GADTs, Functoriality, Parametricity: Pick Two

Patricia Johann, Enrico Ghiorzi, and Daniel Jeffries
Appalachian State University
johannp@appstate.edu, ghiorzie@appstate.edu, jeffriesd@appstate.edu

GADTs can be represented either as their Church encodings *à la* Atkey, or as fixpoints *à la* Johann and Polonsky. While a GADT represented as its Church encoding need not support a `map` function satisfying the functor laws, the fixpoint representation of a GADT must support such a `map` function even to be well-defined. The two representations of a GADT thus need not be the same in general. This observation forces a choice of representation of data types in languages supporting GADTs. In this paper we show that choosing whether to represent data types as their Church encodings or as fixpoints determines whether or not a language supporting GADTs can have parametric models. This choice thus has important consequences for how we can program with, and reason about, these advanced data types.

1 Introduction

There are two standard ways to represent data types when studying modern functional languages: *i*) as their Church encodings in a (possibly higher-kinded) extension of System F [9], the calculus at the core of all such languages, and *ii*) by augmenting System F with primitives for constructing them directly as fixpoints. Models in which the Church encoding and the fixpoint representation of every algebraic data type (ADT) are semantically equivalent include the operationally-based model of [29, 30] and the categorical models of [13, 14, 18]. In the categorical model of [18] this semantic equivalence also holds for the syntactic generalization of ADTs known as nested types [4]. Generalized algebraic data types (GADTs) [21] generalize nested types — and thus further generalize ADTs — syntactically (see Section 3 below):

$$\boxed{\text{ADTs}} \xrightarrow[\text{generalized by}]{\text{syntactically}} \boxed{\text{nested types}} \xrightarrow[\text{generalized by}]{\text{syntactically}} \boxed{\text{GADTs}}$$

This sequence of syntactic generalizations suggests that a corresponding sequence of semantic generalizations might also hold. However, this is not the case. Specifically, the semantic equivalence between Church encodings and fixpoint representations that holds for ADTs and nested types need not hold for GADTs. In a language supporting GADTs we must therefore choose whether to represent data types as their Church encodings or as fixpoints. The main result of this paper shows that the choice of representation of data types in a language supporting GADTs determines whether or not that language can have parametric models [32]. It thus determines whether or not GADTs can enjoy consequences of parametricity such as representation independence [1, 6], equivalences between programs [12], deep induction principles [17, 20], and useful ("free") theorems about programs derived from their types alone [37]. This result is unintuitive, novel, and surprising.

Consider the GADT Seq defined by[1]

$$\begin{array}{l}\mathsf{data}\ \mathsf{Seq}\ \mathsf{a}\ \mathsf{where}\\ \quad \mathsf{Const} :: \mathsf{a} \to \mathsf{Seq}\ \mathsf{a}\\ \quad \mathsf{Pair}\ \ :: \mathsf{Seq}\ \mathsf{a} \to \mathsf{Seq}\ \mathsf{b} \to \mathsf{Seq}\ (\mathsf{a} \times \mathsf{b})\end{array} \tag{1}$$

[1] Although our development applies to GADTs in any language, we will use Haskell syntax for the code in this paper.

Submitted to:
LSFA 2021

This GADT comprises sequences of any type a and sequences obtained by pairing the data in two already existing sequences. When represented as its Church encoding, this GADT contains no data elements other than these. More generally, a GADT's Church encoding contains exactly those data elements that are representable by its syntax. By contrast, the fixpoint representation of a GADT sees that data type as the *functorial completion* [19] of its syntax. Completing a GADT's syntax from a funct*ion* on types to a funct*or* on types is necessary if its interpretation is to be a fixpoint [34], so functoriality is inherent in fixpoint representations of data types. Since being a functor entails supporting a map function satisfying the functor laws, the fixpoint representation of a data type must include the entire "map closure" of its syntax. For example, the fixpoint representation of Seq contains not just those data elements representable by its syntax, but also all data elements of the form $\mathsf{map}\,\mathsf{f}\,\mathsf{s}$ 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 all data elements of the form $\mathsf{map}\,\mathsf{g}\,\mathsf{s}'$ 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, data elements of the form $\mathsf{map}\,\mathsf{g}\,(\mathsf{Pair}\,\mathsf{u}_1\,\mathsf{u}_2)$ even though these may not themselves be of the form $\mathsf{Pair}\,\mathsf{v}_1\,\mathsf{v}_2$ for any terms v_1 and v_2. Importantly, functorial completion adds no new data elements to the syntax of a GADT that is an ADT or other nested type, which perfectly explains why Church encodings and fixpoint representations coincide for ADTs and other nested types. However, the two representations are not the same for GADTs that are not nested types.

Even so, does it really matter how we represent GADTs?

A GADT programmer is likely to use GADTs precisely *because* they exhibit different behaviors at different types, and thus to consider a GADT to be completely specified by its syntax. When used this way, the shape of a particular element of a GADT is actually *determined* by the data it contains. As a result, it cannot be expected to support a functorial map function like ADTs and nested types. Indeed, the definition for Seq above specifies that an element of the form $\mathsf{Pair}\,\mathsf{u}_1\,\mathsf{u}_2$ must have the shape of a sequence of data of pair type rather than a sequence of data of arbitrary type e. The clause of map for the Pair constructor should therefore feed map a function $\mathsf{f} :: (\mathsf{a} \times \mathsf{b}) \to \mathsf{e}$ and a term of the form $\mathsf{Pair}\,\mathsf{u}_1\,\mathsf{u}_2$ for $\mathsf{u}_1 :: \mathsf{Seq}\,\mathsf{a}$ and $\mathsf{u}_2 :: \mathsf{Seq}\,\mathsf{b}$, and produce a term $\mathsf{Pair}\,\mathsf{v}_1\,\mathsf{v}_2$ for some appropriately typed terms v_1 and v_2. However, it is not clear how to achieve this since e need not necessarily be a product type. And even if e *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 u_1 and u_2 unless we knew, e.g., that f was a pair of functions $(\mathsf{f}_1 :: \mathsf{a} \to \mathsf{w}, \mathsf{f}_2 :: \mathsf{b} \to \mathsf{z})$. When it is intended to capture this kind of non-uniformity, a GADT cannot be regarded as a data-independent container that can be filled with data of any type in the same way that ADTs and nested types can. In this situation, a GADT cannot denote a functor, and so must necessarily be represented as its Church encoding.

A semanticist, on the other hand, is likely to expect GADTs to generalize ADTs semantically — i.e., to have the same kind of functorial semantics that ADTs and nested types have [4, 15, 19] — and thus to be useable as a container filled with data that can be changed without changing the shape of the container itself. For a GADT to be used in this way, its syntax must reflect functoriality. Since the Church encoding of a GADT that is not a nested type does not denote a functor, to have a functorial semantics such a GADT must be viewed as its fixpoint representation. The functorial completion inherent in the fixpoint representation of a GADT adds to its syntax those, and only those, data elements needed to support a functorial map function. For example, let 1 be the unit type whose single element is also denoted by 1, and let G be the GADT defined by

$$\begin{array}{c}\mathsf{data}\,\mathsf{G}\,\mathsf{a}\,\mathsf{where}\\ \mathsf{C} :: \mathsf{G}\,1\end{array} \tag{2}$$

Then the functorial completion of G includes elements at any instance $\mathsf{G}\,\mathsf{a}$ for any type a that is not the

empty type 0 (since there is always a function from 1 to such an a), but includes no elements at instance $G\,0$. Indeed, $G\,0$ is not inhabited via the syntactic specification of G, and is not inhabited via functorial completion because it is not possible to define a function from 1 into 0.

The key observation of this paper is that, while the viewpoints of the GADT programmer and the semanticist are both valid, the two are irreconcilable. Importantly, which of our two representations of data types is adopted in any particular setting has significant consequences for the ways GADTs can be used and reasoned about there. In particular, the way that GADTs are represented has deep implications for parametric reasoning about them. Specifically, a programmer who views GADTs as their Church encodings cannot safely use program transformations or reasoning principles that involve map functions for them, although they may be able to program with and reason about GADTs using other consequences of parametricity, such as type inhabitation results. On the other hand, a semanticist who views GADTs as fixpoints will have all naturality-based program transformations and reasoning principles for GADTs at their disposal since these all derive from functoriality. But since, as we show in Section 5, no parametric model can be constructed for fixpoint representations of GADTs, non-naturality consequences of parametricity will not necessarily hold for them. Overall, we show that, as with software engineering's iron triangle [7], we can have any two of *GADTs*, *functoriality*, and *parametricity* we like, but we cannot have all three.

The goal of this paper is to show how the above observations can be made precise, and thereby to answer the question we posed above in the affirmative: Yes, it really does matter how we represent GADTs.

2 Representations of Algebraic Data Types

A (polynomial) *algebraic data type* (ADT) has the form

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

where each 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* $C_1, \ldots, C_n$. 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 examples of ADTs. The shape of the container underlying the type

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

is determined by the types of its two constructors $\mathsf{Nil} :: \mathsf{List}\,a$ and $\mathsf{Cons} :: a \to \mathsf{List}\,a \to \mathsf{List}\,a$. 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 are given by natural numbers ranging from 0 to $n-1$. 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

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

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 $\mathsf{Tree}\,\mathsf{a}$ 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. Moreover, every ADT T can be made an instance of Haskell's $\mathsf{Functor}$ class by defining a type-and-data-uniform, structure-preserving, data-changing function $\mathsf{map}_{\mathsf{T}}$ for it.[2] Then, given a type-independent way of rearranging an ADT structure's shape $\mathsf{T}\,\mathsf{a}$ into the shape for another ADT structure $\mathsf{T}'\,\mathsf{a}$, we get the same structure of type $\mathsf{T}'\,\mathsf{b}$ regardless of whether we first rearrange the original structure of type $\mathsf{T}\,\mathsf{a}$ into one of type $\mathsf{T}'\,\mathsf{a}$ and then use $\mathsf{map}_{\mathsf{T}'}$ to convert that resulting structure to one of type $\mathsf{T}'\,\mathsf{b}$, or we first use $\mathsf{map}_{\mathsf{T}}$ to convert the original structure of type $\mathsf{T}\,\mathsf{a}$ to one of type $\mathsf{T}\,\mathsf{b}$ and then rearrange that resulting structure into one of type $\mathsf{T}'\,\mathsf{b}$. For example, if $\mathsf{f} :: \mathsf{a} \to \mathsf{b}$, $\mathsf{t} :: \mathsf{Tree}\,\mathsf{a}$, and $\mathsf{g} :: \mathsf{Tree}\,\mathsf{a} \to \mathsf{List}\,\mathsf{a}$ 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 Church Encodings of ADTs

One way to represent ADTs is as their Church encodings. A Church encoding is a representation of a data type as a function in a pure lambda calculus, such as System F and its higher-kinded extensions. They, together with other related encodings, have recently been popularized as various *visitor patterns* in object-oriented programming [10, 26].

Church encodings of ADTs can be defined in any language that supports functions. They can therefore be used to represent ADTs in languages that do not support primitives for sum types, product types, or recursion. The Church encodings of the ADTs $\mathsf{List}\,\mathsf{a}$ and $\mathsf{Tree}\,\mathsf{a}$, for example, are

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

and

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

respectively. The argument types in a Church encoding of an ADT are abstractions of the types of the ADT's constructors. For instance, b abstracts the type $\mathsf{List}\,\mathsf{a}$ of the constructor Nil for lists, and $\mathsf{a} \to \mathsf{b} \to \mathsf{b}$ abstracts the type $\mathsf{a} \to \mathsf{List}\,\mathsf{a} \to \mathsf{List}\,\mathsf{a}$ of the constructor Cons.

Because the types of the "abstract constructors" for an ADT are uniform in their argument types, it is always possible to Church encode the type constructors themselves as well. For example, the Church encodings of the type constructors List and Tree are

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

and

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

[2] We write $\mathsf{map}_{\mathsf{T}}$, or simply map when T is clear from context, for the function $\mathsf{fmap} :: (\mathsf{a} \to \mathsf{b}) \to (\mathsf{T}\,\mathsf{a} \to \mathsf{T}\,\mathsf{b})$ witnessing that a type constructor T is an instance of Haskell's $\mathsf{Functor}$ class. We emphasize that fmap 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.

respectively. This fact is what allows an ADT's associated type constructor to be made an instance of Haskell's Functor class. For example, the map function from Haskell's standard library makes the type constructor List an instance of the Functor class, and for Tree we can define

$$\begin{array}{lcl} \mathsf{map}_{\mathsf{Tree}} & :: & (\mathsf{a} \to \mathsf{b}) \to \mathsf{Tree}\,\mathsf{a} \to \mathsf{Tree}\,\mathsf{b} \\ \mathsf{map}_{\mathsf{Tree}}\,\mathsf{f}\,(\mathsf{Leaf}\,\mathsf{x}) & = & \mathsf{Leaf}\,(\mathsf{f}\,\mathsf{x}) \\ \mathsf{map}_{\mathsf{Tree}}\,\mathsf{f}\,(\mathsf{Node}\,\mathsf{t}_1\,\mathsf{x}\,\mathsf{t}_2) & = & \mathsf{Node}\,(\mathsf{map}_{\mathsf{Tree}}\,\mathsf{f}\,\mathsf{t}_1)\,(\mathsf{f}\,\mathsf{x})\,(\mathsf{map}_{\mathsf{Tree}}\,\mathsf{f}\,\mathsf{t}_2) \end{array}$$

Note that the Church encoding of an ADT carries with it no expectation whatsoever that such a type-and-data-uniform, structure-preserving, data-changing map function can be defined or that, if one can, it will satisfy a rearrange-transform property. (Of course, such a map function can always be defined for any ADT precisely because its Church encoding and its fixpoint representation coincide, and this map function will necessarily satisfy a rearrange-transform property.)

2.2 ADTs as Fixpoints

By contrast, the ability to define such a map function is inherent in the view of ADTs as fixpoints. Such a view is possible in any language that supports primitives for sum types, product types, and recursion. In such a language, the fixpoint representations of the ADTs List a and Tree a are

$$\mathsf{List}\,\mathsf{a} = \mu\mathsf{X}.\,1 + \mathsf{a} \times \mathsf{X} \tag{3}$$

and

$$\mathsf{Tree}\,\mathsf{a} = \mu\mathsf{X}.\,\mathsf{a} + \mathsf{X} \times \mathsf{a} \times \mathsf{X} \tag{4}$$

respectively, where μ is a primitive fixpoint operator.

Fixpoint representations capture in syntax the fact that ADTs can be considered as fixpoints of functors. For example, List a is indeed a fixpoint of $\mathsf{F}_{\mathsf{List}\,\mathsf{a}}$, where $\mathsf{F}_{\mathsf{List}\,\mathsf{a}}\,\mathsf{X} = 1 + \mathsf{a} \times \mathsf{X}$ by (3). That is,

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

since every element of List a is either empty or is obtained by consing an element of type a onto an already-existing structure of type List a. In fact, this fixpoint equation is just a rewriting of the Haskell data type declaration for List a. We therefore have that $\mathsf{List}\,\mathsf{a} = \mu\mathsf{F}_{\mathsf{List}\,\mathsf{a}}$ is modeled by $\mu F_{List\,a}$, where the functor $F_{List\,a}$ models the type constructor $\mathsf{F}_{\mathsf{List}\,\mathsf{a}}$.[3] Similarly, the ADT Tree a can be seen to be a fixpoint of $\mathsf{F}_{\mathsf{Tree}\,\mathsf{a}}$, where $\mathsf{F}_{\mathsf{Tree}\,\mathsf{a}}\,\mathsf{X} = \mathsf{a} + \mathsf{X} \times \mathsf{a} \times \mathsf{X}$ by (4), so that $\mathsf{Tree}\,\mathsf{a} = \mu\mathsf{F}_{\mathsf{Tree}\,\mathsf{a}}$ is modeled by $\mu F_{Tree\,a}$ if $F_{Tree\,a}$ models $\mathsf{F}_{\mathsf{Tree}\,\mathsf{a}}$.

These kinds of fixpoint equations 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 semantic fixpoint calculation must converge. If, as is typical, we interpret our types as sets, then the fixpoint being taken must be of a funct*or* on the category *Set* of sets and functions between them, rather than of a mere funct*ion* between sets [34]. That is, the function F interpreting the type constructor F constructing the body of a syntactic fixpoint must be a functor, i.e., 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 map function satisfying the functor laws. That is, F must be an instance of Haskell's Functor class (with the aforementioned caveat about the functor laws).

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

Requiring F to be a functor ensures that the interpretation μF of the ADT $\mathsf{T}\,\mathsf{a} = \mu\mathsf{F}$ exists. But to ensure that μF is itself a functor, so that the type constructor T associated with $\mathsf{T}\,\mathsf{a}$ also supports its own map function, we can require that F be a functor on the category Set^{Set} of functors and natural transformations on *Set*. That is, F must 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 $\mathsf{T}\,\mathsf{a} = (\mu\mathsf{H})\,\mathsf{a}$ for the "type constructor constructor" H that supports suitable[4] map functions.

A concrete example is given by the ADT $\mathsf{List}\,\mathsf{a}$. 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 $F_{List\,a}\,f = 1 + id_a \times f$. 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.

In any parametric model, the Church encoding and the fixpoint representation of an ADT or a nested type will necessarily be semantically equivalent. But whereas it is impossible even to *state* the rearrange-transform property for their Church encodings (unless functorial map functions have somehow been defined for them), for their fixpoint representations such a property is simply the reflection back into syntax of the instance of naturality for the type-independent function that rearranges structures of type $\mathsf{T}\,\mathsf{a}$ into ones of type $\mathsf{T}'\,\mathsf{a}$ and the structure-preserving, data-changing functions $\mathsf{map}_{\mathsf{T}}\,\mathsf{f}$ and $\mathsf{map}_{\mathsf{T}'}\,\mathsf{f}$ for a function $\mathsf{f} :: \mathsf{a} \to \mathsf{b}$, where T and T' are the type constructors associated with these ADTs, respectively.

3 Representations of GADTs

Generalized algebraic data types (GADTs) [21] relax the restriction on the type instances appearing in a data type definition. The special form of GADTs known as *nested types* [4] allow 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 illlustrated by the definition

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

of the nested type $\mathsf{PTree}\,\mathsf{a}$ of perfect trees, which introduces the data constructors $\mathsf{PLeaf} :: \mathsf{a} \to \mathsf{PTree}\,\mathsf{a}$ and $\mathsf{PNode} :: \mathsf{PTree}\,(\mathsf{a} \times \mathsf{a}) \to \mathsf{PTree}\,\mathsf{a}$. It enforces not only the invariant that all of the data in a structure of type $\mathsf{PTree}\,\mathsf{a}$ is of the same type a, but also the invariant that all perfect trees have lengths that are powers of 2. GADTs that are not nested types 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 is the GADT Seq given in (1). Since the return type of the data constructor Pair is not of the form $\mathsf{Seq}\,\mathsf{a}$ for any variable a, Seq is a GADT that is not a nested type.

By contrast with the ADT $\mathsf{List}\,\mathsf{a}$, where the type parameter a is integral to the type being defined, the type parameter a appears in both $\mathsf{PTree}\,\mathsf{a}$ and $\mathsf{Seq}\,\mathsf{a}$ as a "dummy" parameter used only to give the

[4] 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 HF is intended to satisfy syntactic reflections of the functor laws in *Set*, even though there is no mechanism in Haskell for enforcing this.

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

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

and

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

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*.

GADTs that are not nested types 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. 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 [27, 31, 21], to improve memory performance [24], and to design APIs [28].

3.1 Church Encodings of GADTs

The syntax of GADTs allows non-variable type arguments in the return types of their data constructors. This 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 on the type of the data to be contained. For this, it is essential that a GADT calculus support an equality type Equal that is a singleton set when its two type arguments are the same and is the empty set otherwise. That is, the type Equal must be the syntactic reflection of semantic equality function *Equal*. The type Equal can be defined via GADT syntax as

$$\begin{aligned}&\mathsf{data}\,\mathsf{Equal}\,\mathsf{a}\,\mathsf{b}\,\mathsf{where}\\ &\quad \mathsf{Refl} :: \mathsf{Equal}\,\mathsf{c}\,\mathsf{c}\end{aligned}$$

Its Church encoding (in a higher-kinded calculus such as F_ω [2]) is

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

Following the technique in Section 4.5 of [2], we can rewrite its Church encoding as

$$\mathsf{Seq}\,\mathsf{a} = \forall \mathsf{f}.\,(\forall \mathsf{d}.\,(\mathsf{f}\,\mathsf{d} + \exists \mathsf{bc}.\,\mathsf{f}\,\mathsf{b} \times \mathsf{f}\,\mathsf{c} \times \mathsf{Equal}\,\mathsf{d}\,(\mathsf{b} \times \mathsf{c})) \to \mathsf{f}\,\mathsf{d}) \to \mathsf{f}\,\mathsf{a}$$

which is logically equivalent to the more intuitive encoding

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

Importantly, the function *Equal a* cannot be made into a functor. Equivalently, Equal a cannot be made an instance of Haskell's Functor class. Indeed, if Equal a supported a function

$$\mathsf{map}_{\mathsf{Equal}\,\mathsf{a}} :: (\mathsf{b} \to \mathsf{c}) \to \mathsf{Equal}\,\mathsf{a}\,\mathsf{b} \to \mathsf{Equal}\,\mathsf{a}\,\mathsf{c}$$

then defining

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

would allow us to construct an element

$$\mathsf{eqElim}\,(\mathsf{map}_{\mathsf{Equal}\,0}\,\mathsf{absurd}\,\mathsf{Refl})\,1$$

of the empty type 0, where absurd :: 0 → c is the empty function from 0 into c. But this is not possible.

Since the Church encoding of a GADT that is not a nested type involves the equality type, its map function must necessarily involve the equality type's map function. But since the equality type does not support a map function, it is immediate that a Church encoding of a GADT that is not a nested type cannot support a map function either. The underlying problem is illustrated for the GADT Seq in Section 1. In that setting, given a function f :: (a × b) → e, the term $\mathsf{map}_{\mathsf{Equal}\,(\mathsf{a}\times\mathsf{b})}\,\mathsf{f}\,\mathsf{Refl}$ would have type Equal (a × b) e. But we have no way to produce a term of this type in the absence of a functorial map function for Equal, and thus no way to produce a term of type Seq e using the Pair constructor, as would be required by the clause of $\mathsf{map}_{\mathsf{Seq}}$ for Pair.

3.2 GADTs as Fixpoints

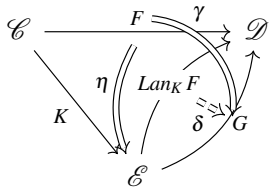
The Church encoding of a GADT corresponds to the data type comprising just those elements specified by the GADT's syntax. By contrast, the fixpoint representation of a GADT corresponds to the data type comprising all data elements in the functorial completion of the GADT's syntax. In this latter reading, 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, the use of (higher-order) *functors* is essential, as in the case of ADTs, to guarantee the existence of the (higher-order) fixpoints being computed [34].

To illustrate, 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 [19], 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 [22]:

Definition 1. *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*



To represent GADTs as fixpoints in a setting in which types are interpreted as sets, a calculus must support a primitive construct Lan such that the type constructor $\mathsf{Lan_K\,F}$ is the syntactic reflection of the left Kan extension $\mathsf{Lan}_K F$ of the functor F interpreting F along the functor K interpreting K. In this setting, the categories $\mathscr{C}$, $\mathscr{D}$, and $\mathscr{E}$ must all be of the form Set^n for some n. Using Lan we can then rewrite the type of a constructor $\mathsf{C} :: \mathsf{F\,a} \to \mathsf{G\,(K\,a)}$ as $\mathsf{C} :: \mathsf{(Lan_K\,F)\,a} \to \mathsf{G\,a}$ since 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 (see e.g., [33]). That is, writing $F \Rightarrow G$ for the set of natural transformations from a functor F to a functor G, we have

$$F \Rightarrow G \circ K \;\simeq\; Lan_K F \Rightarrow G \tag{5}$$

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. The 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, GADTs are generally underspecified by their syntax.

In this section we have seen that GADTs can be represented either as Church encodings or as fixpoints. This exactly mirrors the situation for ADTs and nested types described in Sections 2 and 3. But whereas the two representations are always semantically equivalent for ADTs and nested types, they are not, in general, semantically equivalent for GADTs. This will be shown formally in the next section.

4 Non-Equivalence of Church Encodings and Fixpoint Representations of GADTs

To see that the Church encoding of a GADT and its fixpoint representation need not be semantically equivalent, consider again the GADT G defined in (2). Despite its simplicity, this GADT serves as an informative case study highlighting the differences between a GADT's Church encoding and its fixpoint representation — even if we consider only the data elements it contains, and ignore whether or not it supports a map function.

Example 2. *Syntactically, the GADT* G *defined in* (2) *comprises a single data element, namely* $\mathsf{C} :: \mathsf{G}\,1$. *This is captured by* G*'s Church encoding* $\mathsf{G}\,\mathsf{a} = \forall \mathsf{f}.\,(\forall \mathsf{c}.\,\mathsf{Equal}\,1\,\mathsf{c} \to \mathsf{f}\,\mathsf{c}) \to \mathsf{f}\,\mathsf{a}$ *from [2], which is equivalent in the parametric model given there to*

$$\mathsf{G}\,\mathsf{a} = \mathsf{Equal}\,1\,\mathsf{a}$$

As the above equation makes clear, G*'s effect is simply to test its argument for equality against the unit type* 1. *The interpretation of the Church encoding of* G *is thus, as explained in Section 3.1, the function whose value is a singleton set when the interpretation of* a *is* 1 *and the empty set otherwise. This perfectly accords with* G*'s syntax, which indeed delivers just one data element when* a *is* 1 *and no elements otherwise.*

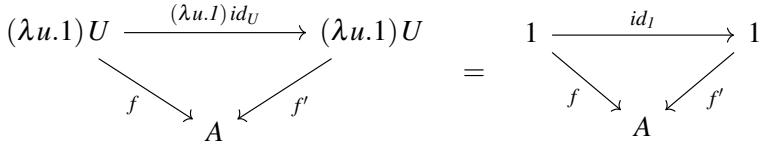
To compute the interpretation of G*'s fixpoint representation we first note that the type of* G*'s solitary constructor* $\mathsf{C} :: \mathsf{G}\,1$ *is equivalently expressed as* $\mathsf{C} :: 1 \to \mathsf{G}\,1$ *or, using* (5), *as* $\mathsf{C} :: (\mathsf{Lan}_{\lambda \mathsf{u}.1}\,\lambda \mathsf{u}.1)\,\mathsf{a} \to \mathsf{G}\,\mathsf{a}$, *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*

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

The interpretation of G *is therefore 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* $\mathsf{G}\,\mathsf{a}$ *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 [5] 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*



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.,* $(Lan_{\lambda u.1}\lambda u.1)A = A$. *This is different from the interpretation of* G*'s Church encoding from the previous paragraph.*

Putting it all together, we see that the Church encoding of G *and its fixpoint representation are not semantically equivalent: the interpretation of the former has exactly one data element at instance* $\mathsf{G}\,1$ *and no elements at any other instances, whereas the interpretation of the latter 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* $map_G\, f_a\, c \in GA$ *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.

Forced to choose, a programmer would likely find the idea that a 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? Why should a GADT contain "hidden" elements that are not specified by the GADT's syntax and are only accessible via applications of `map`? From a semanticist's point of view, however, the primitive representation of GADTs is entirely reasonable. Indeed, they would likely find the nonfunctorial nature of a GADT's Church encoding unnerving at best. After all, they would likely argue, the data in a GADT shouldn't change or become ill-typed just because a function is mapped over it. The fact that this happens when GADTs are represented by their Church encodings actually highlights how GADTs *do not* generalize the essential, container-ish nature of ADTs at all. A semanticist might therefore conclude that GADTs are seriously misnamed.

Because they are not semantically equivalent, the two representations of GADTs have very different implications for parametricity. We explore the differences in parametricity results for the two representations of GADTs in the next section.

5 Parametricity in the Presence of GADTs

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 [32] for System F. It was later popularized as Wadler's "theorems for free" [37], 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, and prove the equivalence of Church encodings and fixpoint representations of ADTs and nested types by validating short cut fusion and other program equivalences for them.

To discuss relational parametricity 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 3. *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 $(fa, gb) \in R'$ 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 $R : Rel(A,B)$ for $(A,B,R) \in Rel$. If $R : Rel(A,B)$ then we write $\pi_1 R$ and $\pi_2 R$ for the *domain* A and *codomain* B of R, respectively. We write $Eq_A = (A,A,\{(x,x) \mid x \in A\})$ for the *equality relation* on the set A.

The key idea underlying parametricity is to give each type `G[a]`[5] with one free variable `a` a *set interpretation* G_0 taking sets to sets and a *relational interpretation* G_1 taking relations $R : Rel(A,B)$ to

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

relations $G_1 R : Rel(G_0 A, G_0 B)$, and to interpret each term $\mathsf{t}(\mathsf{a},\mathsf{x}) :: \mathsf{G}[\mathsf{a}]$ with one free term variable $\mathsf{x} :: \mathsf{F}[\mathsf{a}]$ as a function t associating to each set A a morphism $tA : F_0 A \to G_0 A$ 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 $G_1 Eq_A = Eq_{G_0 A}$, 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 $R : Rel(A,B)$, (tA, tB) 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 $tA \in G_0 A$.

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 [2]. 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 $\forall$-types to just those elements that are "parametric" [32, 37] to ensure that the Identity Extension Lemma holds. If the set interpretation of Equal is the function *Equal* and if the relational interpretation of a closed type a with interpretation A is Eq_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 $\mathsf{G\,a}$ 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 (2) gives that $\mathsf{G\,a}$ contains a single data element if a is semantically equivalent to $\mathsf{1}$ and none otherwise:

Example 4. *Let* t *be a closed term of type* $\mathsf{G\,a}$ *for the GADT* G *defined in* (2)*, let* $G = (G_0, G_1)$ *be the interpretation of the Church encoding of* G*, let* t *be the interpretation of* t*, and let* $R : Rel(A,B)$*. Then* $tA \in G_0 A$ *and* $tB \in G_0 B$ *and, by the Abstraction Theorem (Theorem 3) in [2], tA and tB must be related in* $G_1 R$*. However, under the semantics given in [2], which includes the aforementioned interpretations of* Equal *and closed types, the relational interpretation* $G_1 R$ *of* G *is itself* Eq_1 *when* R *is the relational interpretation* Eq_1 *of* $\mathsf{1}$*, and the empty relation whenever* R *differs from* Eq_1*. Reflecting back into syntax we deduce that there can be no term in the type that is the Church encoding of* $\mathsf{G\,a}$ *unless* a *is semantically equivalent to* $\mathsf{1}$*.*

For fixpoint representations of GADTs the story is completely different. If, as intended, the set interpretation of $\mathsf{Lan_K\,F}$ 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 $\mathsf{Lan_K\,F}$ 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. It is the main technical contribution of this paper.

Example 5. *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* $R : Rel(A,B)$*, we have* $G_1 R : Rel(G_0 A, G_0 B)$*.*

Intuitively, when G *is viewed as a fixpoint, its data elements include those given by functorial completion. Since* G_1 *is a functor, given any relation* $S : Rel(C,D)$ *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 3 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_0A and G_0B, respectively. This suggests that GADTs might fail to have relational interpretation, and thus might fail to have a parametric model, 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 (2). The set interpretation of G is $Lan_{\lambda u.1}\,\lambda u.1$, i.e., is, by the reasoning of Example 2, the identity functor on Set. By the exact same reasoning, this time in Rel rather than in Set, the relational interpretation of G is $Lan_{\lambda u.Eq_1}\,\lambda u.Eq_1$, where $\lambda u.Eq_1$ is the constantly Eq_1-valued functor from the category Rel^0 with a single object to Rel. Indeed, the interpretation is still a left Kan extension, but now it is the left Kan extension determined by the functor interpreting the type $\lambda u.1$ in Rel. For the Identity Extension Lemma to hold we would need that, for every relation $R : Rel(A,B)$, we have that $(\mathsf{Lan}_{\lambda u.Eq_1}\,\lambda u.Eq_1)R$ is 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 $(Lan_{\lambda u.Eq_1}\,\lambda u.Eq_1)R$ to be a relation with domain 1. Since left Kan extensions preserve projections [33], we can compute the domain as

$$\pi_1\left((Lan_{\lambda u.Eq_1}\,\lambda u.Eq_1)\,R\right) \;=\; (Lan_{\lambda u.Eq_1}\,\lambda u.1)\,R$$

(Note that the left Kan extension of $\lambda u.1$ along $\lambda u.Eq_1$ is a functor from Rel to Set.) By the same reasoning as in Example 2, Proposition 7.1 of [5] gives that $(Lan_{\lambda u.Eq_1}\,\lambda u.1)\,R$ can be computed as

$$\Big(\bigcup_{U:Rel^0,\,m:(\lambda u.Eq_1)U\to R} (\lambda u.1)\,U\Big)/\approx \;=\; \Big(\bigcup_{U:Rel^0,\,m:Eq_1\to R} 1\Big)/\approx$$

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

$$\begin{array}{ccc} (\lambda u.Eq_1)U & \xrightarrow{(\lambda u.Eq_1)\,id_U} & (\lambda u.Eq_1)U \\ & {\scriptstyle m}\searrow \quad \swarrow{\scriptstyle m'} & \\ & R & \end{array} \qquad = \qquad \begin{array}{ccc} Eq_1 & \xrightarrow{id_{Eq_1}} & Eq_1 \\ & {\scriptstyle m}\searrow \quad \swarrow{\scriptstyle m'} & \\ & R & \end{array}$$

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, $(Lan_{\lambda u.Eq_1}\,\lambda u.1)\,R = \{m : Eq_1 \to R\}$. 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 $\pi_1\left((Lan_{\lambda u.Eq_1}\,\lambda u.Eq_1)\,R\right)$ 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 the interpretations of their fixpoint representations, such models cannot possibly be parametric.

We note that it is actually possible to construct a simpler counterexample to the Identity Extension Lemma for fixpoint representations 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. We therefore give the above example using the relation $(1,2,1\times 2)$ instead.

Taken together, Examples 4 and 5 show that Church encodings and fixpoint representations of GADTs 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 represented as their Church encodings or their fixpoint representations.

6 Conclusion and Related Work

We have shown that GADTs can be considered as data types in two contrasting ways: as their Church encodings, in which case they are completely determined by their syntax, or as fixpoints, in which case they are regarded as the functorial completion of their syntax. But regardless of which representation we choose, GADTs fail to have *all* expected parametricity properties. Indeed, we can read the results of Section 5 as showing that *i*) inhabitation and other non-naturality results for calculi representing GADTs as Church encodings cannot be extended to calculi representing them as fixpoints, and *ii*) naturality results for calculi representing GADTs as fixpoints cannot be extended to calculi representing them as Church encodings. So, neither representation guarantees *both* naturality consequences *and* non-naturality consequences of parametricity. Intuitively, this is because a GADT is not a data type in the usual container-ish sense, and completing it so that it becomes one destroys its relationally uniform — i.e., parametric — behavior. Which representation of data types a user chooses can therefore have significant consequences for programming with, and reasoning about, GADTs.

Although there are no parametric models for calculi representing GADTs as fixpoints, GADT analogues of most of Wadler's free theorems for ADTs still hold. Indeed, most of the parametricity results in Figure 1 of [37] are actually naturality properties. Naturality properties are often regarded as consequences of *parametricity*, but are, in fact, derivable directly from data types' *functorial semantics*. Even for data types whose Church encodings and fixpoint representations coincide, the analysis in this paper clearly distinguishes those free theorems that are actually consequences of naturality from those that are true consequences of parametricity.

There are treatments of GADTs beyond those discussed in the main body of this paper. Atkey's parametric model for F_ω from [2] represents data types — including GADTs — as Church encodings. It requires the user to supply a `map` function for the 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` in Atkey's model. Similarly, [36] presents 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 [35] to [36] are all inhabitation results rather than naturality results. In [23] 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 [16], as initial algebras of dependent polynomial functors in [8, 11], and as indexed containers in [25]. 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 in [3], but, as with the models mentioned above, this model also does not guarantee that GADTs have functorial semantics.

References

[1] A. Ahmed, D. Dreyer & A. Rossberg (2009): *State-dependent representation independence*. In: *Principles of Programming Languages*, pp. 340–353.

[2] R. Atkey (2012): *Relational parametricity for higher kinds*. In: *Computer Science Logic*, pp. 46–61.

[3] R. Atkey, N. Ghani & P. Johann (2014): *A Relationally Parametric Model of Dependent Type Theory*. In: *Principles of Programming Languages*, pp. 503–516.

[4] R. Bird & L. Meertens (1998): *Nested data types*. In: *Mathematics of Program Construction*, pp. 52–67.

[5] M. R. Bush, M. Leeming & R. F. C. Walters (2003): *Computing left Kan extensions*. Journal of Symbolic Computation 35, pp. 107–126.

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

[7] freeCodeCamp (2020): *The Iron Triangle, or "Pick Two"*. https://www.freecodecamp.org/news/the-iron-triangle-or-pick-two/.

[8] N. Gambino & M. Hyland (2003): *Well-founded trees and dependent polynomial functors*. In: TYPES, pp. 210–225.

[9] J.-Y. Girard (1972): *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. thesis, University of Paris.

[10] G. Gonzalez (2021): *The visitor pattern is essentially the same thing as Church encoding*. https://www.haskellforall.com/2021/01/the-visitor-pattern-is-essentially-same.html.

[11] M. Hamama & M. Fiore (2011): *A foundation for GADTs and inductive families: Dependent polynomial functor approach*. In: Workshop on Generic Programming, pp. 59–70.

[12] C.-K. Hur & D. Dreyer (2011): *A Kriple logical relation between ML and assembly*. In: Principles of Programming Languages, pp. 133–146.

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

[14] P. Johann (2003): *Short cut fusion is correct*. Journal of Functional Programming 13, pp. 797–814.

[15] P. Johann & N. Ghani (2007): *Initial algebra semantics is enough!* In: Typed Lambda Calculus and Applications, pp. 207–222.

[16] P. Johann & N. Ghani (2008): *Foundations for structured programming with GADTs*. In: Principles of Programming Languages, pp. 297–308.

[17] P. Johann, E. Ghiorzi & D. Jeffries (2021): *(Deep) induction for GADTs*. In: Submitted.

[18] P. Johann, E. Ghiorzi & D. Jeffries (2021): *Parametricity for primitive nested types*. In: Foundations of Software Science and Computation Structures, pp. 234–343.

[19] P. Johann & A. Polonsky (2019): *Higher-kinded data types: Syntax and semantics*. In: Logic in Computer Science, pp. 1–13.

[20] P. Johann & A. Polonsky (2020): *Deep Induction: Induction rules for (truly) nested types*. In: Foundations of Software Science and Computation Structures, pp. 339–358.

[21] S. L. Peyton Jones, D. Vytiniotis, G. Washburn & S. Weirich (2006): *Simple unification-based type inference for GADTs*. In: International Conference on Functional Programming, pp. 50–61.

[22] S. MacLane (1971): *Categories for the Working Mathematician*. Springer-Verlag.

[23] Y. Mandelbaum & A. Stump (2009): *GADTs for the OCaml masses*. In: Workshop on ML.

[24] Y. Minsky (2015): *Why GADTs matter for performance*. https://blog.janestreet.com/why-gadts-matter-for-performance/.

[25] P. Morris & T. Altenkirch (2009): *Indexed containers*. In: Logic in Computer Science, pp. 277–285.

[26] B. C. d. S. Oliveira, M. Wang & J. Gibbons (2008): *The visitor pattern as a reusable, generic, type-safe component*. In: Object-Oriented Programming: Systems, Languages, Applications, pp. 439–456.

[27] E. Pasalic & N. Linger (2004): *Meta-programming with typed object-language representations*. In: Generic Programming and Component Engineering, pp. 136–167.

[28] C. Penner (2020): *Simpler and safer API design using GADTs*. https://chrispenner.ca/posts/gadt-design.

[29] A. Pitts (1998): *Parametric polymorphism, recursive types, and operational equivalence*. Unpublished manuscript.

[30] A. Pitts (2000): *Parametric polymorphism and operational equivalence*. Mathematical Structures in Computer Science 10, pp. 1–39.

[31] F. Pottier & Y. Régis-Gianas (2006): *Stratified type inference for generalized algebraic data types*. In: Principles of Programming Languages, pp. 232–244.

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

[33] E. Riehl (2016): *Category Theory in Context*. Dover.

[34] TFCA (2019): *Transfinite Construction of Free Algebras*. http://ncatlab.org/nlab/show/transfinite+construction+of+free+algebras.

[35] D. Vytiniotis & S. Weirich (2006): *Parametricity and GADTs*. https://www.cis.upenn.edu/~sweirich//talks/param-gadt.pdf.

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

[37] P. Wadler (1989): *Theorems for free!* In: Functional Programming Languages and Computer Architecture, pp. 347–359.