leak in either direction.

```
let g' (type a) (x : a t) y =
  match x with Int ->
    (if (y : a) > 0 then (y : a) else 0 : a)
```

By adding type annotations on y and on the conditional, both variable y and conditional may be given unique types, which are unambiguous when leaving the scope of the equation. That is, (y : a) and 0 can be assigned the type $\{\mathtt{a}, \mathtt{int}\}$, which is also that of the conditional if ... else 0, while the annotation (if ... else 0 : a) and variable y both have the singleton type $\{\mathtt{a}\}$.

Of course, it is too verbose to write annotations everywhere, so we let annotations on parameters propagate to their uses and annotations on results propagate inside pattern-matching branches. The function g' may just be written:

```
let g' (type a) (x : a t) (y : a) : a =
  match x with Int -> if y > 0 then y else 0
```

or, using the syntax for explicitly polymorphic types:

```
let g' : type a. a t -> a -> a = fun x y ->
  match x with Int -> if y > 0 then y else 0
```

4 Discussion

A natural question at this point is why not just require that the type of the result of pattern-matching a GADT be fully known from annotations? This would avoid the need for this new notion of ambiguity. This is perhaps good enough if we only consider small functions: as shown for g', we may write the function type in one piece (as in either one of the last two versions) and still get the full type information.

However, the situation degrades when using local let bindings. For example, consider the function h below:

```
let h : type a. a t -> int = fun x ->
  let y = match x with Int -> 1 in y*2
```

The return type int only applies to y*2, so we cannot propagate it automatically as an annotation for the definition of y. Basically, one would have to explicitly annotate all let bindings whose definition uses pattern-matching on GADTs. This may easily become a burden, especially when the type is completely unrelated to the GADTs (or accidentally related as in the definition of f, above).

GHC 7 improves on this in using constraint solving in place of directional annotation propagation. This greatly reduces the need for annotations, and it even accepts some programs that we would deem ambiguous.

```
let k (type a) (x : a t) (y : a) =
  let z = match x with Int -> if y>0 then y else 0
  in z + 1
```

However, constraint solving requires a completely new inference engine. Moreover, it comes together with a non-generalizing typing rule for let, whose impact on ML programs appears to be much bigger than on Haskell programs. Finally, it appears that GHC applies a relaxed version of OutsideIn, where some arbitrary choices are allowed at toplevel. This seems to mean that a strict application of OutsideIn, as required for principality, was deemed too constraining.

Another interesting approach to type inference for GADTS [2] is to use several sophisticated passes that propagate local typing constraints (and not just type annotations) progressively to the rest of the program. In practice, the amount of type annotations required on source programs is roughly comparable with the OutsideIn strategy, although both techniques are quite different and hard to compare.

We believe that our notion of ambiguity is simple enough to be understood easily by users, avoids an important number of seemingly redundant type annotations, and provides an interesting alternative to a strict OutsideIn approach.

References

[1] J. Garrigue and J. L. Normand. Adding GADTs to OCaml: the direct approach. Presented at the Workshop on ML, Sept. 2011.

[2] F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL'06)*, pages 232–244, Charleston, South Carolina, Jan. 2006.

[3] D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. OutsideIn(X) Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5):333–412, Sept. 2011.

A A brief technical overview

The key to our solution is tracing ambiguities during type inference. Although we intuitively infer sets of types rather than types, as suggested above, this is not quite sufficient, because we sometimes need to keep ambiguous types synchronized.

For example, consider the application of a function `choice` of type $\alpha \to \alpha \to \alpha$ to a value v of ambiguous type $\{\texttt{int}, \texttt{a}\}$. Should its type be $\{\texttt{int} \to \texttt{int}, \texttt{a} \to \texttt{a}\}$ obtained by separately considering the application of the function to each possible type for the argument, or $\{\texttt{a}, \texttt{int}\} \to \{\texttt{a}, \texttt{int}\}$ obtained by just instantiating α with the ambiguous type $\{\texttt{a}, \texttt{int}\}$? In fact, both types carry complementary information: the former says that the domain and the codomain are really the same, while the later says that the function itself is non-ambiguous, just sending values of an ambiguous type into another ambiguous type.

Interestingly, we can represent both information simultaneously, in the more precise notation $\Sigma(\alpha \in \{\texttt{a}, \texttt{int}\})\ \alpha \to \alpha$ that means "$\alpha \to \alpha$ for some α in $\{\texttt{a}, \texttt{int}\}$": we recover the later by replacing α with $\{\texttt{a}, \texttt{int}\}$ while we generate the former by successively replacing α with each element of $\{\texttt{a}, \texttt{int}\}$. Moreover, this allows to distinguish between types $\Sigma(\alpha \in \{\texttt{a}, \texttt{int}\})\ \alpha \to \alpha$ and $\Sigma(\alpha \in \{\texttt{a}, \texttt{int}\})\ \Sigma(\beta \in \{\texttt{a}, \texttt{int}\})\ \alpha \to \beta$, where ambiguities are correlated between the domain and the codomain in the former case, but not in the later case. This is a useful technical difference for principality of type inference—but we may ignore it for the moment.

In fact, it is often easier to manipulate types rather than type schemes. This is possible by leaving the assumption $\alpha \in \{\texttt{a}, \texttt{int}\}$ in the typing context, using auxiliary type variables as an indirection to ambiguous types.

The example given above can be typed in Γ_0 by introducing a variable α to refer to the ambiguous type $\{\texttt{a}, \texttt{int}\}$. Let us write Γ for $\Gamma_0, \alpha \in \{\texttt{a}, \texttt{int}\}$. We have both $\Gamma \vdash \texttt{choice} : \alpha \to \alpha \to \alpha$, since `f` is polymorphic and can be applied to an ambiguous type as well, and $\Gamma \vdash v : \alpha$; then, we conclude $\Gamma \vdash \texttt{choice}\ v : \alpha \to \alpha$ using the normal typing rule for application. Finally, we may discharge the assumption and say $\Gamma_0 \vdash \texttt{choice}\ v : \Sigma(\alpha \in \{\texttt{a}, \texttt{int}\})\ \alpha \to \alpha$, so that this expression may be let-bound and reused later in some other context.

Abstracting ambiguous types as type variables has the other advantage that type inference with ambiguities becomes a straightforward adaptation of ML-style type inference. It suffices to modify the unification algorithm so that it accepts equivalence classes with ambiguous types as solved forms, while a standard implementation of unification would reject them as clashes.

More precisely, unification problems can be represented using conjunctions of multi-equations (and existential quantifiers to deal with introduction and renaming of type variables). A multi-equation is just a multiset of types that should be made equal. The standard unification algorithm proceeds by rewriting such problems into solved forms, *i.e.* such that a type variable appears in at most one multi-equation and each multi-equation is itself in solved form.

In the absence of GADTs, a multi-equation is solved when it contains at most one non variable type. Indeed, if a multi-equation contains two types $\tau_1\ \texttt{t}_1$ and $\tau_2\ \texttt{t}_2$, either the top constructors $\texttt{t}_1$ and $\texttt{t}_2$ are equal and the multi-equation can be decomposed, or they are different and the unification problem is unsolvable.

When keeping track of ambiguities in the presence of GADTs, we introduce skolem variables, which lie between unification variables and type constructors. A skolem variable `a` stands for explicit polymorphism: it is, by default, a rigid variable and, as type constructors, it clashes with any other skolem variable or type constructor. However, when pattern matching a GADT, a skolem variable may also be refined by some equation of the form $\texttt{a} = \tau$, which behaves as a type abbreviation: the equation is ignored until the skolem variable `a` clashes with a type constructor or another skolem variable; then, the equation is used as a "joker" and τ is added to the multi-equation involved in the conflict with `a`; the conflict is then ignored as long as the equation remains in scope. Thus, solved multi-equations may now also contain one or several refined skolem variables (whose jokers have been used).

Of course, an equation introduced in the branch of some pattern matching is removed when leaving the branch: a skolem variable `a` that was refined in this branch looses its magical power and recovers its default rigid status that makes it clash with other type constructors or unrefined skolem variables. Only accessible multi-equations have to be checked for clashes: hopefully, multi-equations that contain the skolem variable `a` haven't used their joker (and `a` does not clash) or became inaccessible and "garbage collected"; otherwise, a clash occurs and unification fails. (This is reported in OCaml as out of scope use of an equation.)

For example, consider the definition of `f` in §3. The pattern `Int` introduces the equation $\texttt{a} = \texttt{int}$. The branch `1` can be typed with α where $\alpha \approx \texttt{int}$. The type `a` is not involved and the equation can be safely removed when leaving the branch.

By contrast, in the branch expression `if y > 0 then y else 0` of the definition of `g'` we first type `y` with α where $\alpha \approx a$ and `0` with β where $\beta \approx \texttt{int}$, then add the equation $\alpha \approx \beta$ to get the type of the conditional (we ignore typechecking of the condition). All three equations are merged into $\alpha \approx \beta \approx \texttt{a} \approx \texttt{int}$. Without GADTs, $\texttt{a} \approx \texttt{int}$ would clash and typechecking would fail. Here, since `a` has an associated equation $\texttt{a} = \texttt{int}$, we replace `a` by its expansion (in fact, we keep it, but add parentheses to mark it is no longer conflicting): $\alpha \approx \beta\ (\approx \texttt{a}) \approx \texttt{int} \approx \texttt{int}$, which may be decomposed into the solved form $\alpha \approx \beta\ (\approx \texttt{a}) \approx \texttt{int}$. When leaving the branch, the equation is removed and `a` recovers its normal status; as a result this multi-equation becomes $\alpha \approx \beta \approx \texttt{a} \approx \texttt{int}$, which is conflicting. Since it determines the type of the branch, it is still accessible and becomes the source of an ambiguity. The program fails.

Ambiguity may be avoided if the whole branch is annotated (with either `a` or `int`) so that the ambiguity is confined to the inner scope; alternatively, one could annotate one or the other side of the conditional, *i.e.* `(1:a)` or `(y:int)` so that the conditional itself is non ambiguous.