GADTs Are Not (Even Lax) Functors

Pierre Cagne
Appalachian State University

Abstract

We expose a fundamental incompatibility between GADTs' function-mapping operations and function composition. To do this, we consider sound and complete semantics of GADTs in a series of natural categorical settings. We first consider interpreting GADTs naively as functors on categories interpreting types. But GADTs' function-mapping operations are not total in general, so we consider semantics in categories that allow partial morphisms. Unfortunately, however, GADTs' function-mapping operations do not preserve composition of partial morphisms. This compels us to relax functors' composition-preserving property, and thus to interpret GADTs as lax functors. This exposes yet another behavior of GADTs that is not computationally reasonable: if G interprets a GADT, then $G(f)$ can be defined on more elements than $G(g)$ even though f is defined on fewer elements than g. This is joint work with Patricia Johann.

In this talk, I will present several obstructions to the generalization of the usual Initial Algebra Semantics (IAS) of Algebraic Data Types (ADTs) to their syntactic generalizations known as Generalized Algebraic Data Types (GADTs). ADTs — such are lists, trees, graphs, etc. — are ubiquitous in modern programming. They encode, for example, data structures that are critical in reducing the complexity of algorithms that process them. Moreover, they are safe and easy to reason with precisely because of the theoretical understanding that IAS affords them. Pattern-matching, folds, mapping, etc., are all justified by the fact that IAS is sound and complete for ADTs. For this reason, we would like to construct an IAS for GADTs in such a way that it specializes to the standard IAS for ADTs [MA86]. Although GADTs are extensively used in practice in, say, Haskell and OCaml, they are still missing a sound and complete semantics.

The obstructions to IAS for GADTs can be illustrated using the *equality* defined by

```
data Eq :: * → * → * where
  Refl :: ∀ α. Eq α α
```

According to the above Haskell syntax, `Eq` is a binary GADT with a unique constructor `Refl` inhabiting each instance of the form `Eq` α α. Although it seems simple at first glance, `Eq` support the definition of the following transport function, which will be important below:

```
  trp :: ∀ α β. Eq α β → (α → β)
  trp Refl x = x
```

Naive IAS for GADTs. Our first attempt at an IAS for GADTs is by direct extension. Recall that the standard IAS of ADTs associates to each n-ary ADT `A` a functor $A : \mathcal{C}^n \to \mathcal{C}$ where $\mathcal{C}$ is a category interpreting closed types. Moreover, this functor A is obtained as the (carrier of the) initial algebra of an endofunctor on the functor category $\mathrm{Func}\,(\mathcal{C}^n, \mathcal{C})$ derived from the body of the definition of the ADT `A`; taking $\mathcal{C}$ to be the category Set of sets and functions is usually enough to carry out most reasoning on ADTs. The completeness of IAS for ADTs implies in particular that, for every closed type τ with interpretation X, $A(X)$ is (in natural bijection with) the set of normal forms of closed terms of type `A` τ. The same property cannot be obtained for GADTs. Indeed, suppose that there is a functor $E : \mathrm{Set}^2 \to \mathrm{Set}$ interpreting `Eq`. If we want the interpretation to be sound, then for each type τ with interpretation X we need an element $r_X \in E(X, X)$ interpreting `Refl :: E` τ τ. Moreover, soundness and completeness imply that the interpretation of the unit type must be a singleton set 1, so for any function $x : 1 \to X$, there must be an element $E(x, \mathrm{id}_1)(r_1) \in E(X, 1)$. We can prove that, for any types τ_1 and τ_2 with interpretations X_1 and X_2, respectively, the interpretation of the function `trp` sends elements of $E(X_1, X_2)$ to bijections $X_1 \simeq X_2$. Thus, whenever the type τ has at least one element, the element found in $E(X, 1)$ forces $X \simeq 1$. Clearly, this semantics fails to be complete. As shown in [CJ23], this argument is not specific to the category Set and can be replayed in any category $\mathcal{C}$ with the enough structure to express the usual IAS of ADTs.

Allowing for partial functions. The failure of the first attempt is instructive: the core issue there is that functions of the form $E(x, \mathrm{id}_1)$ must send the element r_1 to elements in $E(X, 1)$ even when completeness would require $E(X, 1)$ to be empty. In other words, the totality of functions of the form $E(x, \mathrm{id}_1)$ is problematic. We can try to overcome this problem by allowing such functions to be partial, but we need a notion of partiality that is coherent with computations. In our view, the core feature of such a notion is that computations cannot recover from failure. We therefore propose the following definition:

Definition 1. A *structure of computational partiality* on a category $\mathcal{C}$ is a wide subcategory of $\mathcal{C}$ whose complement is a cosieve.

A *wide subcategory* of $\mathcal{C}$ is a subcategory of $\mathcal{C}$ that contains all objects of $\mathcal{C}$, and a *cosieve* on $\mathcal{C}$ is a subclass S of the morphisms of $\mathcal{C}$ such that for all morphisms $f : A \to B$ and $g : B \to C$ in $\mathcal{C}$, if $f \in S$ then $gf \in S$. The category PSet of sets and partial functions between them is an example of such a category when we take the structure of computational partiality to be Set seen as a subcategory of PSet. For this reason, we often call the morphisms of a structure of computational partiality *total*, and the morphisms in the cosieve that is its complement *partial*.

Given a category $\mathcal{C}$ with a structure of computational partiality $\mathcal{D}$ on it, we aim to repair the naive attempt above by interpreting the ambient language in $\mathcal{D}$, except for the GADTs, which will be interpreted as functors from $\mathcal{C}^n$ to $\mathcal{C}$ rather than from $\mathcal{D}^n$ to $\mathcal{D}$. Informally, we interpret everything definable in syntax in $\mathcal{D}$, but we allow the function-mapping operations of GADTs to send total morphisms to partial morphisms. In particular, the unit type must be interpreted as a terminal object 1 in $\mathcal{D}$. Now the functor $E : \mathcal{C}^2 \to \mathcal{C}$ interpreting `Eq` is free, a priori, to give a partial morphism $E(x, \mathrm{id}_1)$ even when $x : 1 \to X$ is a total morphism. However, such a morphism $x : 1 \to X$ is always a section, with its retraction being the unique total morphism from X to 1 given by the fact that 1 is terminal in $\mathcal{D}$. Being a functor, $E(\_, 1)$ must send sections to sections, so that $E(x, \mathrm{id}_1)$ is a section as well. Now our hope to fix the issue is dashed because sections in $\mathcal{C}$ are always total: if $s : X \to Y$ were a partial section, then, for any retraction r of s, $\mathrm{id}_X = rs$ would need to be partial as well. But identities are total by definition. We can then replay the argument of the naive attempt: being total, $E(x, \mathrm{id}_1)$ must send the (now global) element r_1 of $E(1, 1)$ to a (now global) element of $E(X, 1)$, which again forces $X \simeq 1$. This shows, as before, that the interpretation of any type τ with at least one element is the object 1, making the interpretation highly non-complete.

Relaxing GADTs' functorial behavior. Allowing GADTs' functorial behavior to target partial morphisms failed because classes of morphisms defined by equations, such as the class of sections, are necessarily preserved by functors. We therefore can't expect GADTs' interpretations to respect composition on the nose. However, if we can map a function `f` over an element of a GADT, and if we can also map a function `g` over that result, then we should obtain the same result by mapping `g . f` in one go over the original element. That is, mapping a composition should be the same as mapping the components of the composition one after the other provided all these operations are well-defined. This suggests that $\mathcal{C}$ should include an order $\leq$ on each hom-set, where $f \leq g$ is read informally as "g is an extension of f to a bigger domain of definition". We therefore require that $\mathcal{C}$ is enriched over the category Pos of posets and monotonic functions between them. We intend to interpret GADTs as normal lax functors, rather than simple functors, into $\mathcal{C}$. A*normal lax functor* $G : \mathcal{B} \to \mathcal{C}$ between Pos-enriched categories is defined in the same way as an enriched functor, except that we require $G(g)G(f) \leq G(gf)$ instead of $G(g)G(f) = G(gf)$ for all composable morphisms f and g of $\mathcal{B}$. Interpreting GADTs as normal lax functors resolves the issue raised by sections in the previous paragraph: if s is a section with retraction r, and G is a normal lax functor, then $G(r)G(s)$ only has to be less than or equal to $G(rs) = G(\mathrm{id}) = \mathrm{id}$, and so $G(s)$ must no longer be a section and can be non-total.

Now consider the Pos-enriched category PSet, where $f \leq g$ holds for $f, g : X \to Y$ if and only if, for all $x \in X$, f defined on x implies $g(x) = f(x)$. A sound and complete interpretation of the type `Bool` of booleans in PSet must be a 2-element set, say $B = \{\bot, \top\}$, where $\bot$ interprets `false` and $\top$ interprets `true`. It must also take product types must take sets X and Y to the cartesian product $X \times Y$. Now, consider the partial function $f : B \times B \to B \times B$ defined only on the pairs of the form $(\bot, y)$, and given by $f(\bot, y) = (\bot, y)$. Then $f \leq g$, where g is total and defined by $g(x, y) = (x, x \vee y)$. The monotonicity of the normal lax functor E interpreting `Eq` implies $E(f, f) \leq E(g, f)$. In particular, $E(g, f)$ is defined on any element of $E(B \times B, B \times B)$ on which $E(f, f)$ is defined, and agrees with $E(f, f)$ on it. Thus, $E(g, f)(r_{B\times B}) = E(f, f)(r_{B\times B}) = r_{B\times B}$. According to the algorithm given in [JC22], this is a contradiction: only the pairs of function of the form (h, h) for a given $h : B \times B \to B \times B$ can be mapped over $r_{B\times B}$.

It might be difficult to spot the source of the problem with `Eq` because it is so degenerate, but the above example is important because `Eq` is quintessential in the theory of GADTs. Nevertheless, the issue deriving from $E(f, f) \leq E(g, f)$ is perhaps better illustrated by a properly recursive GADT that uses `Eq`, such as

```
data Seq :: * → * where
  inj :: ∀ α. α → Seq α
  pair :: ∀ α β γ. Seq α → Seq β → Eq γ (α,β) → Seq γ
```

Write $S : \mathrm{PSet} \to \mathrm{PSet}$ for the normal lax functor interpreting `Seq`, $i_B : B \to S(B)$ for the interpretation of `inj` instantiated at `Bool`, and $p_B : S(B) \times S(B) \times E(B \times B, B \times B) \to S(B \times B)$ for the interpretation of `pair` instantiated at `Bool`, `Bool`, and `(Bool,Bool)`. It should be intuitively clear that the partial function f can be mapped over (the interpretation of) `pair (inj false) (inj true) Refl :: S (Bool,Bool)`. The formal justification, given in [JC22], is that f can be written as $(x, y) \mapsto (f_1(x), f_2(y))$ for the partial functions $f_1, f_2 : B \to B$, where f_1 is defined only on $\bot$ with value $\bot$ and f_2 is id_B. To perform this mapping operation, we simply strip the constructors, apply f_1 and f_2 to the relevant data, and reapply the constructors. The result is the element we started with, namely, $S(f)(p_B(i_B(\bot), i_B(\top), r_{B\times B})) = p_B(i_B(\bot), i_B(\top), r_{B\times B})$. Since $f \leq g$, since $S(g)$ is defined wherever $S(f)$ is defined, and since $S(g)$ agrees with $S(f)$ there, $S(g)$ is also defined on $p_B(i_B(\bot), i_B(\top), r_{B\times B})$ with value $p_B(i_B(\bot), i_B(\top), r_{B\times B})$. This implies that g is also of the form $(x, y) \mapsto (g_1(x), g_2(y))$ for some (necessarily total) functions $g_1, g_2 : B \to B$. But we easily check that it is not: $g(\bot, \bot) = (\bot, \bot)$ implies that $g_2(\bot) = \bot$ and $g(\top, \bot) = (\top, \top)$ implies that $g_2(\bot) = \top$.

References

[CJ23] Pierre Cagne and Patricia Johann. Are GADTS really data structures? Submitted, 2023.

[JC22] Patricia Johann and Pierre Cagne. Characterizing Functions Mappable over GADTs. In Ilya Sergey, editor, *Programming Languages and Systems*. Springer Nature Switzerland, 2022.

[MA86] Ernest G. Manes and Michael A. Arbib. *Algebraic Approaches to Program Semantics*. Monographs in Computer Science. Springer New York, 1986.