Mathematical Structures in Computer Science (2024), 1–24
doi:10.1017/S0960129524000161

PAPER

GADTs are not (Even partial) functors

Pierre Cagne[1], Enrico Ghiorzi[2] and Patricia Johann[1]

[1]Department of Computer Science, Appalachian State University, Boone, NC, USA, Email: cagnep@appstate.edu and
[2]Università degli Studi di Genova, Genova, Italy, Email: enrico.ghiorzi@edu.unige.it
Corresponding author: Patricia Johann; Email: johannp@appstate.edu

(Received 5 June 2023; revised 22 January 2024; accepted 21 April 2024)

Abstract
Generalized Algebraic Data Types (GADTs) are a syntactic generalization of the usual algebraic data types (ADTs), such as lists, trees, etc. ADTs' standard initial algebra semantics (IAS) in the category *Set* of sets justify critical syntactic constructs – such as recursion, pattern matching, and fold – for programming with them. In this paper, we show that semantics for GADTs that specialize to the IAS for ADTs are necessarily unsatisfactory. First, we show that the functorial nature of such semantics for GADTs in *Set* introduces *ghost* elements, i.e., elements not writable in syntax. Next, we show how such ghost elements break parametricity. We observe that the situation for GADTs contrasts dramatically with that for ADTs, whose IAS coincides with the parametric model constructed via their Church encodings in System F. Our analysis reveals that the fundamental obstacle to giving a functorial IAS for GADTs is the inherently partial nature of their map functions. We show that this obstacle cannot be overcome by replacing *Set* with other categories that account for this partiality.

Keywords: GADTs; functorial semantics; (higher-order) initial algebra semantics; partial functions

1. Introduction

As functional languages have become increasingly sophisticated, so too have the data types they support. Algebraic data types (ADTs) – i.e., data types that are essentially tree types – are well-known to support an initial algebra semantics (IAS) in any category with enough structure (Manes and Arbib 1986). The IAS interpretation of an ADT comprises exactly the interpretations of the terms that are writable in its syntax. IAS interpretations of ADTs provide semantic justification for useful computational tools for programming with them, such as pattern matching, recursion rules, induction rules, etc. Also fundamental is that the interpretation of the type constructor defined by an ADT can be extended to a functor whose action on morphisms interprets the ADT's syntactic map function. Categorical models in which ADTs have IAS interpretations include those of (Johann and Ghani 2007; Johann et al. 2021; Manes and Arbib 1986). In the model of (Johann et al. 2021), the syntactic generalization of ADTs known as nested types (Bird and Meertens 1998) also has IAS interpretations (Manes and Arbib 1986; nLab authors 2019). Examples of nested types include perfect trees and bushes; see Section 3 below.

In addition to ADTs and nested types, modern functional languages such as Haskell, Agda, and OCaml support *generalized algebraic data types* (GADTs) (Peyton Jones et al. 2006). As their name suggests, GADTs syntactically generalize nested types (and thus further generalize ADTs), so that

(1)

$$\begin{tikzcd} \boxed{\text{ADTs}} \arrow[r, hook, "\text{syntactically}", "\text{generalized by}"'] & \boxed{\text{nested types}} \arrow[r, hook, "\text{syntactically}", "\text{generalized by}"'] & \boxed{\text{GADTs}} \end{tikzcd}$$

This sequence of syntactic generalizations suggests that a corresponding sequence of semantic generalizations might also hold. Specifically, since nested types (and thus ADTs) have IAS interpretations as functors on the category *Set* of sets and functions between them, we might expect that there is also an IAS interpretation of GADTs as functors on *Set*. Further, we might expect that this IAS of GADTs specializes to the standard IAS for nested types. Then, this interpretation of GADTs, like the standard IAS of nested types, would specialize to the standard IAS of ADTs. A semantic analogue of (1), if it existed, would formally justify the use of the word "generalized" in the moniker "generalized algebraic data type."

In this paper, we show that, unfortunately, there is no semantic analogue of (1) because there is no reasonable IAS interpretation of GADTs as functors on *Set*. A GADT programmer is likely to use GADTs precisely *because* they exhibit different behaviors at different types. They are therefore likely to consider a GADT to be completely specified by its syntax, i.e., to contain no data elements other than those writable in its syntax. But when GADTs are thought of in this way, the shape of a particular element of a GADT is actually *determined* by the data it contains. This is in marked contrast to the situation for ADTs and nested types, whose type uniformity gives rise to IAS interpretations that allow them to be regarded as type-independent containers that can be filled with data of any type. The main result of the first part of this paper shows that the type non-uniformity of GADTs entails that they cannot have IAS interpretations as functors on *Set* without violating properties that we expect of models of languages that support them. In particular, Example 8 of Section 5 shows that a language supporting GADTs cannot have parametric models (Reynolds 1983) in which GADTs are interpreted as functors on *Set*, at least not if those interpretations are to coincide with the standard IAS for those GADTs that are ADTs and nested types.[1] This finding is unintuitive, novel, and surprising.

The fact that GADTs do not have such specializable IAS interpretations as functors on *Set* prompts us to consider interpreting GADTs as functors on other categories. Observing that mapping functions over elements of GADTs is notorious for being only partially defined (Johann and Cagne 2022; Johann and Ghani 2008; Johann and Polonsky 2019) compels us to seek IAS interpretations of GADTs as functors on categories with inherent notions of partiality. But since the IAS interpretation of a GADT should support pattern matching, recursion rules, induction rules, and other IAS-derived programming tools for that GADT, it should comprise exactly the interpretations of the terms that are writable in the GADT's syntax. And since the IAS interpretation of a GADT should coincide with the standard IAS interpretation whenever the GADT is actually a nested type or an ADT, the semantic treatment of partial functions must specialize to the standard semantic treatment of everywhere-defined functions. The latter entails that we take our categories with inherent notions of partiality to have as subcategories the usual categories supporting IAS for nested types (including ADTs).

A concrete example of such a category is the category *PSet* of sets and partial functions between them. Indeed, this category has *Set* as a subcategory, and its objects are precisely those of *Set*. Drawing inspiration from *PSet*, in Section 6 we use the considerations of the previous paragraph to identify other categories that similarly capture computationally relevant notions of partiality. We consider the main feature of computationally relevant partiality to be that functions propagate undefinedness. This behavior is akin to that of functions that are strict in the sense of (Burn et al. 1986) and explicitly requiring it captures the computational expectation that a function cannot produce a value when given as an input an undefined value produced by a(nother) function. The main result of the second part of this paper is Theorem 18, which considers IAS interpretations of GADTs in categories equipped with computationally relevant notions of partiality. It shows that, in any such category, any extension to GADTs of the standard IAS for nested types in which the interpretations of the type constructors defined by GADTs extend to functors must be trivial.

Taken together, the two main results of this paper lead us to conclude that GADTs cannot be interpreted as functors in any reasonable computational setting. So if a functorial semantics like the one we seek is possible, then it will be necessarily in a setting much more exotic than the ones GADT programmers are accustomed to using to understand their programs.

This paper is an extended version of the LSFA 2021 paper (Johann et al. 2021). The results of Sections 2 through 5 were first reported by (Johann et al. 2021), although they have been reworked here. The work in Section 6 is entirely new.

2. Syntax and Semantics of ADTs

A (unary) ADT has the form[2]

$$\mathsf{T\ a} = \mathsf{C_1 t_{11} \ldots t_{1k_1}} \mid \ldots \mid \mathsf{C_n t_{n1} \ldots t_{nk_n}}$$

where each $\mathsf{t_{ij}}$ is a type depending only on a. Such a data type can be thought of as a "container" for data of type a. The data in an ADT are arranged at various *positions* in its underlying *shape*, which is determined by the types of its *constructors* $\mathsf{C_1, \ldots, C_n}$. An ADT's constructors are used to build the data values of the data type, as well as to analyze those values using *pattern matching*. ADTs are used extensively in functional programming to structure computations, to express invariants of the data over which computations are defined, and to ensure the type safety of programs specifying those computations.

List types are the quintessential ADTs. The shape of the container underlying the type

$$\mathsf{List\ a} = \mathsf{Nil} \mid \mathsf{Cons\ a\ (List\ a)}$$

is determined by the types of its two constructors $\mathsf{Nil} :: \mathsf{List\ a}$ and $\mathsf{Cons} :: \mathsf{a} \to \mathsf{List\ a} \to \mathsf{List\ a}$. These constructors specify that the data in a list of type $\mathsf{List\ a}$ are arranged linearly. The shape underlying the type $\mathsf{List\ a}$ is therefore given by the set $\mathbb{N}$ of natural numbers, with each natural number representing a choice of length for a list structure, and the positions in a structure of shape n given by natural numbers ranging from 0 to $\mathsf{n}-1$. Since the type argument to every occurrence of the type constructor List in the right-hand side of the above definition is the same as the type instance being defined on its left-hand side, the type $\mathsf{List\ a}$ enforces the invariant that all of the data in a structure of this type have the same type a. In a similar way, the tree type

$$\mathsf{Tree\ a} = \mathsf{Leaf\ a} \mid \mathsf{Node\ (Tree\ a)\ a\ (Tree\ a)}$$

of binary trees has as its underlying shape the type of binary trees of units, and the positions in a structure of this type are given by sequences of L (for "left") and R (for "right") navigating a path through the structure. The type $\mathsf{Tree\ a}$ enforces the invariant that all of the data at the nodes and leaves in a structure of this type have the same type a.

Since the shape of an ADT structure – i.e., a structure whose type is an instance of an ADT – is independent of the type of data it contains, ADTs can be defined polymorphically. As a result, an ADT structure containing data of type a can be transformed into another ADT structure of the exact same shape containing data of another type b simply by applying a given function $\mathsf{f} :: \mathsf{a} \to \mathsf{b}$ to each of its elements. In other words, for every ADT $\mathsf{T\ a}$, its underlying type constructor T can be made an instance of Haskell's $\mathsf{Functor}$ class by defining a type-and-data-uniform, structure-preserving, data-changing function $\mathsf{map_T}$ for it.[3] Then, given a type-independent way of rearranging an ADT structure's shape $\mathsf{T\ a}$ into the shape for another ADT structure $\mathsf{T'\ a}$, we get the same structure of type $\mathsf{T'\ b}$ regardless of whether we first rearrange the original structure of type $\mathsf{T\ a}$ into one of type $\mathsf{T'\ a}$ and then use $\mathsf{map_{T'}}$ to convert that resulting structure to one of type $\mathsf{T'\ b}$, or we first use $\mathsf{map_T}$ to convert the original structure of type $\mathsf{T\ a}$ to one of type $\mathsf{T\ b}$ and then rearrange that resulting structure into one of type $\mathsf{T'\ b}$. For example, if $\mathsf{f} :: \mathsf{a} \to \mathsf{b}$, $\mathsf{t} :: \mathsf{Tree\ a}$, and

$\mathsf{g} :: \mathsf{Tree\ c} \to \mathsf{List\ c}$ is a polymorphic function (note that g's type is implicitly universally quantified over c) that arranges trees into lists in a type-independent way, then we have the following rearrange-transform property:

$$\mathsf{map}_{\mathsf{List}}\ \mathsf{f}\ (\mathsf{g}\ \mathsf{t}) = \mathsf{g}\ (\mathsf{map}_{\mathsf{Tree}}\ \mathsf{f}\ \mathsf{t})$$

2.1 ADTs as functors

The standard way to understand ADTs is at least fixpoints[4] of (first-order) functors on the category interpreting types. This category is typically taken to be the category *Set*, whose objects are sets and whose morphisms are functions between them, and we will do so here unless otherwise specified. But, as shown in (Johann and Polonsky 2019), we can interpret types as objects in any locally presentable category (Adámek and Rosický 1994) without affecting the development below.

Syntactically, ADTs can be represented as fixpoints in any language with primitives for sum types, product types, and recursion. If μ is the fixpoint operator in such a language, and if $\mathsf{1}$ is its unit type, then the fixpoint representations of the ADTs $\mathsf{List\ a}$ and $\mathsf{Tree\ a}$ are

$$\mathsf{List\ a} = \mu \mathsf{X}.\ \mathsf{1} + \mathsf{a} \times \mathsf{X} \tag{2}$$

and

$$\mathsf{Tree\ a} = \mu \mathsf{X}.\ \mathsf{a} + \mathsf{X} \times \mathsf{a} \times \mathsf{X} \tag{3}$$

respectively. That is, $\mathsf{List\ a}$ can be seen as a fixpoint of $\mathsf{F}_{\mathsf{List\ a}}$, where $\mathsf{F}_{\mathsf{List\ a}}\ \mathsf{X} = \mathsf{1} + \mathsf{a} \times \mathsf{X}$, by (2), since

$$\mathsf{List\ a} = \mathsf{1} + \mathsf{a} \times \mathsf{List\ a}$$

Indeed, every element of $\mathsf{List\ a}$ is either empty or is obtained by consing an element of type a onto an already-existing structure of type $\mathsf{List\ a}$, so that the above fixpoint equation is nothing more than a rewriting of the Haskell data type declaration for $\mathsf{List\ a}$. Analogously, $\mathsf{Tree\ a}$ can be seen as a fixpoint of $\mathsf{F}_{\mathsf{Tree\ a}}$, where $\mathsf{F}_{\mathsf{Tree\ a}}\ \mathsf{X} = \mathsf{a} + \mathsf{X} \times \mathsf{a} \times \mathsf{X}$, by (3). The intent here is that $\mathsf{List\ a}$, i.e., $\mu\mathsf{F}_{\mathsf{List\ a}}$, should be interpreted by the semantic fixpoint $\mu F_{List\ a}$, where the functor $F_{List\ a}$ interprets the type constructor $\mathsf{F}_{\mathsf{List\ a}}$, and $\mathsf{Tree\ a}$, i.e., $\mu\mathsf{F}_{\mathsf{Tree\ a}}$, should be interpreted by the semantic fixpoint $\mu F_{Tree\ a}$, where the functor $F_{Tree\ a}$ interprets $\mathsf{F}_{\mathsf{Tree\ a}}$.[5] A similar situation obtains for other ADTs.

The fixpoint equations above are entirely sensible at the level of types. But to ensure that the syntactic fixpoint representing an ADT actually denotes a semantic object computed as a semantic fixpoint, the corresponding semantic fixpoint calculation must converge. If, as is typical, we interpret our types as sets, then the fixpoint being taken must be of a functor on the category *Set* of sets and functions between them, rather than merely of a function between sets nLab authors (2019). That is, the function F interpreting the type constructor F constructing the body of a syntactic fixpoint must not only have an action on sets but must also have a *functorial action* on functions between sets. Reflecting this requirement back into syntax gives that F must support a function map satisfying the functor laws. That is, F must be an instance of Haskell's $\mathsf{Functor}$ class (with the aforementioned caveat about the functor laws).

Requiring F to be a functor is critical for the interpretation μF of the ADT $\mathsf{T\ a} = \mu\mathsf{F}$ to exist. But to ensure that μF is itself a functor, so that the type constructor T associated with $\mathsf{T\ a}$ also supports its own map function $\mathsf{map}_{\mathsf{T}}$, we can require that F be a functor on the category Set^{Set} of functors and natural transformations on *Set*. That is, we can require that F be a *higher-order* functor on *Set*. Writing H in place of F to emphasize that it is higher-order, and reflecting this requirement back into syntax, we have that $\mathsf{T\ a} = (\mu\mathsf{H})\ \mathsf{a}$ for a "type constructor constructor" H that supports suitable[6] map functions.

A concrete example is given by the ADT $\mathsf{List\ a}$. This type is modeled as the fixpoint $\mu F_{List\ a}$ of the first-order functor whose action on sets is given by $F_{List\ a}\ X = 1 + a \times X$ and whose action on functions is given by $F_{List\,a}\,f = id_1 + id_a \times f$. The type constructor List is modeled by the functor that is the fixpoint μH of the higher-order functor H whose action on a functor F is given by the functor $H\,F$ whose actions on sets and functions between them are given by $H\,F\,X = 1 + X \times F\,X$ and $H\,F\,f = id_1 + f \times F\,f$, respectively, and whose action on a natural transformation η is the natural transformation whose component at X is given by $(H\,\eta)_X = id_1 + id_X \times \eta_X$. Reflecting the functorial action of μH back into syntax gives exactly Haskell's built-in map function as the type-and-data-uniform, structure-preserving, data-changing function associated with List.

Note that the rearrange-transform property for fixpoint representations of ADTs is simply the reflection back into syntax of the instance of naturality for the type-independent function that rearranges structures of type T a into ones of type T′ a and the structure-preserving, data-changing functions map_T f and $\mathsf{map}_{\mathsf{T}'}$ f for a function f :: a → b, where T and T′ are the type constructors associated with these ADTs, respectively.

Whenever a data type has an interpretation as the least fixpoint of a functor – or, equivalently by Lambek's Lemma, as the carrier of the initial algebra of that functor – we say that the data type has an *initial algebra semantics* (IAS). As mentioned in Section 1, ADTs and nested types are well-known to have IAS (Johann and Ghani 2007; Johann et al. 2021; Manes and Arbib 1986). Having an IAS is the gold standard semantics for a data type: an IAS guarantees that the data type supports pattern matching on its constructors, an induction rule that can be used to prove properties on it, an elimination rule guaranteeing that functions over it can be written by recursion, etc. These essential programming tools are just reflections back into syntax of fundamental properties of IAS.

3. Syntax and Semantics of GADTs

Generalized algebraic data types (GADTs) (Peyton Jones et al. 2006) relax the restriction on the type instances appearing in a data type definition. The special form of GADTs known as *nested types* (Bird and Meertens 1998) allows the data constructors of a GADT to take as arguments data whose types involve type instances of the GADT other than the one being defined. However, the return type of each constructor of a nested type must still be precisely the one being defined. This is illustrated by the definition

$$\mathsf{PTree\ a} = \mathsf{PLeaf\ a} \mid \mathsf{PNode\ (PTree\ (a \times a))}$$

of the nested type PTree a of perfect trees, which introduces the data constructors PLeaf :: a → PTree a and PNode :: PTree (a × a) → PTree a. It enforces not only the invariant that all of the data in a structure of type PTree a is of the same type a but also the invariant that all perfect trees have lengths that are powers of 2. GADTs allow their constructors both to take as arguments *and return as results* data whose types involve type instances of the GADT other than the one being defined. An example of a GADT is Seq, whose definition is

$$\begin{aligned}
&\mathsf{data\ Seq\ a\ where}\\
&\qquad \mathsf{Const} :: \mathsf{a} \to \mathsf{Seq\ a}\\
&\qquad \mathsf{Pair}\ \ :: \mathsf{Seq\ a} \to \mathsf{Seq\ b} \to \mathsf{Seq\ (a \times b)}
\end{aligned} \tag{4}$$

Since the return type of the data constructor Pair is not of the form Seq a for any variable a, Seq is a *proper GADT*, i.e., a GADT that is not a nested type.

By contrast with the ADT List a, where the type parameter a is integral to the type being defined, the type parameter a appears in both PTree a and Seq a as a "dummy" parameter used only to give the kind $* \to *$ of the type constructors PTree and Seq. This is explicitly captured in the alternative "kind signature" Haskell syntax, which represents PTree and Seq as

$$\begin{aligned}&\mathsf{data\ PTree} :: * \to * \ \mathsf{where}\\ &\quad \mathsf{PLeaf} \ :: \ \mathsf{a} \to \mathsf{PTree\ a}\\ &\quad \mathsf{PNode} \ :: \ \mathsf{PTree\ (a \times a)} \to \mathsf{PTree\ a}\end{aligned}$$

and

$$\begin{aligned}&\mathsf{data\ Seq} :: * \to * \ \mathsf{where}\\ &\quad \mathsf{Const} :: \ \mathsf{a} \to \mathsf{Seq\ a}\\ &\quad \mathsf{Pair} \ :: \ \mathsf{Seq\ a} \to \mathsf{Seq\ b} \to \mathsf{Seq\ (a \times b)}\end{aligned}$$

respectively. A GADT – even a nested type – thus does not define a *family of inductive types*, one for each type argument, like an ADT does, but instead defines an entire family of types that must be constructed simultaneously. That is, a GADT defines an *inductive family of types*. Letting $*^n \to *$ denote $\overbrace{* \to * \to \cdots \to *}^{n \text{ asterisks}} \to *$, we take the general form of a GADT to be

$$\begin{aligned}&\mathsf{data\ G} :: *^n \to * \ \mathsf{where}\\ &\quad \mathsf{C_1} \ :: \ \mathsf{F_1}\ \overline{\mathsf{a_1}} \to \mathsf{G}\ \overline{\mathsf{K_1}\ \overline{\mathsf{a_1}}}\\ &\quad \vdots\\ &\quad \mathsf{C}_m \ :: \ \mathsf{F}_m\ \overline{\mathsf{a}_m} \to \mathsf{G}\ \overline{\mathsf{K}_m\ \overline{\mathsf{a}_m}}\end{aligned} \tag{5}$$

Here, for each $i \in \{1, \dots, m\}$, F_i is a type constructor with type signature $*^{n_i} \to *$ that can involve G itself, and each component $\overline{\mathsf{K}_{ij}}$ of $\overline{\mathsf{K}_i}$ is either a projection or a type constructor in the language without GADTs whose type signature is $*^{n_i} \to *$. In general, a type constructor T with type signature $*^k \to *$ is said to have *arity k*. The overline notation denotes a finite list whose length is exactly the arity of the type constructor being applied to it. The number of type constructors in each $\overline{\mathsf{K}_i}$ is thus n, and the number of type variables in $\overline{\mathsf{a}_i}$ is thus n_i. In addition, we require that each type constructor F_i is constructed inductively according to the following grammar, where p ranges from 1 to the length of $\overline{\mathsf{a}}$:

$$\mathsf{F}\ \overline{\mathsf{a}},\ \mathsf{F}_1\ \overline{\mathsf{a}},\ \mathsf{F}_2\ \overline{\mathsf{a}} := \mathsf{a}_p \mid \mathsf{F}_1\ \overline{\mathsf{a}} \times \mathsf{F}_2\ \overline{\mathsf{a}} \mid \mathsf{F}_1\ \overline{\mathsf{a}} + \mathsf{F}_2\ \overline{\mathsf{a}} \mid \mathsf{L} \to \mathsf{F}\ \overline{\mathsf{a}} \mid \mathsf{G}\ (\overline{\mathsf{F}}\ \overline{\mathsf{a}}) \mid \mathsf{H}\ (\overline{\mathsf{F}}\ \overline{\mathsf{a}}) \tag{6}$$

This grammar is subject to the following restrictions. In the fourth clause, neither G nor any (other) proper GADT appears in F, and L is a closed type. In the fifth clause, neither G nor any (other) proper GADT appears in any of the n type constructors in $\overline{\mathsf{F}}$. These requirements prevent the nesting of proper GADTs, which would not only render the ambient language inconsistent (Norell 2022) but would also make it impossible to obtain a parametricity theorem for any language extended with GADTs. In the sixth clause, $\mathsf{H} :: *^k \to *$ is a data type constructor defined by any nested type (including truly nested types). It thus subsumes the case in which $\mathsf{F}\ \overline{\mathsf{a}}$ is a closed type.

All of the particular GADTs considered in this paper conform to the syntax in (5). In addition to specifying the syntax of the GADTs we consider, we also assume that GADTs come equipped with constructs for defining functions (uniquely) over them. More precisely, each n-ary GADT G comes together with a rule of the following form:

$$\begin{gathered}\text{Given a type expression } \mathsf{E}\ \overline{\mathsf{a}}, \text{ whose } n \text{ free type variables are the components of}\\ \overline{\mathsf{a}}, \text{ and terms } \mathsf{t}_i :: \forall \overline{\mathsf{a_i}}.\ \mathsf{F_i}\ \overline{\mathsf{a_i}} \to \mathsf{E}\ \overline{\mathsf{K_i}}\ \overline{\mathsf{a_i}} \text{ for each } i \in \{1, \dots, m\}, \text{ there is a unique}\\ \text{term } \mathsf{t} :: \forall \overline{\mathsf{a}}.\ \mathsf{G}\ \overline{\mathsf{a}} \to \mathsf{E}\ \overline{\mathsf{a}} \text{ such that } \mathsf{t} \circ \mathsf{c}_i = \mathsf{t}_i \text{ for each } i \in \{1, \dots, m\}.\end{gathered} \tag{7}$$

This will be important in the proof of Theorem 18 below.

Proper GADTs are used in precisely those situations in which different behaviors at different instances of a data type are desired. This is achieved by allowing the programmer to give the type signatures of the GADT's data constructors independently – as is made explicit by the alternative syntax above – and then using pattern matching to force the desired type refinement. The same technique can be used to support type-indexed inductive families in dependent type theory. Note, however, that the impredicative nature of languages supporting GADTs entails they behave quite differently from those supporting type-indexed inductive families.

Applications of GADTs include generic programming, modeling programming languages via higher-order abstract syntax, maintaining invariants in data structures, and expressing constraints in embedded domain-specific languages. GADTs have also been used, e.g., to implement tagless interpreters (Pasalic and Linger 2004; Peyton Jones et al. 2006; Pottier and Régis-Gianas 2006), to improve memory performance (Minsky 2015), and to design APIs (Penner 2020).

3.1 GADTs are not functors

In this section, we show that, by contrast with the situation for ADTs and nested types, proper GADTs do not support map functions. That is, since proper GADTs are not uniform in their type parameters, they cannot be regarded as type-independent containers that can be filled with data of any type the way that ADTs and nested types can. But since a GADT's map function is just the reflection back into syntax of the functorial action of the functor interpreting it, this entails that, by contrast with the situation for ADTs and nested types, proper GADT syntax cannot be interpreted as functors on *Set*. We will consider various approaches to recovering functorial interpretations of GADTs in the remainder of the paper.

Example 1. *The GADT* Seq *defined in* (4) *comprises sequences of any type* a, *and sequences obtained by pairing the data in two already-existing sequences. Syntactically,* Seq *contains no elements other than these. We naturally expect a GADT's interpretation, like those for ADTs and nested types, to contain only those data elements that are representable by its syntax. However, the definition in* (4) *specifies that an element of* Seq *of the form* $\mathsf{Pair}\,\mathsf{t}_1\,\mathsf{t}_2$ *must have the shape of a sequence of data of pair type rather than a sequence of data of an arbitrary type. This means that the clause of* map *for the* Pair *constructor should feed* map *a function* $\mathsf{f} :: (\mathsf{a} \times \mathsf{b}) \to \mathsf{c}$ *and a term of the form* $\mathsf{Pair}\,\mathsf{t}_1\,\mathsf{t}_2$ *for* $\mathsf{t}_1 :: \mathsf{Seq}\,\mathsf{a}$ *and* $\mathsf{t}_2 :: \mathsf{Seq}\,\mathsf{b}$, *and produce a term* $\mathsf{Pair}\,\mathsf{t}_3\,\mathsf{t}_4$ *for some appropriately typed terms* t_3 *and* t_4. *However, it is not clear how to achieve this since* c *need not necessarily be a product type. And even if* c *were known to be of the form* $\mathsf{w} \times \mathsf{z}$, *we still wouldn't necessarily have a way to produce data of type* $\mathsf{w} \times \mathsf{z}$ *from only* $\mathsf{f} :: \mathsf{a} \times \mathsf{b} \to \mathsf{w} \times \mathsf{z}$ *and* t_1 *and* t_2 *unless we knew, e.g., that* f *was a product* $(\mathsf{f}_1 :: \mathsf{a} \to \mathsf{w}) \times (\mathsf{f}_2 :: \mathsf{b} \to \mathsf{z})$ *of functions. Since* Seq *does not support a map function, it cannot be made into a functor.*

As already noted, the fact that the syntax of GADTs allows non-variable type arguments in the return types of their data constructors establishes a strong connection between a GADT's shape and the data it contains. With ADTs, we first choose the shape of the container and then fill that container with data of whatever type we like; critically, the choice of shape is independent of the data to be stored. With GADTs, however, the shape of the container may actually *depend* on (the type of) the data to be contained. For example, Const can create data of any shape $\mathsf{Seq}\,\mathsf{a}$, but Pair can produce data of shape $\mathsf{Seq}\,\mathsf{a}$ only if a is a pair type. As a result, modifying the data in a GADT's container may change the shape of that container, or even produce an ill-typed result.

To determine the possible shapes of a GADT's container, we must pattern match the type of the data to be contained. For this, it is essential that a GADT calculus supports an *equality type* Eq. This type is a singleton set when its two type arguments are the same and is the empty set otherwise. That is, it is the syntactic reflection of semantic equality function *Eq*. Then, every GADT can be written in terms of Eq (Cheney and Hinze 2003; Hinze 2003; McBride 1999; Schrijvers et al. 2009; Sheard and Pasalic 2004), so that Eq is, in some sense, the quintessential GADT. Unfortunately, however, Eq itself cannot be interpreted as a functor on *Set*, as the next example shows.

Example 2. *The equality GADT* Eq *is defined by*

$$\begin{aligned}&\mathsf{data\ Eq\ a\ b\ where}\\ &\mathsf{Refl} :: \ \mathsf{Eq\ a\ a}\end{aligned} \tag{8}$$

This GADT cannot be made into a functor: if Eq *supported a function*

$$\mathsf{map}_{\mathsf{Eq}} :: (\mathsf{a} \to \mathsf{c}) \to (\mathsf{b} \to \mathsf{d}) \to \mathsf{Eq\ a\ b} \to \mathsf{Eq\ c\ d}$$

then defining

$$\begin{aligned}&\mathsf{eqElim} :: \mathsf{Eq\ a\ b} \to \mathsf{b} \to \mathsf{a}\\ &\mathsf{eqElim\ Refl\ x} = \mathsf{x}\end{aligned}$$

would allow us to construct an element

$$\mathsf{eqElim}\ (\mathsf{map}_{\mathsf{Eq}}\ \mathsf{id}_0\ \mathsf{absurd\ Refl})\ 1$$

of the empty type 0, *where* 1 *is (by abuse of notation) the unique element of the unit type* 1, *and* $\mathsf{absurd} :: 0 \to 1$ *is the empty function. But this is not possible.*

Since a proper GADT can always be written in terms of Eq, its map function must necessarily involve Eq's map function, too. But since the Eq does not support a map function, it is immediate that GADTs cannot, in general, support map functions either. For Seq, this can be seen by noting that, given a function $\mathsf{f} :: (\mathsf{a} \times \mathsf{b}) \to \mathsf{c}$, the term $\mathsf{map}_{\mathsf{Eq}\,(\mathsf{a}\times\mathsf{b})}\ \mathsf{f\ Refl}$ would have type $\mathsf{Eq}\ (\mathsf{a} \times \mathsf{b})\ \mathsf{c}$. But, as above, we have no way to produce a term of this type in the absence of a functorial map function for Eq, and thus no way to produce a term of type $\mathsf{Seq\ c}$ using the Pair constructor, as is required by the clause of $\mathsf{map}_{\mathsf{Seq}}$ for Pair. A similar analysis is obtained for other proper GADTs.

4. Recovering Functoriality

One way to read the results of Section 3.1 is as saying that, if we want to interpret types in *Set*, then we must be willing to accept that the interpretations of proper GADTs will necessarily contain "extra" elements that are not reachable in syntax. We will call such extra elements in the semantics *ghost elements*. The *functorial completion* (Johann and Polonsky 2019) of a GADT adds ghost elements to complete the interpretation of its syntax from a function on types to a functo*r* on types. As we have seen, functoriality is absolutely essential to the initial algebra semantics of data types.

Since being a functor entails supporting a map function satisfying the functor laws, the functorial interpretation of a data type must include the entire "map closure" of its syntax. Intuitively, this means that the functorial interpretation of Eq, for example, contains not just interpretations of those data elements representable by its syntax, but also interpretations of all data elements of the form $\mathsf{map}_{\mathsf{Eq}}\ \mathsf{f\ g\ s}$ for all types t_1, t_2, t_3, and t_4, all functions $\mathsf{f} :: \mathsf{t}_1 \to \mathsf{t}_3$ and $\mathsf{g} :: \mathsf{t}_2 \to \mathsf{t}_4$ definable in the language, and all $\mathsf{s} :: \mathsf{Eq}\ \mathsf{t}_1\ \mathsf{t}_2$, as well as all interpretations of data elements of the form $\mathsf{map}_{\mathsf{Eq}}\ \mathsf{h\ k\ s'}$ for all appropriately typed functions h and k and each element $\mathsf{s'}$ already added to the data type, and so on. Functorial completion for Eq adds, in particular, interpretations of the problematic data elements of the form $\mathsf{map}_{\mathsf{Eq}}\ \mathsf{h\ k\ Refl}$ from Example 2, even though these may not themselves be of the form Refl. All of these elements are ghost elements for Eq. Similarly, we see that the functorial interpretation of Seq contains not just interpretations of those data elements representable by its syntax, but also interpretations of all data elements of the form $\mathsf{map}_{\mathsf{Seq}}\ \mathsf{f}\ \mathsf{s}$ for all types t_1 and t_2, all functions $\mathsf{f} :: \mathsf{t}_1 \to \mathsf{t}_2$ definable in the language, and all $\mathsf{s} :: \mathsf{Seq}\ \mathsf{t}_1$, as well as interpretations of all data elements of the form $\mathsf{map}_{\mathsf{Seq}}\ \mathsf{g}\ \mathsf{s}'$ for each appropriately typed function g and each element s' already added to the data type, and so on. Functorial completion for Seq adds, in particular, interpretations of the problematic data elements of the form $\mathsf{map}_{\mathsf{Seq}}\ \mathsf{g}\ (\mathsf{Pair}\ \mathsf{t}_1\ \mathsf{t}_2)$ from Section 3.1, even though these may not themselves be of the form $\mathsf{Pair}\ \mathsf{t}_3\ \mathsf{t}_4$ for any terms t_3 and t_4. All of these elements are ghost elements for Seq. Importantly, functorial completion adds no ghost data elements to the interpretations of GADTs that are ADTs or other nested types.

We will now make the above precise, by showing formally that the functorial completion of a proper GADT necessarily contains more data elements than those representable in the GADT's syntax because it adds ghost elements to the GADT's interpretation. When interpreted as their functorial completions, GADTs can, like ADTs, be modeled as fixpoints of higher-order functors. Syntactically, *higher-orderness* is essential: since the type arguments to the GADT being defined are not necessarily uniform across all of its instances in the types of its data constructors, GADTs cannot be seen as first-order fixpoints the way ADTs can. Semantically, (higher-order) *functors* are essential: as in the standard semantics for ADTs, functoriality guarantees the existence of the (higher-order) fixpoints being computed (nLab authors 2019).

To illustrate the process of computing the functorial completion of a proper GADT, consider again the GADT Seq. Because its type argument varies in the instances of Seq appearing in the types of its data constructor Pair, Seq cannot be modeled as the fixpoint of any first-order functor. As shown in (Johann and Polonsky 2019), it can, however, be modeled as a solution to the higher-order fixpoint equation

$$H\,F\,b = b \,+\, (Lan_{\lambda cd.c\times d}\,\lambda cd.Fc \times Fd)\,b$$

where $Lan_K\,F$ is the left Kan extension of the functor F along the functor K. In general, the left Kan extension $Lan_K\,F : \mathscr{E} \to \mathscr{D}$ of $F : \mathscr{C} \to \mathscr{D}$ along $K : \mathscr{C} \to \mathscr{E}$ is the best functorial approximation to F that factors through K. Intuitively, "best functorial approximation" means that $Lan_K\,F$ is the smallest functor that both extends the image of K to $\mathscr{D}$ and agrees with F on $\mathscr{C}$, in the sense that, for any other such functor G, there is a morphism of functors (i.e., a natural transformation) from $Lan_K\,F$ to G. Formally, this is captured by the following definition (MacLane 1971):

Definition 3. *If $F : \mathscr{C} \to \mathscr{D}$ and $K : \mathscr{C} \to \mathscr{E}$ are functors, then the left Kan extension of F along K is a functor $Lan_K\,F : \mathscr{E} \to \mathscr{D}$ together with a natural transformation $\eta : F \to (Lan_K\,F) \circ K$ such that, for every functor $G : \mathscr{E} \to \mathscr{D}$ and natural transformation $\gamma : F \to G \circ K$, there exists a unique natural transformation $\delta : Lan_K\,F \to G$ such that $(\delta K) \circ \eta = \gamma$. This is depicted in the diagram*

$$\begin{tikzcd}
\mathscr{C} \arrow[rr, "F"] \arrow[dr, "K"'] & & \mathscr{D} \\
& \mathscr{E} \arrow[ur, "Lan_K\,F"] \arrow[ur, bend right=60, "G"'] &
\end{tikzcd}
\qquad \eta : F \Rightarrow Lan_K\,F \circ K,\quad \gamma : F \Rightarrow G \circ K,\quad \delta : Lan_K\,F \Rightarrow G$$

To represent GADTs as fixpoints in a setting in which types are interpreted as sets, a calculus must support a primitive construct Lan in such a way that the type constructor $\mathsf{Lan}_{\overline{\mathsf{K}}}\,\mathsf{F}$ is the syntactic reflection of the left Kan extension $\mathsf{Lan}_K\,F$ of the functor F interpreting F along the functor K interpreting $\overline{\mathsf{K}}$. If F and the n components of $\overline{\mathsf{K}}$ all have type signature $*^k \to *$, then $\mathsf{Lan}_{\overline{\mathsf{K}}}\,\mathsf{F}$ has type signature $*^n \to *$. It also comes together with a term $\mathsf{eta} :: \forall \mathsf{b}.\ \mathsf{F}\,\overline{\mathsf{b}} \to \mathsf{Lan}_{\overline{\mathsf{K}}}\,\mathsf{F}\,\overline{\mathsf{K}\,\overline{\mathsf{b}}}$

and the following rule: Given a type expression $\mathsf{E}\,\overline{\mathsf{a}}$, whose n free type variables are the components of $\overline{\mathsf{a}}$, and a term $\mathsf{u} :: \forall \overline{\mathsf{b}}.\, \mathsf{F}\,\overline{\mathsf{b}} \to \mathsf{E}\,\overline{\mathsf{K}\,\overline{\mathsf{b}}}$, there is a unique term $\mathsf{t} :: \forall \overline{\mathsf{a}}.\, \mathsf{Lan}_{\overline{\mathsf{K}}}\, \mathsf{F}\,\overline{\mathsf{a}} \to \mathsf{E}\,\overline{\mathsf{a}}$ such that $\mathsf{t} \circ \mathsf{eta} = \mathsf{u}$. For our purposes, the categories $\mathscr{C}$, $\mathscr{D}$, and $\mathscr{E}$ must all be of the form Set^m for some m, and the functors F and K must be finitely accessible. This ensures that the left Kan extensions $\mathsf{Lan}_\mathsf{K}\, F$ all exist, since each category Set^m is locally finitely presentable (see (Johann and Polonsky 2019) for a detailed account). Using Lan we can then rewrite the type of a constructor $\mathsf{C} :: \mathsf{F\,a} \to \mathsf{G\,(K\,a)}$ as $\mathsf{C} :: (\mathsf{Lan}_\mathsf{K}\,\mathsf{F})\,\mathsf{a} \to \mathsf{G\,a}$ since, by Definition 3, morphisms (i.e., natural transformations) from F to $G \circ K$ are in one-to-one correspondence with those from $Lan_K\, F$ to G. That is, writing $F \Rightarrow G$ for the set of natural transformations from a functor F to a functor G, we have

$$F \Rightarrow G \circ K \;\simeq\; Lan_K\, F \Rightarrow G \tag{9}$$

The calculus must also support a primitive type constructor μ that is the syntactic reflection of the (now higher-order) fixpoint operator on Set^{Set}. Using μ and Lan we can then represent a GADT as a higher-order fixpoint. For example, we can represent the GADT Seq as

$$\mathsf{Seq\ a} = (\mu\phi.\lambda \mathsf{b.\ b} + (\mathsf{Lan}_{\lambda \mathsf{cd.c} \times \mathsf{d}} \lambda \mathsf{cd}.\phi \mathsf{c} \times \phi \mathsf{d})\ \mathsf{b})\ \mathsf{a}$$

The fact that $Lan_K\, F$ is the best functorial approximation to F factoring through K means that the type constructor $\mathsf{Lan_K\,F}$ computes the smallest collection of data that is generated by the corresponding GADT data constructor's syntax and also supports a map function. Such a fixpoint representation of any GADT thus comprises the smallest data type that both includes the data specified by that GADT's syntax and also supports a map function. When viewed as fixpoints, then, proper GADTs are underspecified by their syntax.

We can use Definition 3 to make precise the intuition that functorial completion adds in new elements to interpretations of proper GADTs. This will be shown explicitly in the next example. Despite its simplicity, the GADT defined in (10) serves as an informative case study highlighting the difference between simply interpreting a proper GADT's syntax and interpreting a proper GADT's syntax as a functor – even if we are content to consider only the data elements it contains and ignore whether or not it supports a map function.

Example 4. *Syntactically, the GADT* G *defined by*

$$\begin{array}{c}\mathsf{data\ G\ a\ where}\\ \mathsf{C :: G\ 1}\end{array} \tag{10}$$

comprises a single data element, namely $\mathsf{C :: G\ 1}$. *As the definition of* G *makes clear,* G*'s effect is simply to test its argument for equality against the unit type* 1. *To compute the interpretation of* G*'s fixpoint representation we first note that the type of* G*'s solitary constructor* $\mathsf{C :: G\ 1}$ *is equivalently expressed as* $\mathsf{C :: 1 \to G\ 1}$ *or, using* (9), *as* $\mathsf{C} :: (\mathsf{Lan}_{\lambda \mathsf{u}.1}\, \lambda \mathsf{u}.1)\ \mathsf{a} \to \mathsf{G\ a}$, *where* $\lambda \mathsf{u}.1$ *is the syntactic reflection of the constantly* 1*-valued functor from the category* Set^0 *with a single object to* Set. *We can therefore represent* G *as*

$$\mathsf{G\ a} = (\mu\phi.\lambda \mathsf{b}.(\mathsf{Lan}_{\lambda \mathsf{u}.1}\, \lambda \mathsf{u}.1)\ \mathsf{b})\ \mathsf{a}$$

The interpretation of G *is obtained by computing the fixpoint of the interpretation of the body* $\lambda \mathsf{b}.(\mathsf{Lan}_{\lambda \mathsf{u}.1}\, \lambda \mathsf{u}.1)\ \mathsf{b}$ *of the syntactic fixpoint* $\mu\phi.\lambda \mathsf{b}.(\mathsf{Lan}_{\lambda \mathsf{u}.1}\, \lambda \mathsf{u}.1)\ \mathsf{b}$ *and applying the result to* a. *But since the recursion variable* ϕ *does not appear in this body, the interpretation of the fixpoint is just the interpretation of the body itself. The interpretation of* $\mathsf{G\ a}$ *is therefore* $(Lan_{\lambda u.1} \lambda u.1)\, A$, *where* A *interprets* a. *It turns out, however, that, for any set* A, $(Lan_{\lambda u.1} \lambda u.1)\, A$ *is, in fact, exactly* A. *Indeed, Proposition 7.1 of (Bush et al., 2003) gives that* $(Lan_{\lambda u.1} \lambda u.1)\, A$ *can be computed as*

$$\Big(\bigcup_{U:Set^0,\, f:(\lambda u.\,1)\,U \to A} (\lambda u.1)\, U \Big) / \sim \;=\; \Big(\bigcup_{U:Set^0,\, f:1 \to A} 1 \Big) / \sim$$

where U is the unique object of Set^0, $*$ is the unique element of the singleton set 1, and $\sim$ is the smallest equivalence relation such that $(U, f, *)$ and $(U, f', *)$ are related if

$$\begin{tikzcd}
(\lambda u.1)\, U \arrow[rr, "(\lambda u.1)\, id_U"] \arrow[dr, "f"'] & & (\lambda u.1)\, U \arrow[dl, "f'"] \\
& A &
\end{tikzcd}
\quad = \quad
\begin{tikzcd}
1 \arrow[rr, "id_1"] \arrow[dr, "f"'] & & 1 \arrow[dl, "f'"] \\
& A &
\end{tikzcd}$$

commutes, i.e., if $f = f'$. Since the relation generating $\sim$ is already an equivalence relation, we have that $(U, f, *) \sim (U, f', *)$ iff $f = f'$. Thus, up to isomorphism, $(Lan_{\lambda u.1}\lambda u.1)\, A = \{f : 1 \to A\}$, i.e., $(Lan_{\lambda u.1}\lambda u.1)\, A = A$.

Notice that this is different from what we expect just by looking at G*'s syntax. Indeed, we expect exactly one data element at instance* $\mathsf{G\ 1}$ *and no elements at any other instances. However, the interpretation of the fixpoint representation of* G *has data elements at every instance other than* $\mathsf{G\ 0}$*. These additional data elements can be obtained by reflecting back into syntax the elements $map_G\, f_a\, c \in G\, A$ resulting from applying the functorial action map_G of* G*'s interpretation G to the functions $f_a : 1 \to A$ determined by the elements a of $A \neq \emptyset$ and the interpretation c of* C.

More fundamentally, we have

Example 5. *Syntactically, the GADT* Eq *defined in* (8) *comprises the data elements* $\mathsf{Refl :: Eq\ c\ c}$ *for each type* c*, and no others. In other words,* Eq*'s effect is to test the equality of its two arguments. To compute the interpretation of the binary GADT* Eq*'s fixpoint representation, we first note that* Eq*'s sole constructor* $\mathsf{Refl :: Eq\ c\ c}$ *is equivalently expressed as* $\mathsf{Refl :: 1 \to Eq\ c\ c}$ *or, using* (9)*, as* $\mathsf{Refl :: (Lan_{\lambda c.(c,c)}\ \lambda c.1)\ a\ b) \to Eq\ a\ b}$*, where* $\mathsf{\lambda c.1}$ *is the syntactic reflection of the constantly 1-valued functor from the category Set to itself, and* $\mathsf{\lambda c.(c, c)}$ *is that of the diagonal functor from Set to Set^2 mapping every set C to the pair (C, C). We can therefore represent* Eq *as*

$$\mathsf{Eq\ a\ b} = (\mu\phi.\mathsf{\lambda c.(Lan_{\lambda c.(c,c)}\ \lambda c.1)\ c)\ a\ b})$$

The interpretation of Eq *is obtained by computing the fixpoint of the interpretation of the body* $\mathsf{\lambda d\ e.(Lan_{\lambda c.(c,c)}\ \lambda c.1)\ d\ e}$ *of the syntactic fixpoint* $\mu\phi.\mathsf{\lambda d\ e.(Lan_{\lambda c.(c,c)}\ \lambda c.1)\ d\ e}$ *and applying the result to* a *and* b*. But since the recursion variable ϕ does not appear in this body, the interpretation of the fixpoint is just the interpretation of the body itself. The interpretation of* $\mathsf{Eq\ a\ b}$ *is therefore $(Lan_{\lambda c.(c,c)}\lambda c.1)\, (A, B)$, where A and B interpret* a *and* b*, respectively. It turns out, however, that, for any sets A and B, $(Lan_{\lambda c.(c,c)}\lambda c.1)\, (A, B)$ is, in fact, the singleton set 1. Indeed, Proposition 7.1 of (Bush et al. 2003) gives that $(Lan_{\lambda c.(c,c)}\lambda c.1)\, (A, B)$ can be computed as*

$$\Big(\bigcup_{C:Set,\, f:(\lambda c.\,(c,c))\, C \to (A,B)} (\lambda c.1)\, C \Big) / \sim \;\; = \;\; \Big(\bigcup_{C:Set,\, f:C \to A,\, g:C \to B} 1 \Big) / \sim$$

where $\sim$ is the smallest equivalence relation such that $(C, f, g, *)$ and $(C', f', g', *)$ are related if there exists $h : C \to C'$ such that

$$\begin{tikzcd}
(\lambda c.(c, c))\, C \arrow[rr, "(\lambda c.(c,c))\, h"] \arrow[dr, "(f{,}g)"'] & & (\lambda c.(c, c))\, C' \arrow[dl, "(f'{,}g')"] \\
& (A, B) &
\end{tikzcd}
\quad = \quad
\begin{tikzcd}
(C, C) \arrow[rr, "(h{,}h)"] \arrow[dr, "(f{,}g)"'] & & (C', C') \arrow[dl, "(f'{,}g')"] \\
& (A, B) &
\end{tikzcd}$$

commutes. We can see that any element $(C, f, g, *)$ *is related to* $(A \times B, \pi_1, \pi_2, *)$, *where* $\pi_1 : A \times B \to A$ *and* $\pi_2 : A \times B \to B$ *are the first and second projections out of* $A \times B$, *respectively. In other words,* $\sim$ *relates any element with any other. Thus, up to isomorphism,* $(Lan_{\lambda c.(c,c)} \lambda c.1)\ (A, B) = 1$.

Notice that this is different from what we expect just looking at `Eq`*'s syntax: We expect exactly one data element at instance* `Eq a a` *for each type* `a` *and no elements at any other instances. However, the interpretation of the fixpoint representation of* `Eq a b` *has data elements at every instance. These additional data elements can be obtained by reflecting back into syntax the elements* $map_{Eq}\ (\pi_1, \pi_2)\ r \in Eq(A, B)$ *resulting from applying the functorial action* map_{Eq} *of* `Eq`*'s interpretation* Eq *to* π_1, π_2, *and the interpretation* r *of* `Refl` *in* $Eq(A \times B, A \times B)$.

4.1 Functorial interpretations in Set are insufficient

We have now seen that even though the functorial completions of `G` and `Eq` give the *smallest* extending functor interpreting GADTs, they must still introduce ghost elements. And since all proper GADTs can be written in terms of `Eq`, the same is true for them. But why should a programmer accept elements in the interpretation of a proper GADT that are not reachable in syntax? Forced to choose, a programmer would likely find the idea that a proper GADT contains data not specified by its syntax more than a little disturbing. What, they might ask, should a data type contain other than data that are constructed using its data constructors? That is, why should a proper GADT's interpretation contain ghost elements that are not specified by its syntax, and are only accessible via applications of its interpretation's functorial action?

From a semanticist's point of view, on the other hand, functorial completions of GADTs are entirely reasonable. Indeed, a semanticist would likely find the nonfunctorial nature of a GADT's syntax unnerving at best. After all, they would likely argue, if a GADT is supposed to be a data type, then the data in it shouldn't change or become ill-typed just because a function is mapped over it. The fact that this happens to GADTs when regarded just as their syntax actually highlights how GADTs *do not* generalize the essential, container-ish nature of ADTs at all. A semanticist might therefore conclude that GADTs are, at the very least, seriously misnamed.

Note, however, that even if we do accept ghost elements in the interpretation of GADTs, they still don't behave as expected. Indeed, as we show in Section 5, a language interpreting GADTs as their functorial completions cannot have parametric models (Reynolds 1983), as we would expect them to do. This means that languages with GADTs do not necessarily enjoy consequences of parametricity such as representation independence (Ahmed et al. 2009; Dreyer et al. 2012), equivalences between programs (Hur and Dreyer 2011), deep induction principles (Johann and Ghiorzi 2022; Johann and Polonsky 2020), and useful ("free") theorems about programs derived from their types alone (Wadler 1989).

5. Functorial Completion Does Not Support Parametric Semantics

Relational parametricity encodes a powerful notion of type uniformity, or representation independence, for data types in functional languages. It formalizes the intuition that a polymorphic program must act uniformly on all of its possible type instantiations by requiring that every such program preserves all relations between pairs of types at which it is instantiated. Parametricity was originally put forth by Reynolds (Reynolds 1983) for System F. It was later popularized as Wadler's "theorems for free" (Wadler 1989), so called because it can deduce properties of programs solely from their types, i.e., with no knowledge whatsoever of the text of the programs involved. Most of Wadler's free theorems are consequences of naturality for polymorphic list-processing functions. However, parametricity can also derive results that go beyond just naturality, such as inhabitation results. It can also be used to prove the equivalence of Church encodings and fixpoint representations of ADTs and nested types by validating shortcut fusion and other program equivalences for them (Johann 2002; Wadler 1989).

To show that interpreting GADTs as their functorial completions cannot lead to a parametric semantics, and thus that the semantics they do lead to are unsatisfactory for reasoning about programs involving GADTs, we will need to interpret data types not just in *Set*, but in a suitable category of relations as well. The following definition is standard:

Definition 6. *The category Rel has:*

- *objects: A relation is a triple (A, B, R), where R is a subset of $A \times B$.*
- *morphisms: A morphism from (A, B, R) to (A', B', R') is a pair $(f : A \to A', g : B \to B')$ of functions in Set such that $(fa, g\, b) \in R'$ if $(a, b) \in R$.*
- *identities: The identity morphism on (A, B, R) is the pair $(id_A : A \to A, id_B : B \to B)$.*
- *composition: Composition is the componentwise composition in Set. That is, $(g_1, g_2) \circ (f_1, f_2) = (g_1 \circ f_1, g_2 \circ f_2)$, where the composition being defined on the left-hand side is in Rel, and the two componentwise compositions on the right-hand side are in Set.*

We write $R \in Rel\,(A, B)$ for $(A, B, R) \in Rel$. If $R \in Rel\,(A, B)$ then we write $\pi_1 R$ and $\pi_2 R$ for the *domain* A and *codomain* B of R, respectively. We write $I_A = (A, A, \{(x, x) \mid x \in A\})$ for the *equality relation* on the set A.

The key idea underlying parametricity is to give each type $\mathsf{G[a]}$[7] with one free variable a a *set interpretation* G_0 taking sets to sets and a *relational interpretation* G_1 taking relations $R \in Rel\,(A, B)$ to relations $G_1\, R \in Rel\,(G_0\, A, G_0\, B)$, and to interpret each term $\mathsf{t\,(a, x) :: G[a]}$ with one free term variable $\mathsf{x :: F[a]}$ as a function t associating to each set A a morphism $t\,A : F_0\,A \to G_0\,A$ in *Set*. Here, F_0 is the set interpretation of F. These interpretations are given inductively on the structures of G and t in such a way that they imply two fundamental theorems. The first is an *Identity Extension Lemma*, which states that $G_1\, I_A = I_{G_0 A}$, and is the essential property that makes a model relationally parametric rather than just induced by a logical relation. The second is an *Abstraction Theorem*, which states that, for any $R \in Rel\,(A, B)$, $(t\,A, t\,B)$ is a morphism in *Rel* from $(F_0\,A, F_0\,B, F_1\,R)$ to $(G_0\,A, G_0\,B, G_1\,R)$. The Identity Extension Lemma is similar to the Abstraction Theorem except that it holds for *all* elements of a type's interpretation, not just those that interpret terms. Similar theorems are required for types and terms with any number of free variables. In particular, if t is closed (i.e., has no free term variables) then $t\,A \in G_0\,A$ for all $A \in Set$, and $(t\,A, t\,B) \in G_1\,R$ for all $A, B \in Set$ and all $R \in Rel(A, B)$.

Before showing that languages interpreting GADTs as their functorial completions cannot have parametric models, we first show that languages interpreting them as the interpretations of their Church encodings can. The Church encoding of an ADT or nested type (see, e.g., (Geuvers 2014; Koopman et al. 2014; Pierce 2002)) represents that data type as a term in the higher-order polymorphic lambda calculus F_ω (Barendregt 1984), an extension of System F whose type expressions include functions from types to types (i.e., type constructors) and whose terms can abstract over types of all kinds. In particular, expressions of any kind can be universally quantified over variables of any kind. This makes it possible to give Church encodings of ADTs and nested types in F_ω that are similar to, e.g., the standard Church encoding of natural numbers in System F. For example, the Church encoding of the type $\mathsf{List\ a}$ in F_ω is

$$\forall \mathsf{f}.(\forall \mathsf{b}.\, \mathsf{f\,b}) \to (\forall \mathsf{b}.\, \mathsf{b} \to \mathsf{f\,b} \to \mathsf{f\,b}) \to \mathsf{f\,a}$$

and that of $\mathsf{PTree\ a}$ is

$$\forall \mathsf{f}.(\forall \mathsf{b}.\, \mathsf{b} \to \mathsf{f\,b}) \to (\forall \mathsf{b}.\, \mathsf{f\,(b \times b)} \to \mathsf{f\,b}) \to \mathsf{f\,a}$$

GADTs can be encoded in the same way. For instance, the Church encoding of the type $\mathsf{Seq}\ \mathsf{a}$ is

$$\forall \mathsf{f}.(\forall \mathsf{b}.\, \mathsf{b} \to \mathsf{f}\,\mathsf{b}) \to (\forall \mathsf{c}\,\mathsf{d}.\, \mathsf{f}\,\mathsf{c} \to \mathsf{f}\,\mathsf{d} \to \mathsf{f}\,(\mathsf{c} \times \mathsf{d})) \to \mathsf{f}\,\mathsf{a}$$

while that of $\mathsf{G}\ \mathsf{a}$ is

$$\forall \mathsf{f}.\, \mathsf{f}\,1 \to \mathsf{f}\,\mathsf{a}$$

The fact that interpreting GADTs as the interpretations of their Church encodings does admit parametric models (Atkey 2012), whereas interpreting them as their functorial completions does not, means that these two interpretations of GADTs cannot possibly be equivalent. Taken together, Examples 7 and 8 will show that they actually behave very differently with respect to parametricity. This contrasts sharply with the fact that both ADTs and nested types have the same parametricity properties regardless of whether they are interpreted as the interpretations of their Church encodings or as their functorial completions. This is yet another way in which the functorial completion semantics for GADTs is unsatisfactory: it specializes in the standard IAS for ADTs, but it doesn't share the same parametricity properties.

The existence of parametric models that interpret GADTs as the interpretations of their Church encodings follows from, e.g., the existence of the parametric model of F_ω constructed in (Atkey 2012). In that model, types are interpreted "in parallel" in (types corresponding to) *Set* and *Rel* in the usual way, including the familiar "cutting down" of the interpretations of ∀-types to just those elements that are "parametric" (Reynolds 1983; Wadler 1989) to ensure that the Identity Extension Lemma holds. If the set interpretation of Eq is the function Eq and if the relational interpretation of a closed type a with interpretation A is I_A as intended, then the parametricity property for a GADT is an inhabitation result saying that the set interpreting any instance of that GADT contains exactly the interpretations of the data elements that can be formed using its data constructors and whose type is that instance. The parametricity property for the Church encoding of a GADT G gives that $\mathsf{G}\ \mathsf{a}$ is inhabited iff data elements of the instance of G at a can be formed using G's data constructors. In particular, the parametricity property for the GADT G from (10) gives that $\mathsf{G}\ \mathsf{a}$ contains a single data element if a is semantically equivalent to 1 and none otherwise. Indeed, we have

Example 7. *Let* t *be a closed term of type* $\mathsf{G}\ \mathsf{a}$ *for the GADT* G *defined in* (10), *let* $G = (G_0, G_1)$ *be the interpretation of the Church encoding of* G, *let* t *be the interpretation of* t, *and let* $R \in Rel\,(A, B)$. *Then* $t\,A \in G_0\,A$ *and* $t\,B \in G_0\,B$ *and, by the Abstraction Theorem (Theorem 3) in* (Atkey 2012), $t\,A$ *and* $t\,B$ *must be related in* $G_1\,R$. *However, under the semantics given in* (Atkey 2012), *which includes the aforementioned interpretations of* Eq *and closed types, the relational interpretation* $G_1\,R$ *of* G *is itself* $I_{G_0 1}$ *when* R *is the relational interpretation* I_1 *of* 1, *and the empty relation whenever* R *differs from* I_1. *Reflecting back into syntax we deduce that there can be no term in the type that is the Church encoding of* $\mathsf{G}\ \mathsf{a}$ *unless* a *is semantically equivalent to* 1.

For functorial completion interpretations of GADTs, the story is completely different. If, as intended, the set interpretation of $\mathsf{Lan}_\mathsf{K}\,\mathsf{F}$ is $Lan_K\,F$, where K is the set interpretation of K and F is the set interpretation of F, then the exact same reasoning gives that the relational interpretation of $\mathsf{Lan}_\mathsf{K}\,\mathsf{F}$ is $Lan_K\,F$, where K is the relational interpretation of K and F is the relational interpretation of F. But under these interpretations, there can be no parametric model. The following counterexample establishes this surprising result.

Example 8. *In any parametric model, we must give both a set interpretation and a relational interpretation for every type as described at the start of this section. In particular, for every GADT* G *we must give an interpretation* $G = (G_0, G_1)$ *such that, for every relation* $R \in Rel\,(A, B)$, *we have* $G_1\,R \in Rel\,(G_0\,A, G_0\,B)$. *Intuitively, when* G *is viewed as a fixpoint, its data elements include those given by its functorial completion. Since* G_1 *is a functor, given any relation* $S \in Rel\,(C, D)$ *and any* *morphism $m : S \to R$, $G_1\,R$ must contain all elements of the form $G_1\,m\,x$ for $x \in G_1\,S$. But the two components $m_1 : C \to A$ and $m_2 : D \to B$ of m cannot be given independently of one another, since Definition 6 entails that $(m_1\,c, m_2\,d)$ must be in R whenever (c, d) is in S for (m_1, m_2) to be a well-defined morphism of relations. The domain of $G_1\,R$ thus depends on both A and B, rather than simply on A. Likewise, the codomain of $G_1\,R$ also depends on both A and B. The domain and codomain therefore cannot simply be $G_0\,A$ and $G_0\,B$, respectively. This suggests that GADTs might fail to have relational interpretations, and thus might fail to have parametric models, as described in the previous paragraph.*

We can make this informal argument formal by providing a concrete counterexample. Consider again the GADT G given by (10). The set functorial completion interpretation of G is $Lan_{\lambda u.1}\,\lambda u.1$, i.e., is, by the reasoning of Example 4, the identity functor on Set. By the exact same reasoning, this time in Rel rather than in Set, the relational functorial completion interpretation of G is $Lan_{\lambda u.\,I_1}\,\lambda u.\,I_1$, where $\lambda u.\,I_1$ is the constantly I_1-valued functor from the category $Rel^{\,0}$ with a single object to Rel. Indeed, this interpretation is still a left Kan extension, but now it is the left Kan extension determined by the functor interpreting $\lambda \mathsf{u}.1$ in Rel. For the Identity Extension Lemma to hold, for every relation $R \in Rel\,(A, B)$ we would need $(\mathsf{Lan}_{\lambda u.\,I_1}\,\lambda u.\,I_1)\,R$ to be a relation between the sets $(Lan_{\lambda u.1}\,\lambda u.1)\,A$ and $(Lan_{\lambda u.1}\,\lambda u.1)\,B$, i.e., between the sets A and B. However, this need not be the case.

Consider the relation $R = (1, 2, 1 \times 2)$, where 1×2 relates the single element of 1 to both elements of the two-element set 2. We expect $(Lan_{\lambda u.\,I_1}\lambda u.\,I_1)\,R$ to be a relation with domain 1. Since left Kan extensions preserve projections (Riehl 2016), *we can compute the domain as*

$$\pi_1\,\big((Lan_{\lambda u.\,I_1}\lambda u.\,I_1)\,R\big) \;=\; (Lan_{\lambda u.\,I_1}\lambda u.\,1)\,R$$

(Note that the left Kan extension of $\lambda u.\,1$ along $\lambda u.\,I_1$ is a functor from Rel to Set.) By the same reasoning as in Example 4, Proposition 7.1 of (Bush et al. 2003) gives that $(Lan_{\lambda u.I_1}\lambda u.1)\,R$ can be computed as

$$\Big(\bigcup_{U:Rel^0,\; m:(\lambda u.\,I_1)\,U \to R} (\lambda u.1)\,U\Big)\,/ \approx \;=\; \Big(\bigcup_{U:Rel^0,\; m:I_1 \to R} 1\Big)\,/ \approx$$

where U is the unique object of Rel^0, $*$ is the unique element of the singleton set 1, and $\approx$ is the smallest equivalence relation such that $(U, m, *)$ and $(U, m', *)$ are related if

$$\begin{tikzcd} (\lambda u.I_1)\,U \arrow[rr, "(\lambda u.I_1)\,id_U"] \arrow[dr, "m"'] & & (\lambda u.I_1)\,U \arrow[dl, "m'"] \\ & R & \end{tikzcd} \;=\; \begin{tikzcd} I_1 \arrow[rr, "id_{I_1}"] \arrow[dr, "m"'] & & I_1 \arrow[dl, "m'"] \\ & R & \end{tikzcd}$$

commutes, i.e., if $m = m'$. Since the relation generating $\approx$ is already an equivalence relation, we have that $(U, m, *) \approx (U, m', *)$ iff $m = m'$. Thus, up to isomorphism, $(Lan_{\lambda u.\,I_1}\lambda u.\,1)\,R = \{m : I_1 \to R\}$. But this set is $\{(!, k_0), (!, k_1)\}$, where $k_0, k_1 : 1 \to 2$ are the constantly 0-valued and 1-valued functions in Set, respectively, and therefore $\pi_1\,\big((Lan_{\lambda u.\,I_1}\lambda u.\,I_1)\,R\big)$ is not 1, as would be needed for the Identity Extension Lemma to hold. Since the Identity Extension Lemma does not hold for models in which GADTs are interpreted by their functorial completions, such models cannot possibly be parametric.

It is actually possible to construct a simpler counterexample to the Identity Extension Lemma for functorial completion interpretations of GADTs using the relation $R = (1, \emptyset, \emptyset)$. However, this relation is somewhat artificial, in the sense that its domain is larger than is strictly necessary to define an empty relation. Since it is also too degenerate to properly expose the mismatch between left Kan extensions at the level of sets and left Kan extensions at the level of relations, we give the above example using the relation $(1, 2, 1 \times 2)$ instead.

One way to read the result of this subsection is as in (Johann et al. 2021): if we interpret types in *Set* and n-ary GADTs as functors from Set^n to *Set*, then, as with software engineering's iron triangle, we can have any two of *GADTs*, *functoriality*, and *parametricity* we like, but we cannot have all three.

6. Functorial Interpretations in *PSet* and Beyond

We have seen in Section 4.1 that interpreting GADTs as their functorial completions is unsatisfactory. We note, however, that partiality is inherent in the syntax of GADTs. Indeed, one way to understand Examples 1 and 2 is as showing that the map functions for Seq, as well as for Eq (and thus those for all proper GADTs), do not map arbitrary functions over their elements. That is, proper GADTs' map functions are only partially defined (Johann and Cagne 2022; Johann and Ghani 2008; Johann and Polonsky 2019). To interpret GADTs as functors without adding ghost elements to their interpretations, the category in which we interpret them must therefore account for partial functions.

To find a semantics of GADTs that accounts for the partiality of their map functions, we need to allow the functorial actions of GADTs' interpretations to yield partial functions. However, the interpretations of the functions that are representable in syntax and don't involve mapping over elements of GADTs should still be total. That is, we seek categories $\mathscr{C}$ that capture partiality in the computationally relevant sense that once a computation diverges it does not become defined again, and in which *Set* embeds in such a way that it can be considered the subcategory of "total morphisms" of $\mathscr{C}$ for some reasonable notion of totality. Obviously, the category *PSet* of sets and partial functions between them is such a category $\mathscr{C}$ since morphisms there propagate undefinedness. But rather than focusing on the specific category *PSet*, we present here an abstract framework that encompasses much more. The advantage of introducing a framework is two-fold. First, our main result (Theorem 18) holds in categories more general than *PSet*, including exotic categories $\mathscr{C}$ in which *Set* embeds nicely as a subcategory of total morphisms. And secondly, the framework does not require that the category of total morphisms is *Set* specifically. The latter entails that Theorem 18 holds not just when the subcategory of total morphisms is *Set*, but when it is any locally presentable category in which ADTs and nested types can be given their standard IAS.

The reader may wonder why we develop an abstract framework for our results about interpreting GADTs as functors in *PSet*, but were content to restrict attention to the specific category *Set* in the previous sections even though, as noted there, we could also have developed the results in those sections for arbitrary locally presentable categories. The main reason is that how to move from *Set* to locally presentable categories is well-known in the literature, so moving to the more abstract setting in Sections 2 through 5 provides no additional insights that cannot be gleaned from *Set*. On the other hand, how to move from *PSet* to more general categories $\mathscr{C}$ of the kind we seek now has not previously been known, so the categorical framework for partiality that we provide here is itself part of the contribution of this paper.

We now identify those categories capturing computationally relevant partiality. We then show in Theorem 18 that any semantics in such a category is trivial if we insist that the interpretations of GADTs in them must extend to functors. Given a category $\mathscr{C}$, we write $\mathsf{Mor}(\mathscr{C})$ for its (possibily large) set of morphisms. We begin by recalling two classic definitions, the first of which "categorifies" the notion of an ideal in monoids.

Definition 9. *A cosieve in a category $\mathscr{C}$ is a (possibly large) subset $S \subseteq \mathsf{Mor}(\mathscr{C})$ such that for all morphisms $f : A \to B$ and $g : B \to C$ in $\mathscr{C}$, if $f \in S$ then $gf \in S$.*

Definition 10. *A wide subcategory of a category $\mathscr{C}$ is a subcategory of $\mathscr{C}$ that contains all objects of $\mathscr{C}$.*

If $\mathscr{D}$ is any subcategory of $\mathscr{C}$, we write $\overline{\mathscr{D}}$ for the *complement* of $\mathscr{D}$, i.e., for the (possibly large) set $\mathsf{Mor}\,(\mathscr{C}) \setminus \mathsf{Mor}\,(\mathscr{D})$. We can then introduce the following new definition:

Definition 11. *A structure of computational partiality on a category $\mathscr{C}$ is a wide subcategory $\mathscr{D}$ of $\mathscr{C}$ such that $\overline{\mathscr{D}}$ is a cosieve.*

In a category $\mathscr{C}$ equipped with a structure of computational partiality $\mathscr{D}$, we call the morphisms of $\mathscr{D}$ *total* and those of $\overline{\mathscr{D}}$ *properly partial*. Morphisms of $\mathscr{C}$ might be referred to simply as *partial*. It is not hard to see that *Set* is a structure of computational partiality on *PSet* and that the terminology introduced in this paragraph accords with what is used there. Indeed, the intuition behind Definition 11 is that $\overline{\mathscr{D}}$ is the collection of computations that are actually undefined on a non-empty set of objects of $\mathscr{C}$ (equivalently, of $\mathscr{D}$). Following that intuition, both identities and compositions of total functions must be total functions, and when a function yields an error on an input there is no way to come back from the error by postcomposing with another function. Other categorical frameworks capturing partiality include p-categories (Robinson and Rosolini 1988), (bi)categories of partial maps (Carboni 1987), categories of partial morphisms (Curien and Obtułowicz 1989), and restriction categories (Cockett and Lack 2002). These all give rise to structures of computational partiality.

Recall that a *split monomorphism* $s : A \to B$ in a category $\mathscr{C}$ is a monomorphism in $\mathscr{C}$ for which there exists a morphism $r : B \to A$ such that $rs = id_A$. We have the following basic fact about split monomorphisms in a category equipped with a structure of computational partiality:

Lemma 12. *In a category equipped with a structure of computational partiality, split monomorphisms are always total.*

Proof. Let $s : A \to B$ be a split monomorphism in a category $\mathscr{C}$, and suppose $r : B \to A$ is such that $rs = id_A$. If s was properly partial in a structure of computational partiality on $\mathscr{C}$, then rs, and thus id_A, would be properly partial as well. But id_A is total by definition, so it cannot be. Thus, s must be total. □

Our aim is to consider interpretations of (languages supporting) GADTs in a category $\mathscr{C}$ equipped with a structure of computational partiality $\mathscr{D}$. However, to do so we require a bit more structure. Specifically, $\mathscr{D}$ must have finite products, and those products in $\mathscr{D}$ must extend to $\mathscr{C}$ in the sense introduced and justified below.

Definition 13. *Let $\mathscr{C}$ be a category equipped with a structure of computational partiality $\mathscr{D}$, and write $\iota : \mathscr{D} \to \mathscr{C}$ for the inclusion functor from $\mathscr{D}$ to $\mathscr{C}$. Suppose further that $\mathscr{D}$ has finite products, and write $\prod_n^{\mathscr{D}} : \mathscr{D}^n \to \mathscr{D}$ for the n-ary product functor from $\mathscr{D}^n$ to $\mathscr{D}$. Then the products of $\mathscr{D}$ extend to $\mathscr{C}$ if, for each $n \geq 0$, there exists a functor $\bigotimes_n^{\mathscr{C}} : \mathscr{C}^n \to \mathscr{C}$ such that the following square commutes:*

$$\begin{tikzcd}
\mathscr{D}^n \arrow[r, "\prod_n^{\mathscr{D}}"] \arrow[d, "\iota^n"'] & \mathscr{D} \arrow[d, "\iota"] \\
\mathscr{C}^n \arrow[r, "\bigotimes_n^{\mathscr{C}}"'] & \mathscr{C}
\end{tikzcd}$$

We write $\times^{\mathscr{D}}$ and $\otimes^{\mathscr{C}}$ infix rather than $\Pi_2^{\mathscr{D}}$ and $\bigotimes_2^{\mathscr{C}}$ prefix when $n = 2$. Crucially, even when $\mathscr{C}$ has products, the object $\bigotimes_n^{\mathscr{C}} A_i$ need not be a product of $A_1, \ldots, A_n$ in $\mathscr{C}$. The following examples help clarify this observation.

Example 14. *For any category $\mathscr{C}$, $\mathscr{C}$ itself is (trivially) a structure of computational partiality on $\mathscr{C}$. Moreover, if $\mathscr{C}$ has finite products, then they extend (trivially) to $\mathscr{C}$. For each n, $\bigotimes_n^{\mathscr{C}}$ is simply the actual n-ary product functor $\prod_n^{\mathscr{C}}$.*

Example 15. *The category Set is a structure of computational partiality on the category PSet whose products extend to PSet. We consider the case for $n = 2$ explicitly; those for other values of n are analogous.*

Define the functor $\_ \otimes \_ : PSet^2 \to PSet$ whose action on objects is given by $A_1 \otimes A_2 = A_1 \times A_2$, where $\_ \times \_$ denotes the usual cartesian product in Set, and whose action on morphisms sends partial functions $f_1 : A_1 \to B_1$ and $f_2 : A_2 \to B_2$ to the partial function $f_1 \otimes f_2 : A_1 \times A_2 \to B_1 \times B_2$ given by

$$(f_1 \otimes f_2)(a_1, a_2) = \begin{cases} (f_1(a_1), f_2(a_2)) & \text{if both } f_1(a_1) \text{ and } f_2(a_2) \text{ are defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Then if f_1 and f_2 are total functions, $f_1 \otimes f_2$ is total as well and actually coincides with the cartesian product of f_1 and f_2 in Set. That is, for the inclusion $\iota : Set \to PSet$, we have $\iota(\_) \otimes \iota(\_) = \iota(\_ \times \_)$.

Notice, however, that $f_1 \otimes f_2$ is not the product of f_1 and f_2 in PSet. Indeed, the product of two objects A_1 and A_2 in PSet is the disjoint union $A_1 + (A_1 \times A_2) + A_2$. Writing i, j, and k for the three canonical injections into this disjoint union, the product $f_1 \times f_2$ in PSet is the partial function from $A_1 + (A_1 \times A_2) + A_2$ to $B_1 + (B_1 \times B_2) + B_2$ defined by

$$\begin{array}{rlr} (f_1 \times f_2)(i(a_1)) & = i(f_1(a_1)) & \text{if } f_1(a_1) \text{ defined} \\ (f_1 \times f_2)(j(a_1, a_2)) & = j(f_1(a_1), f_2(a_2))) & \text{if } f_1(a_1) \text{ and } f_2(a_2) \text{ both defined} \\ (f_1 \times f_2)(j(a_1, a_2)) & = i(f_1(a_1)) & \text{if } f_1(a_1) \text{ defined and } f_2(a_2) \text{ undefined} \\ (f_1 \times f_2)(j(a_1, a_2)) & = k(f_2(a_2)) & \text{if } f_1(a_1) \text{ undefined and } f_2(a_2) \text{ defined} \\ (f_1 \times f_2)(k(a_2)) & = k(f_2(a_2)) & \text{if } f_2(a_2) \text{ defined} \\ (f_1 \times f_2)(x) & = \text{undefined} & \text{otherwise} \end{array}$$

The square

$$\begin{tikzcd}
A_1 \times A_2 \arrow[r, "f_1 \otimes f_2"] \arrow[d, "j"'] & B_1 \times B_2 \arrow[d, "j"] \\
A_1 + (A_1 \times A_2) + A_2 \arrow[r, "f_1 \times f_2"'] & B_1 + (B_1 \times B_2) + B_2
\end{tikzcd}$$

thus need not commute in PSet.

Example 16. *Consider the category ωpCPO of complete partial orders with bottom elements and strict Scott-continuous functions between them. Write ωpCPO_t for the wide subcategory containing only those morphisms $f : D \to D'$ such that $f^{-1}(\bot_{D'}) = \{\bot_D\}$, where $\bot_D$ is the bottom element of D and $\bot_{D'}$ is the bottom element of D'. The product $\prod_n^{\omega\text{pCPO}_t} D_i$ of objects $D_1, \ldots, D_n$ in ωpCPO_t is the subset of the cartesian product in Set of the sets underlying the D_is comprising those tuples $(x_1, \ldots, x_n)$ such that either $x_i = \bot_{D_i}$ for all i or $x_i \neq \bot_{D_i}$ for all i. The product $\prod_n^{\omega\text{pCPO}_t} f_i$*

of morphims $f_1 : D_1 \to D'_1, \ldots, f_n : D_n \to D'_n$ in ωpCPO$_t$ maps $(x_1, \ldots, x_n)$ to $(\bot_{D'_1}, \ldots, \bot_{D'_n})$ if $x_i = \bot_{D_i}$ for some i, and maps $(x_1, \ldots, x_n)$ to $(f_1(x_1), \ldots, f_n(x_n))$ otherwise. The category ωpCPO$_t$ is thus a structure of computational partiality on ωpCPO whose products extend to ωpCPO. Indeed, if $f_i : D_i \to D'_i$, $i = 1, \ldots, n$, are strict Scott-continuous functions, then there is a strict Scott-continuous function $\bigotimes_n^{\omega\mathrm{pCPO}} f_i : \prod_n^{\omega\mathrm{pCPO}_t} D_i \to \prod_n^{\omega\mathrm{pCPO}_t} D'_i$ that maps $(x_1, \ldots, x_n)$ to $(\bot_{D'_1}, \ldots, \bot_{D'_n})$ if $f_i(x_i) = \bot_{D'_i}$ for some i, and maps $(x_1, \ldots, x_n)$ to $(f_1(x_1), \ldots, f_n(x_n))$ otherwise. Note that if each f_i is in ωpCPO$_t$ then $\bigotimes_n^{\omega\mathrm{pCPO}} f_i$ is as well and actually coincides with the product of the f_is in ωpCPO$_t$. Moreover, the construction above is clearly functorial in the f_is. Thus, for the inclusion $\iota : \omega\mathrm{pCPO}_t \to \omega\mathrm{pCPO}$, we have $\bigotimes_n^{\omega\mathrm{pCPO}} \circ \iota^n = \iota \circ \prod_n^{\omega\mathrm{pCPO}_t}$.

Note, however, that $\bigotimes_n^{\omega\mathrm{pCPO}} f_i$ is not the product of $f_1, \ldots, f_n$ in ωpCPO. The product $\Pi_n^{\omega\mathrm{pCPO}} D_i$ of $D_1, \ldots, D_n$ in ωpCPO is the cartesian product of the sets underlying the D_is, ordered componentwise, and the product $\Pi_n^{\omega\mathrm{pCPO}} f_i : \prod_n^{\omega\mathrm{pCPO}} D_i \to \prod_n^{\omega\mathrm{pCPO}} D'_i$ in ωpCPO of morphisms $f_i : D_i \to D'_i$ is the morphism mapping $(x_1, \ldots, x_n)$ to $(f_1(x_1), \ldots, f_n(x_n))$. Writing $j_{X_1,\ldots,X_n}$ for the canonical inclusion of $\prod_n^{\omega\mathrm{pCPO}_t} X_i$ into $\prod_n^{\omega\mathrm{pCPO}} X_i$, we thus see that the square

$$\begin{tikzcd}
\prod_n^{\omega\mathrm{pCPO}_t} D_i \arrow[r, "\bigotimes_n^{\omega\mathrm{pCPO}} f_i"] \arrow[d, "j_{D_1,\ldots,D_n}"'] & \prod_n^{\omega\mathrm{pCPO}_t} D'_i \arrow[d, "j_{D'_1,\ldots,D'_n}"] \\
\prod_n^{\omega\mathrm{pCPO}} D_i \arrow[r, "\prod_n^{\omega\mathrm{pCPO}} f_i"'] & \prod_n^{\omega\mathrm{pCPO}} D'_i
\end{tikzcd}$$

need not commute in ωpCPO.

Now fix a category $\mathscr{C}$ equipped with a structure of computational partiality $\mathscr{D}$. Suppose that $\mathscr{D}$ has finite products and that they extend to $\mathscr{C}$. We want to define a good notion of an *interpretation of (a language supporting) GADTs in* $(\mathscr{C}, \mathscr{D})$. Although we want to remain as language-agnostic as possible, so that our result can be replayed in as many different settings as possible, we must still make some reasonable assumptions about the type theory underlying the ambient language. Such a type theory necessarily describes how to construct types from a given context of type variables, and how to construct typed terms from a given context of typed term variables. It should also come with a mechanism (such as an operational semantics) that verifies that a term is *terminating*. Non-terminating terms represent those programs that loop infinitely or are undefined on certain inputs. Terminating terms represent those programs that compute actual values in finite time.

The key idea guiding Definition 17 below is to interpret contexts (and thus types) by objects of $\mathscr{C}$ and to interpret terms by morphisms of $\mathscr{C}$ in such a way that the interpretations of terminating terms are actually in $\mathscr{D}$. With this in mind, we are led to interpret a context Γ comprising variables of types $\mathsf{a}_1, \ldots, \mathsf{a}_n$ as the product $\Pi_n^{\mathscr{D}} [\![\mathsf{a}_i]\!]$ in $\mathscr{D}$ of the interpretations $[\![\mathsf{a}_i]\!]$ of a_i for $i = 1, \ldots, n$. Now, in any context Γ, we can always define the term that picks out the i^{th} variable: it is a terminating term (it represents the program that takes n inputs and simply returns the i^{th}), so its interpretation must be a morphism $\pi_i : [\![\Gamma]\!] \to [\![\mathsf{a}_i]\!]$ in $\mathscr{D}$. To see that $[\![\Gamma]\!]$, together with the π_is, actually constitute a product in $\mathscr{D}$, we have to turn to the rules that govern the production of morphisms of contexts in the underlying type theory. A morphism from a context Δ to a context Γ is given by a term of type a in context Δ for each type a appearing in Γ. Such a morphism of contexts is considered terminating only if each of the terms defining it is terminating; this corresponds to the intuitive expectation that the simultaneous run of the programs represented by the terms terminates successfully if and only if each of the runs terminates individually.

This programming language feature can be expressed in the semantics by requiring that $[\![\Gamma]\!]$ has the following property: for any object C of $\mathscr{C}$, every tuple of morphisms $f_i : C \to [\![\mathsf{a}_i]\!]$ in $\mathscr{C}$ defines a morphism $f : C \to [\![\Gamma]\!]$ in $\mathscr{C}$. Moreover, f is in $\mathscr{D}$ whenever all the f_is are, in which case $f_i = \pi_i f$ for $i = 1, \ldots, n$, and f is uniquely determined by the f_is. In other words, $[\![\Gamma]\!]$, together with the π_is, is a product of the $[\![\mathsf{a}_i]\!]$s in $\mathscr{D}$.

Now that we have established that it is natural to interpret each context Γ as the product $\prod_n^{\mathscr{D}} [\![\mathsf{a}_i]\!]$ in $\mathscr{D}$ of the types a_i, $i = 1, \ldots, n$, it comprises, we explain why it is equally natural to expect the products of $\mathscr{D}$ to extend to $\mathscr{C}$. We focus on the case $n = 2$ for illustrational purposes. Suppose we are given terms t_1 of type b_1 and t_2 of type b_2 in context Γ, and suppose t_1 and t_2 are interpreted by morphisms $[\![\mathsf{t}_1]\!] : [\![\Gamma]\!] \to [\![\mathsf{b}_1]\!]$ and $[\![\mathsf{t}_2]\!] : [\![\Gamma]\!] \to [\![\mathsf{b}_2]\!]$ in $\mathscr{C}$, respectively. Write Γ' for the context containing only a variable of type b_1 and a variable of type b_2. We need to be able to construct the interpretation of the morphism of contexts from Γ to Γ' given by the pair $(\mathsf{t}_1, \mathsf{t}_2)$. That is, we want to construct from $[\![\mathsf{t}_1]\!]$ and $[\![\mathsf{t}_2]\!]$ a morphism $[\![(\mathsf{t}_1, \mathsf{t}_2)]\!] : [\![\Gamma]\!] \to [\![\mathsf{b}_1]\!] \times^{\mathscr{D}} [\![\mathsf{b}_2]\!]$ in $\mathscr{C}$. Moreover, when t_1 and t_2 are terminating terms, their interpretations are morphisms of $\mathscr{D}$ and we want $[\![(\mathsf{t}_1, \mathsf{t}_2)]\!]$ to be in $\mathscr{D}$ as well. In addition, in that case we want to recover the usual interpretation of morphisms of contexts, i.e., we want $[\![(\mathsf{t}_1, \mathsf{t}_2)]\!]$ to be the morphism

$$[\![\Gamma]\!] \xrightarrow{\text{diag}} [\![\Gamma]\!] \times^{\mathscr{D}} [\![\Gamma]\!] \xrightarrow{[\![\mathsf{t}_1]\!] \times^{\mathscr{D}} [\![\mathsf{t}_2]\!]} [\![\mathsf{b}_1]\!] \times^{\mathscr{D}} [\![\mathsf{b}_2]\!]$$

in $\mathscr{D}$. (Note that diag is a morphism in $\mathscr{D}$.) When $[\![\mathsf{t}_1]\!]$ and $[\![\mathsf{t}_2]\!]$ are not total, this construction cannot be done unless the product $\_ \times^{\mathscr{D}} \_$ of $\mathscr{D}$ extends to $\mathscr{C}$ as a functor $\_ \otimes \_$. In this case, we can define $[\![(\mathsf{t}_1, \mathsf{t}_2)]\!]$ to be the morphism

$$[\![\Gamma]\!] \xrightarrow{\text{diag}} [\![\Gamma]\!] \times^{\mathscr{D}} [\![\Gamma]\!] \xrightarrow{[\![\mathsf{t}_1]\!] \otimes [\![\mathsf{t}_2]\!]} [\![\mathsf{b}_1]\!] \times^{\mathscr{D}} [\![\mathsf{b}_2]\!]$$

in $\mathscr{C}$.

Definition 17. *Let $\mathscr{C}$ be a category equipped with a structure of computational partiality $\mathscr{D}$. Suppose further that $\mathscr{D}$ has finite products, and that these extend to $\mathscr{C}$. An interpretation $[\![\_]\!]$ of (a language supporting) GADTs in $(\mathscr{C}, \mathscr{D})$ associates to each closed type a an object $[\![\mathsf{a}]\!]$ of $\mathscr{C}$, and to each term t of type a in context Γ comprising variables of types $\mathsf{a}_1, \ldots, \mathsf{a}_n$, a morphism $[\![\mathsf{t}]\!] : \prod_n^{\mathscr{D}} [\![\mathsf{a}_i]\!] \to [\![\mathsf{a}]\!]$ in $\mathscr{C}$ such that $[\![\mathsf{t}]\!]$ is in $\mathscr{D}$ when t is terminating. We require that $[\![\_]\!]$ maps the unit type $\top$ to 1 and term-substitution to composition in $\mathscr{C}$. Given a n-ary GADT G, a functor $[\![\mathsf{G}]\!] : \mathscr{C}^n \to \mathscr{C}$ is said to manifest G relative to $[\![\_]\!]$ if the action of $[\![\mathsf{G}]\!]$ on every object of the form $([\![\mathsf{a}_1]\!], \ldots, [\![\mathsf{a}_n]\!])$ is precisely $[\![\mathsf{G}\ \mathsf{a}_1 \ldots \mathsf{a}_n]\!]$.*

When $\mathscr{C}$ is *Set* (or any other locally presentable category), Definition 17 recovers the functorial semantics for GADTs in *Set* (or, more generally, in $\mathscr{C}$) used in Sections 2 through 5 by taking $\mathscr{D}$ to be $\mathscr{C}$ as in Example 14. This perfectly captures the fact that all morphisms are total in that semantics.

We can now prove that, like functorial interpretations of GADTs in *Set*, functorial interpretations of GADTs in more general categories equipped with structures of computational partiality are also insufficient.

Theorem 18. *Let $\mathscr{C}$ be a category equipped with structure of computational partiality $\mathscr{D}$. Suppose $[\![\_]\!]$ is an interpretation of GADTs in $(\mathscr{C}, \mathscr{D})$ relative to which each GADT is manifested by a functor. Then $[\![\mathsf{a}]\!] \simeq 1$ for all closed types* a *containing terminating terms.*

Proof. Among the GADTs in our language is the GADT Eq defined in (8). In addition to the function eqElim in Example 2, we can also use the recursion rule for GADTs to define its companion function

$$\begin{aligned} &\mathsf{eqElim}^{-1} :: \mathsf{Eq\ a\ b} \to \mathsf{a} \to \mathsf{b} \\ &\mathsf{eqElim}^{-1}\ \mathsf{Refl\ y} = \mathsf{y} \end{aligned}$$

Instantiating a and b to the closed types a_1 and a_2, respectively, the anonymous function

$$\lambda \mathsf{y} \to \mathsf{eqElim\ Refl}\ (\mathsf{eqElim}^{-1}\ \mathsf{Refl\ y})$$

reduces to the identity function on type a_1. By the uniqueness property of functions defined over GADTs given in (7),

$$\lambda \mathsf{p\ y} \to \mathsf{eqElim\ p}\ (\mathsf{eqElim}^{-1}\ \mathsf{p\ y})$$

reduces to the identity function on type a_1 for any input p. Semantically this entails that, if φ_a is the canonical isomorphism $a \simeq 1 \times a$, and if $p : 1 \to [\![\mathsf{Eq\ a_1 a_2}]\!]$ is any total function, then

$$[\![\mathsf{eqElim}]\!] \circ (p \times id_{[\![\mathsf{a}_2]\!]}) \circ \varphi_{[\![\mathsf{a}_2]\!]} \circ [\![\mathsf{eqElim}^{-1}]\!] \circ (p \times id_{[\![\mathsf{a}_1]\!]}) \circ \varphi_{[\![\mathsf{a}_1]\!]} \ = \ id_{[\![\mathsf{a}_1]\!]}$$

Now, let a be a closed type and let t be a terminating closed term of type a. We abuse notation and write $[\![\mathsf{t}]\!] : 1 \to [\![\mathsf{a}]\!]$ for the total morphism $[\![\lambda\_ \to \mathsf{t}]\!]$. Since every morphism with domain 1 in $\mathscr{D}$ is a split monomorphism, so is $[\![\mathsf{t}]\!]$. Thus, $([\![\mathsf{t}]\!], id_1)$ is a split monomorphism as well. Moreover, since there is a functor $[\![\mathsf{Eq}]\!] : \mathscr{C}^2 \to \mathscr{C}$ that manifests Eq relative to $[\![\_]\!]$, and since split monomorphisms are preserved by all functors, $[\![\mathsf{Eq}]\!]([\![\mathsf{t}]\!], id_1)$ must also be a split monomorphism. By Lemma 12, $[\![\mathsf{Eq}]\!]([\![\mathsf{t}]\!], id_1)$ is a total morphism from $[\![\mathsf{Eq\ 1\ 1}]\!]$ to $[\![\mathsf{Eq\ a\ 1}]\!]$. Consider the following morphisms:

$$\begin{aligned} s &= [\![\mathsf{eqElim}]\!] \circ (p \times id_1) \circ \varphi_1 && : 1 \to [\![\mathsf{a}]\!] \\ r &= [\![\mathsf{eqElim}^{-1}]\!] \circ (p \times id_{[\![\mathsf{a}]\!]}) \circ \varphi_{[\![\mathsf{a}]\!]} && : [\![\mathsf{a}]\!] \to 1 \end{aligned}$$

The observation at the end of the previous paragraph instantiated with $\mathsf{a}_1 = \mathsf{a}$, $\mathsf{a}_2 = 1$, and p being the total morphism $[\![\mathsf{Eq}]\!]([\![\mathsf{t}]\!], id_1) \circ [\![\mathsf{Refl}]\!]$ shows that $sr = id_{[\![\mathsf{a}]\!]}$. The composition rs is necessarily id_1 because it is total and 1 is terminal in $\mathscr{D}$. This explicitly gives that $[\![\mathsf{a}]\!] \simeq 1$, as announced in the statement of the theorem. $\square$

7. Conclusion and Related Work

The first part of this paper shows that GADTs do not have satisfactory IAS as functors on *Set*: the functorial completion semantics, which would give the smallest functorial IAS for them, do not have the expected parametricity properties so no others do either. Recognizing that the underlying reason for this is that GADTs' map functions are inherently partial naturally led us to consider analogous semantics on *PSet*. But we have shown in the second part of the paper that, unfortunately, GADTs do not have IAS interpretations as functors on *PSet* either. These results show that if we hope to find IAS interpretations for GADTs as functors on some category, and if we hope that the resulting IAS will specialize to the standard one for ADTs and nested types, then we will have to look in categories far more esoteric than *Set* and *PSet*.

The fundamental obstruction we have exposed in Section 6 is that GADTs' map functions are partial in ways that are not compatible with composition. Indeed, as Theorem 18 shows, inconsistencies arise when a composition can be mapped over an element of a GADT even though the first function in the composition cannot. These kinds of pitfalls can be avoided either (i) by abandoning the partiality modeled by categories equipped with structures of computational partiality, or (ii) by abandoning the classical notion of functoriality. But (i) seems neither feasible nor desirable, since categories equipped with structures of computational partiality appear to capture exactly the computationally relevant notion of partiality it needs to. On the other hand, (ii) involves changing the notion of compositionality for GADTs' map functions so that, in Theorem 18, a GADT's interpretation needn't send the composition of t_1 and t_2 to the composition of their images under that interpretation. The challenge here is that if G is an n-ary GADT then the interpretation of G cannot simply be a functor. It must still have actions on objects and morphisms of $\mathscr{C}^n$ like functors do, and must still send identities to identities, but it need not respect composition. Instead, the image of the composition of t_1 and t_2 under G's interpretation must only be "more defined" than the composition of the images of t_1 and t_2 under that interpretation. We will therefore consider, in future work, semantics in categories equipped with an ordering on morphisms and such that the interpretations of GADTs are normal lax functors instead of functors. Jay (Jay 1991) lays out the basic theory of such categories and lax functors that we plan to exploit to define the kind of semantics for GADTs we seek.

There are treatments of GADTs beyond those discussed in the main body of this paper. Atkey's parametric model for F_ω from (Atkey 2012) represents data types – including GADTs – as Church encodings. It requires the user to supply a map function for the (higher-order) type constructor whose fixpoint characterizes the data type. But, importantly, functoriality of an underlying type constructor does not imply functoriality of its fixpoint, so the data type itself still need not necessarily support a map function in Atkey's model. Similarly, (Vytiniotis and Weirich 2010) present a parametric model for an extension of F_ω that supports type equality and thus can encode GADTs, but this model still does not guarantee functoriality; accordingly, the parametric properties of GADTs described in the precursor work (Vytiniotis and Weirich 2006) to (Vytiniotis and Weirich 2010) are all inhabitation results rather than naturality results. In (Mandelbaum and Stump 2009) GADTs are represented as Scott encodings rather than Church encodings but, again, only inhabitation results are cited for them. GADTs are treated explicitly as fixpoints of discrete functors in (Johann and Ghani 2008), as initial algebras of dependent polynomial functors in (Gambino and Hyland 2004; Hamana and Fiore 2011), and as indexed containers in (Morris and Altenkirch 2009). The latter two treatments move toward seeing GADTs as data types in a dependent type theory. A categorical parametric model of dependent types has been given by (Atkey et al. 2014), but, as with the models mentioned above, this model also does not guarantee that GADTs have functorial semantics.

Acknowledgment. This work was supported by National Science Foundation awards 1713389 and 2203217.

Notes

1 The issue here is not a meta-theoretic one (like the size issue in Reynolds' failed model in *Set* for System F (Reynolds 1983)), but is more fundamental: even when the host language without GADTs admits a parametric model in *Set*, the very presence of GADTs still breaks parametricity.

2 Although our development applies to any language, we will use Haskell-like syntax for the code in this paper.

3 We write map_T or simply map when T is clear from context, for the function fmap :: (a → b) → (T a → T b) witnessing that a type constructor T is an instance of Haskell's Functor class. With this convention, $\mathsf{map}_\mathsf{List}$ coincides exactly with Haskell's built-in function map for lists. We emphasize that map functions in Haskell are intended to satisfy syntactic reflections of the functor laws – i.e., preservation of identity functions and composition of functions – even though this is not enforced by the compiler and is instead left to the good intentions of the programmer.

4 In the rest of the paper, we will refer to the *least fixpoints* of endofunctors simply as *fixpoints*. Other fixpoints are of no interest to us because they are not carriers of initial algebras.

5 Throughout this paper, we use `sans serif` font for program text and *math italic* font for semantic objects.

6 The map function for H is intended to satisfy syntactic reflections of the functor laws in Set^{Set} – i.e., preservation of identity natural transformations and composition of natural transformations – and the map function for H F is intended to satisfy syntactic reflections of the functor laws in *Set*, even though there is no mechanism in Haskell for enforcing this.

7 The notation G[a] indicates that G is a type with one hole which has been filled with the type a.

References

Adámek, J. and Rosický, J. (1994). *Locally Presentable and Accessible Categories*, London Mathematical Society Lecture Note Series, vol. **189**. Cambridge University Press.

Ahmed, A., Dreyer, D. and Rossberg, A. (2009) State-dependent representation independence. In: *Principles of Programming Languages*, vol. **44**, Association for Computing Machinery, 340–353.

Atkey, R. (2012). Relational Parametricity for Higher Kinds. In: Cégielski, P. and Durand, A., Leibniz International Proceedings in Informatics (LIPIcs), *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*, Dagstuhl, Germany, vol. **16**, 46–61, Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Atkey, R., Ghani, N. and Johann, P. (2014). A relationally parametric model of dependent type theory. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, Association for Computing Machinery, 503–516.

Barendregt, H. P. (1984) *The lambda calculus: its syntax and semantics*, *Studies in Logic and the Foundations of Mathematics*, vol. **103**, Elsevier.

Bird, R. and Meertens, L. (1998) Nested datatypes. In: *Mathematics of Program Construction. Lecture Notes in Computer Science*, Jeuring, J. (ed.), vol. **1422**, Berlin Heidelberg: Springer, 52–67.

Burn, G. L., Hankin, C. and Abramsky, S. (1986). Strictness analysis for higher-order functions. *Science of Computer Programming* **7** 249–278.

Bush, M. R., Leeming, M. and Walters, R. F. C. (2003). Computing left Kan extensions. *Journal of Symbolic Computation*. **35** (2) 107–126.

Carboni, A. (1987). Bicategories of partial maps. *Cahiers de topologie et géométrie différentielle catégoriques*. **28** (2) 111–126.

Cheney, J. and Hinze, R. (2003). First-class phantom types, CUCIS TR2003-1901.Cornell University.

Cockett, J. R. B. and Lack, S. (2002). Restriction categories I: Categories of partial maps. *Theoretical Computer Science*. **270** (1-2) 223–259.

Curien, P.-L. and Obtułowicz, A. (1989). Partiality, cartesian closedness, and toposes. *Information and Computation*. **270** (1) 50–95.

Dreyer, D., Neis, G. and Birkedal, L. (2012). The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming* **22** (4-5) 477–528.

Gambino, N. and Hyland, M. (2004) Well-founded trees and dependent polynomial functors. *In: Types for Proofs and Programs (TYPES 2023)*, Berardi, S., Coppo, M. and Damiani, F., vol. **3085**, *Lecture Notes in Computer Science*. Berlin Heidelberg: Springer, 210–225

Geuvers, H. (2014). The Church-Scott representation of inductive and coinductive data. *Unpublished.*

Hamana, M. and Fiore, M. (2011). A foundation for GADTs and inductive families: Dependent polynomial functor approach. In: *In: Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming, WGP '11*, Association for Computing Machinery, 59–70.

Hinze, R. (2003) Fun with phantom types. *In: The Fun of Programming*, 245–262.

Hur, C.-K. and Dreyer, D. (2011). A Kripke Logical Relation between ML and Assembly. *In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, New York, NY, USA: Association for Computing Machinery, vol. **11**, 133–146.

Jay, C. B. (1991) Partial functions, ordered categories, limits and cartesian closure. *In: IV Higher Order Workshop, Banff 1990, Workshops in Computing*, Birtwistle, G. (ed.), London: Springer London, pp. 151–161.

Johann, P. (2002). A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation* **,15** (4) 273–300.

Johann, P. and Cagne, P. (2022) Characterizing functions Mappable over GADTs. In: *Programming Languages and Systems*, Sergey, I. (ed.), Cham: Springer Nature Switzerland, 135–154.

Johann, P. and Ghani, N. (2007) Initial algebra semantics is enough!, *Typed Lambda Calculus and Applications*, vol. **4583**. Berlin Heidelberg: Springer, 207–222, Lecture Notes in Computer Science.

Johann, P. and Ghani, N. (2008). Foundations for structured programming with GADTs. In: ACM SIGPLAN Notices, *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*, ACM SIGPLAN Notices, New York, NY, USA: Association for Computing Machinery, vol. **43**, 297–308.

Johann, P. and Ghiorzi, E. (2022). (Deep) induction for GADTs. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2022*, Association for Computing Machinery. 324–337.

Johann, P., Ghiorzi, E. and Jeffries, D. (2021). Parametricity for primitive nested types. In: *Foundations of Software Science and Computation Structures: 24th International Conference, FOSSACS 2021*, Springer-Verlag. 324–343.

Johann, P. and Polonsky, A. (2019). Higher-Kinded Data Types: Syntax and Semantics. In: *34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 1–13.

Johann, P. and Polonsky, A. (2020). Deep induction: Induction rules for (truly) nested types. In: *Foundations of Software Science and Computation Structures: 23rd International Conference, FOSSACS 2020*, Springer-Verlag. 339–358.

Koopman, P., Plasmeijer, R. and Jansen, J. M. (2014). Church encoding of data types considered harmful for implementations: Functional pearl. In: *Proceedings of the 26th 2014 International Symposium on Implementation and Application of Programming Languages, IFL '14*, 1–12.

MacLane, S. (1971). *Categories for the Working Mathematician*, Springer-Verlag.

Mandelbaum, Y. and Stump, A. (2009). GADTs for the OCaml masses. *Unpublished*.

Manes, E. G. and Arbib, M. A. (1986). *Algebraic Approaches to Program Semantics. Monographs in Computer Science*, Springer New York.

McBride, C. (1999). Dependently typed programs and their proofs, PhD thesis, University of Edinburgh.

Minsky, Y. (2015). Why GADTs matter for performance. Available at https://blog.janestreet.com/why-gadts-matter-for-performance/.

Morris, P. and Altenkirch, T. (2009). Indexed containers. In: *24th Annual IEEE Symposium on Logic In Computer Science, LICS 2009*, 277–285.

nLab AUTHORS (2019). Transfinite construction of free algebras. Available at http://ncatlab.org/nlab/show/transfinite+construction+of+free+algebras.

Norell, U. (2022). Russell's paradox in agda. Available at https://github.com/agda/agda/blob/master/test/Succeed/Russell.agda.

Pasalic, E. and Linger, N. (2004) Meta-programming with typed object-language representations. In: *International Conference on Generative Programming and Component Engineering*, Gabor Karsai, E. V., vol. **3286**, Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 136–167

Penner, C. (2020). Simpler and safer API design using GADTs. Available at https://chrispenner.ca/posts/gadt-design.

Peyton Jones, S. L., Vytiniotis, D., Washburn, G. and Weirich, S. (2006). Simple unification-based type inference for GADTs. In: *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, ICFP '06*, Association for Computing Machinery, 50–61.

Pierce, B. C. (2002). *Types and Programming Languages*, The MIT Press.

Pottier, F. and Régis-Gianas, Y. (2006). Stratified type inference for generalized algebraic data types. In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, Association for Computing Machinery, vol. **06**, 232–244.

Reynolds, J. C. (1983). Types, abstraction, and parametric polymorphism. *Information Processing*. **83** (1) 513–523.

Riehl, E. (2016). Category theory in context. Dover..

Robinson, E. and Rosolini, G. (1988). Categories of partial maps. *Information and Computation*. **79** (2) 95–130.

Schrijvers, T., Jones, S. L. P., Sulzmann, M. and Vytiniotis, D. (2009). Complete and decidable type inference for GADTs. In: *International Conference on Functional Programming*, 341–352.

Sheard, T. and Pasalic, E. (2004) Meta-programming with built-in type equality. In: *Workshop on Logical Frameworks and Meta-languages*, 106–124.

Vytiniotis, D. and Weirich, S. (2006). Parametricity and GADTs. Available at https://www.cis.upenn.edu/~sweirich/talks/param-gadt.pdf.

Vytiniotis, D. and Weirich, S. (2010). Parametricity, type equality, and higher-order polymorphism. *Journal of Functional Programming* **20** (2) 175–210.

Wadler, P. (1989). Theorems for free! In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA*, Association for Computing Machinery, vol. **89**, 347–359.

Cite this article: Cagne P, Ghiorzi E and Johann P (2024). GADTs are not (Even partial) functors. *Mathematical Structures in Computer Science*. https://doi.org/10.1017/S0960129524000161