# Foundations for Structured Programming with GADTs

Patricia Johann
Rutgers University, Camden, NJ, USA
pjohann@crab.rutgers.edu

Neil Ghani
University of Nottingham, Nottingham, UK
nxg@cs.nott.ac.uk

## Abstract

GADTs are at the cutting edge of functional programming and become more widely used every day. Nevertheless, the semantic foundations underlying GADTs are not well understood. In this paper we solve this problem by showing that the standard theory of data types as carriers of initial algebras of functors can be extended from algebraic and nested data types to GADTs. We then use this observation to derive an initial algebra semantics for GADTs, thus ensuring that all of the accumulated knowledge about initial algebras can be brought to bear on them. Next, we use our initial algebra semantics for GADTs to derive expressive and principled tools — analogous to the well-known and widely-used ones for algebraic and nested data types — for reasoning about, programming with, and improving the performance of programs involving, GADTs; we christen such a collection of tools for a GADT an *initial algebra package*. Along the way, we give a constructive demonstration that every GADT can be reduced to one which uses only the equality GADT and existential quantification. Although other such reductions exist in the literature, ours is entirely local, is independent of any particular syntactic presentation of GADTs, and can be implemented in the host language, rather than existing solely as a metatheoretical artifact. The main technical ideas underlying our approach are (i) to modify the notion of a higher-order functor so that GADTs can be seen as carriers of initial algebras of higher-order functors, and (ii) to use left Kan extensions to trade arbitrary GADTs for simpler-but-equivalent ones for which initial algebra semantics can be derived.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features-Data types and structures

***General Terms*** Languages, Theory

## 1. Introduction

*Generalized algebraic data types*, or GADTs, are at the cutting edge of functional programming and are finding an ever-increasing number of applications. Types are traditionally used to guarantee that programs do not ‘go wrong’ by using data in inappropriate ways. GADTs extend this traditional use of types, allowing them to index, and thereby capture more sophisticated properties of, data types. Properties of interest may include the size or shape of data, the state of program components, or some invariant that the data type is expected to satisfy. By encoding properties as types, GADTs provide a means of abstracting them into a form that compilers and other language tools can exploit. This makes properties of programs that would otherwise be available only dynamically, if at all, statically checkable and analyzable, and thus makes progress toward closing what Sheard calls the *semantic gap* between what the programmer knows about a program and what the programming language allows to be stated. GADTs are especially useful when combined with higher-kinded types, as in (extensions of) Haskell, and with user-definable kinds, as in Ωmega [Omega, Sheard et al. (2005)]. In such settings, GADTs can represent refinement types, witness types, existential types, and certain dependent types. GADTs are closely related to several other concepts, including first-class phantom types [Cheney & Hinze (2003)], equality-qualified types [Sheard & Pasalic (2004)], guarded recursive types [Xi et al. (2003)], and inductive families [Dybjer (1994)]. They can even be regarded as a variant of dependent types [McBride (2004), Xi & Pfenning (1999)] in which the distinction between types and values is maintained.

It is widely accepted that a good theory of data types requires principled foundations in order to provide expressive tools for structured programming with them. One of the most successful foundations for algebraic data types — such as the natural numbers, lists, trees — is that of *initial algebra semantics*. In initial algebra semantics, every data type is seen as the carrier of the initial algebra of a functor. The value of initial algebra semantics lies not only in its theoretical clarity, but also in that it supports principled structured programming techniques for data types including:

- *Combinators* which uniformly produce, uniformly consume, and otherwise capture commonly-occurring type-independent programming idioms for data types. Among the most significant of these are the `build` combinator, which uniformly produces structures of a given data type; the structured recursion combinator `fold`, which uniformly consumes structures of a given data type; and the `map` combinator, which applies a specified function to all data in a structure of a given data type.
- *Church encodings* for data types, which represent structures of data types as functions in the Girard-Reynolds polymorphic lambda calculus, the core formal calculus on which many functional languages are based. In addition to being interesting in their own right, Church encodings are the key to defining `build` combinators. Indeed, the `build` combinator for each data type derives from the isomorphism between that data type and its Church encoding.
- `fold`/`build` *fusion rules*, also derived from the isomorphisms between data types and their Church encodings, that can be used to improve the performance of modularly constructed programs which manipulate data of those types. Such a rule replaces a call to the `build` combinator for a given data type which is immediately followed by a call to the `fold` combinator for that type with an equivalent computation that does not

*POPL'08*, January 7–12, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-689-9/08/0001...$5.00

construct the intermediate structure introduced by `build` and immediately consumed by `fold`. Local program transformations [Gill et al. (1993), Johann (2002), Svenningsson (2002)] based on `fold`/`build` rules can result in order-of-magnitude increases in program efficiency. They also open the way for developing a so-called *algebra of fusion*.

- *Generic programming* techniques based on the observation that initial algebra semantics allows us to derive a *single* generic `fold` combinator, a *single* generic `build` combinator, and a *single* generic `fold`/`build` rule — each of which can be specialized to any particular data type of interest.

Despite their ubiquity and utility, the semantic foundations of programming with GADTs are not well understood. In particular, there is currently no known initial algebra semantics for GADTs. From a theoretical standpoint, it is unsettling to think that our understanding of data types in terms of initial algebras of functors is limited to algebraic and nested data types [Bird & Meertens (1998)], and not applicable to more advanced ones such as GADTs. But the lack of an initial algebra semantics for GADTs is also problematic from a programmer's point of view, since it forestalls the principled derivation of `folds`, `builds`, `fold`/`build` fusion rules, and other tools for structured programming with GADTs. Thus, for both theoretical and practical reasons, the time has come to ask whether or not an initial algebra semantics for GADTs can be given.

This paper provides an affirmative answer to this question for covariant GADTs. The covariance restriction is very mild: it is the natural analogue of the standard restrictions to covariant algebraic data types and covariant nested data types which appear in the literature, and is also satisfied by virtually all GADTs which arise in practice. Moreover, covariance is central to the derivation of initial algebra semantics for algebraic and nested data types, since initial algebras of functors are taken, and functors are, by their very nature, covariant. The covariance restriction is similarly central to the derivation of initial algebra semantics for GADTs.[1] We therefore speak simply of GADTs below, and leave the covariance restriction implicit.

With this understanding, we derive initial algebra packages for GADTs in any language which also supports universal and existential type quantification. We use categorical tools as our main technical devices, and use Haskell to illustrate and make these categorical techniques more accessible to the programming languages community. We offer two kinds of results:

- For those languages supporting formal parametric models, our results can be read as formal theorems about GADTs and their semantics. This is because the existence of a parametric model entails that existentially quantified types are interpreted as coends, that left Kan extensions are definable, and that initial algebras exist for all functors interpreting type constructors in the underlying language. The canonical candidate for a parametric model is the category of PERs [Bainbridge et al. (1990)].
- For languages not supporting formal parametric models, we use category theory as a heuristic to guide the derivation initial algebra packages for GADTs. In this case, no formal proofs are claimed for the initial algebra packages, and all results about initial algebra packages must be verified independently.

As mentioned above, irrespective of the target language, we use Haskell to illustrate our categorical ideas. We stress, however, that we do not claim the existence of a parametric model for full Haskell. The existence of such a model is a standard assumption in the literature, but this assumption becomes increasingly speculative as Haskell is extended by more and more features. Thus, although we use Haskell to illustrate our ideas, we presently regard full Haskell as being in the second group of languages discussed above.

In the remainder of this paper we assume that we have at our disposal a parametric model in the form of a category $\mathcal{C}$ whose objects interpret the types of the target language and whose morphisms interpret its functions. Our first major result is

THEOREM 1. *Given a parametric model, every GADT has an initial algebra semantics.*

Theorem 1 is significant both theoretically and practically. On the one hand, it entails that GADTs can be understood entirely in terms of initial algebras and thereby extends the semantics of algebraic types to cover GADTs. This is advantageous, since it ensures that all of our knowledge about initial algebras can be brought to bear on GADTs. On the other hand, Theorem 1 is the key to structured programming with GADTs since it can be used to derive from every GADT, directly from its declaration, a collection of tools for structured programming with data of that type. Indeed, in Section 5 we use the initial algebra semantics of GADTs to give `fold` and `build` combinators, Church encodings, and `fold`/`build` rules for GADTs, and to show that these constructs are worthy of the names, i.e., that they have properties analogous to those of the corresponding constructs for algebraic and nested data types. If we call such a collection of tools an *initial algebra package*, then a second contribution of this paper is to show that

THEOREM 2. *Every GADT has an initial algebra package.*

Note that this result holds even when no parametric model exists.

The difficulty in deriving initial algebra semantics for GADTs is two-fold. The first issue is that, although the interpretation of a GADT is most naturally a functor $\mathcal{C} \to \mathcal{C}$, such an interpretation turns out not actually to be possible. It is therefore not clear how to model GADTs as carriers of initial algebras of higher-order functors $(\mathcal{C} \to \mathcal{C}) \to \mathcal{C} \to \mathcal{C}$. Our solution to this problem lies in modeling GADTs as carriers of initial algebras of higher-order functors $(|\mathcal{C}| \to \mathcal{C}) \to |\mathcal{C}| \to \mathcal{C}$, where $|\mathcal{C}|$ is the discrete category derived from $\mathcal{C}$. The second issue is that an arbitrary GADT `G` has data constructors whose return types are not of the form `G a` for some type variable `a`, but rather are of the form `G (h a)`. (The possible presence of a non-identity `h` is what distinguishes GADTs from nested types.) We solve this second problem by using left Kan extensions to show how to trade a GADT whose data constructors have return types of the form `G (h a)` for an equivalent one all of whose constructors have return types of the form `G a`. An initial algebra semantics for any GADT can be derived from the initial algebra semantics of the equivalent GADT obtained by transforming all of its constructors in this way.

This paper is a follow-up paper to [Johann & Ghani (2007a), Johann & Ghani (2007b)], which derive initial algebra semantics for nested data types. Those papers view nested data types as carriers of initial algebras of higher-order Haskell functors mapping Haskell functors to Haskell functors. They also use Kan extensions to establish the expressiveness of the resulting `fold` and `build` combinators for nested data types. There are thus similarities with this paper, but note that i) GADTs are interpreted here as carriers of initial algebras of higher-order functors which map functors with discrete domains to functors with discrete domains, rather than functors with possibly nondiscrete domains to functors with possibly nondiscrete domains, and ii) Kan extensions are used in this paper to derive initial algebra semantics for GADTs rather than to establish their expressiveness. As a result, this paper is related only in spirit to [Johann & Ghani (2007a), Johann & Ghani (2007b)], via their shared use of functor categories and Kan extensions. But as

[1] We do, however, expect the standard techniques for dealing with mixed variance algebraic and nested data types to scale to mixed variance GADTs.

with those papers, the ideas in this one can be understood even if one has no prior knowledge of Kan extensions.

In the course of our development we also give a constructive derivation of a third fundamental result, namely

THEOREM 3. *Every GADT can be reduced to one involving only the equality GADT and existential quantification.*

Like Theorem 2, Theorem 3 does not require a parametric model. Yet in the presence of one it ensures that we can reason about GADTs by reasoning about the equality GADT and about existential quantification. Although this result is implicit in the literature (see, e.g., [Sulzmann & Wang (2005), Sulzmann & Wang (2004)]), our reduction has the benefit of being local to the particular GADT under consideration, and so does not require manipulation of the entire host language. It therefore is not impacted by the addition or deletion of other language features. Furthermore, our reduction is independent of any particular syntactic presentation of GADTs. And it is easily implemented in Haskell, rather than existing solely as a metatheoretical artifact.

We make several other important contributions as well. First, we execute our program for deriving initial algebra semantics and packages for GADTs in a generic style by parameterizing our GADTs over the types occurring in them. While this incurs a certain notational overhead, the inconvenience is outweighed by having *a single* generic `fold` combinator, *a single* generic `build` operator, and *a single* generic `fold`/`build` rule, each of which can be specialized to any particular GADT of interest. Secondly, while the theory of GADTs has previously been developed only for syntactically defined classes of GADTs, our development is not restricted to a particular syntactic definitional format for GADTs, but rather is based on the semantic notion of the carrier of the initial algebra of a (in this case, higher-order) functor. Thirdly, this paper provides a compelling demonstration of the practical applicability of Kan extensions, and thus has the potential to render them more accessible to functional programmers. Finally, we give a complete implementation of our ideas in Haskell, available at `http://www.cs.nott.ac.uk/~nxg`. This demonstrates their practical applicability, makes them more accessible, and provides a partial guarantee of their correctness via the Haskell type-checker. This paper can therefore be read *both* as abstract mathematics *and* as providing the basis for experiments and practical applications.

The remainder of this paper is organized as follows. In Section 2 we introduce GADTs, use Haskell (for which our abstract category $\mathcal{C}$ interprets the kind `*`) as a target language to show that GADTs are not obviously functors $\mathcal{C} \to \mathcal{C}$, and discuss the difficulty this poses for deriving initial algebra semantics for them. In Section 3 we show that GADTs from a restricted class are semantically carriers of initial algebras of higher-order functors $(|\mathcal{C}| \to \mathcal{C}) \to |\mathcal{C}| \to \mathcal{C}$ (although, as just noted, *not* necessarily higher-order functors $(\mathcal{C} \to \mathcal{C}) \to \mathcal{C} \to \mathcal{C}$). We then use this observation to derive initial algebra semantics for such GADTs. In Section 4 we recall the derivation of initial algebra packages of structured programming tools for inductive types and recap the well-known categorical ideas which underlie them. In Section 5 we instantiate these ideas in the relevant category to derive initial algebra packages for GADTs from our restricted class. In Section 6 we extend our derivation to all (covariant) GADTs. We derive initial algebra packages for some GADTs from the literature in Section 7. In Section 8 we conclude and indicate our next step in this line of research.

## 2. The Problem

### 2.1 GADTs

Syntactically, GADTs are generalizations of algebraic data types. An *algebraic data type* is a data type with the property that all applications of its type constructor which occur in its declaration have precisely the same type variables as arguments.[2] Consider, for example, the data type of lists, realizable in Haskell as

```
data List a = Nil | Cons a (List a)
```

This declaration introduces the data constructors `Nil :: List a` and `Cons :: a -> List a -> List a`. The type variables appearing in the declaration of an algebraic data type are implicitly universally quantified; algebraic data types and their data constructors are thus polymorphic in general. Since every application of the type constructor `List` in the declaration of `List a` — and thus in the types of `Nil` and `Cons` — is of the form `List a` for the same type variable `a`, the `List` data type is indeed algebraic.

GADTs relax the restriction on the types of data constructors of algebraic data types. Data constructors for a GADT can both take as arguments, and return as results, data whose types involve type instances of the GADT other than the one being defined. (By contrast, data constructors of nested types can take as arguments, but *cannot* return as results, data whose types involve instances of the nested type other than the one being defined.) For example, the following Haskell declaration defines the GADT `Term`:

```
data Term a where
  Const :: a -> Term a
  Pair  :: Term a -> Term b -> Term (a,b)
  App   :: Term (a -> b) -> Term a -> Term b
```

As is customary, the types of the data constructors of this GADT are given explicitly. The data constructor `Pair` takes as input data of the instances `Term a` and `Term b`, and returns as its result data of the instance `Term (a,b)` of the `Term` GADT. Similarly, `App` takes as input data of the instances `Term (a -> b)` and `Term a` of the GADT, and return data of the instance `Term b`. Although it is not illustrated by this example, a single data constructor for a GADT can simultaneously take as input *and* return as its result data of new instances of that GADT as well. As for algebraic and nested data types, all of the type variables appearing in the types of the data constructors of a GADT are implicitly universally quantified, so that the type parameter `a` in the expression `Term a` is a “dummy” parameter used only to give the kind of the GADT type constructor `Term`. Using Haskell's kind syntax, we could have instead written the above declaration as `data Term :: * -> * where ....`

### 2.2 Are GADTs Haskell Functors?

The goal of this paper is to show that GADTs have initial algebra semantics. To have an initial algebra semantics, a GADT `G` must have an interpretation as the initial algebra of a functor (see Section 4.2). Since `G` is a type constructor, it is natural to try to model it as a functor $\mathcal{C} \to \mathcal{C}$ which is the carrier of the initial algebra of a functor $(\mathcal{C} \to \mathcal{C}) \to \mathcal{C} \to \mathcal{C}$. The first question which arises is thus whether or not every GADT can be interpreted as a functor $\mathcal{C} \to \mathcal{C}$.

To answer this question, consider the situation in Haskell, where the category $\mathcal{C}$ interprets the Haskell kind `*`. Functors are implemented in Haskell as type constructors supporting `fmap` functions, as captured by the built-in type class

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

The function `fmap` is expected to satisfy the functor laws

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

stating that `fmap` preserves identities and composition. But satisfaction is not enforced by the compiler: it is the programmer's respon-

[2] There are different uses of the word *algebraic* in the literature. We use this definition consistently in this paper.

sibility to ensure that the `fmap` function for each Haskell functor, i.e., instance of Haskell's `Functor` class, behaves appropriately.

If `G` is to be a Haskell functor then it must support an `fmap` operation. Establishing that `Term` is a Haskell functor, for example, would require a declaration

```
instance Functor Term where
  fmap :: (d -> e) -> Term d -> Term e
```

The clause for terms of the form `Pair x y` would have `f :: (b,c) -> e` and `fmap f (Pair x y) :: Term e`. Keeping in mind that `fmap` should change the data in a term while preserving its structure, we should have

```
fmap f (Pair x y) = Pair u v
```

for some appropriately typed `u` and `v`. But it is not clear how to achieve this since `e` cannot be guaranteed to have a tuple structure. Even if `e` *were* guaranteed to be of the form `(b', c')`, we'd then have `f :: (b,c) -> (b',c')`, but still no way to produce data of type `Term (b',c')` from only `f :: (b,c) -> (b',c')` and `x :: b` and `y :: c`. The problem stems from the fact that `f` is not guaranteed to be a pair of functions. So it is not clear how to treat an arbitrary GADT in Haskell as a Haskell functor. It is thus not clear how to treat GADTs semantically as functors in general.

## 3. The Key Ideas

### 3.1 Recovering Functoriality

The above considerations lead us to try to model a GADT `G` as a functor $\mathcal{C}' \rightarrow \mathcal{C}$ for a category $\mathcal{C}'$ distinct from the category $\mathcal{C}$. A natural candidate for $\mathcal{C}'$ is the category whose objects model the (inhabited) types of `G`. For the `Fin` GADT from Example 1 in Section 7, for instance, the inhabited types are essentially the natural numbers. But for some GADTs, such as `Term`, the inhabited types will be all types. So a simpler and more uniform approach is always to take the objects of $\mathcal{C}'$ to be those of $\mathcal{C}$.

At first glance this seems to have gotten us nowhere, but we have not yet fully specified the category $\mathcal{C}'$. In addition to specifying the objects of $\mathcal{C}'$, we must also specify its morphisms. Since the only functions between types that we know for sure can be lifted to functions between GADTs parameterized over those types are the identity functions, we simply take $\mathcal{C}'$ to have as morphisms only the identities. That is, we take the category $\mathcal{C}'$ to be the discrete category derived from $\mathcal{C}$. Writing $|\mathcal{C}|$ for this category, we observe that every function from the objects of $|\mathcal{C}|$ to $\mathcal{C}$ is trivially a *functor* $|\mathcal{C}| \rightarrow \mathcal{C}$. We now turn our attention to showing that every GADT can be modeled as the carrier of the initial algebra of a functor $(|\mathcal{C}| \rightarrow \mathcal{C}) \rightarrow |\mathcal{C}| \rightarrow \mathcal{C}$.

### 3.2 Basic GADTs and Higher-order Functors

To keep syntactic overhead to a minimum, and to highlight the key ideas underlying of our approach, we make some simplifying assumptions in this section about the syntax of GADTs. We show how to extend our approach to GADTs which do not satisfy these assumptions in Section 6. On the other hand, to emphasize that our approach is generic over all GADTs, as discussed in the introduction, our notation explicitly parameterizes GADTs over the types from which they are constructed.

We say a GADT `G f h` is *basic* if it has the form

```
data G f h a where
  GCon :: f (G f h) a -> G f h (h a)
```

The covariance restriction mentioned in the introduction is captured by the requirement that `G` does not appear in `h`, and that `f` is an instance of the higher-order functor class which we now discuss. A *higher-order functor* maps functors to functors, and maps between functors to maps between functors, i.e., natural transformations to natural transformations. In particular, higher-order functors on `*` can be implemented in Haskell by the following analogue of the built-in `Functor` class:

```
class HFunctor f where
  ffmap :: Functor g => (a -> b) -> f g a -> f g b
  hfmap :: Nat g h -> Nat (f g) (f h)
```

That a higher-order functor maps functors to functors is captured by the requirement that an `HFunctor` — i.e., an instance of the `HFunctor` class — supports an `ffmap` function; that a higher-order functor maps natural transformations to natural transformations is captured by the requirement that an `HFunctor` supports an `hfmap` function. The type of natural transformations can be given in Haskell by

```
type Nat g h = forall a. g a -> h a
```

A parametric interpretation of the `forall` quantifier ensures that a function of type `Nat g h` can be thought of as a uniform family of maps from `g` to `h`, so that the relevant naturality squares commute. While not explicit in the class definition above, the programmer is expected to verify that if `g` is a Haskell functor, then `f g` is also a Haskell functor. Moreover, like the `fmap` functions for functors, the `hfmap` functions are expected to preserve identities and composition — here for natural transformations.

An example of a basic GADT is the following alternative presentation of the GADT `Fin` of finite sets from Example 1 in Section 7. Here, `Either` is the standard Haskell type for disjunctions, `f g a` is `Either Unit (g a)`, `h a` is `S a`, and `G f h` is `BFin`.

```
data BFin a where
  BFinCon :: Either Unit (BFin a) -> BFin (S a)
```

Now, if `G f h` is to be interpreted as the carrier of the initial algebra of a higher-order functor $(|\mathcal{C}| \rightarrow \mathcal{C}) \rightarrow |\mathcal{C}| \rightarrow \mathcal{C}$, then in order for `G f h` and its data constructor `GCon` to be well-kinded we should have in Haskell:

```
h :: |*| -> |*|
f :: (|*| -> *) -> |*| -> *
G f h :: |*| -> *
```

We can tailor the `HFunctor` class above to accommodate higher-order functors of kind `(|*| -> *) -> |*| -> *`. This amounts to eliminating the `ffmap` function from the class definition, since every functor necessarily maps identities to identities. When working with GADTs, we therefore use the following specialized version of the `HFunctor` class:

```
class HFunctor f where
   hfmap :: Nat g h -> Nat (f g) (f h)
```

### 3.3 Initial Algebra Semantics for Basic GADTs

We seek to derive from the syntax of `G f h` a higher-order functor $(|\mathcal{C}| \rightarrow \mathcal{C}) \rightarrow |\mathcal{C}| \rightarrow \mathcal{C}$ such that the carrier of its initial algebra interprets `G f h`.

In a parametric model, the interpretation of the type `f (G f h) a -> G f h (h a)` of `GCon` is isomorphic to the interpretation of the type `Lan h (f (G f h)) a -> G f h a`, where

```
data Lan h g c = forall b. Lan (Eql (h b) c, g b)
```

is the Haskell representation of the *left Kan extension* [MacLane (1971)] of `g` along `h`, and `Eql` is the *equality GADT*

```
data Eql a b where
  Refl :: Eql a a
```

The use of `Eql` in the definition of `Lan` reflects the fact that, in Haskell, the domain of `h` is the discrete category `|*|`. Constructing an element of type `G f h c` requires using `GCon`, and thus finding a type `b` such that `h b = c` and giving an element of `f (G f h) b` to which `GCon` can be applied. The interpretation of `G f h c` is therefore isomorphic to the interpretation of `exists b. (h b = c, f (G f h) b)`. Writing this type as `Lan h (f (G f h)) c` captures this observation precisely — once we remember that existential type quantification is written using top-level universal type quantification in Haskell.

From this it follows that the interpretation of `G f h` is the carrier of the initial algebra of the higher-order functor interpreting `K f h`, where `K f h` is defined by

```
K f h g a = Lan h (f g) a
```

Unfortunately, we cannot simply use this specification in a Haskell type synonym, since this would render the partial application `K f h` unavailable for computations. We therefore make the following data type declaration instead:

```
data K f h g a = forall b. HFunctor f =>
                       K (Eql (h b) a, f g b)
```

We can verify that `K f h` is indeed an `HFunctor` by observing that it is the composition of `f`, which is an `HFunctor` by assumption, and `Lan h`, which is an `HFunctor` as shown here:

```
instance HFunctor (Lan h) where
  hfmap k (Lan (p, v)) = Lan (p, k v)
```

Composition of `HFunctors` can be coded in Haskell using

```
newtype (HFunctor g, HFunctor h) =>
              HComp g h k a = HComp (g (h k) a)
```

The following instance declaration shows that the composition of two `HFunctors` is again an `HFunctor`:

```
instance (HFunctor g, HFunctor h) =>
  HFunctor (g `HComp` h) where
    hfmap k (HComp t) = HComp (hfmap (hfmap k) t)
```

We can instantiate this declaration for `Lan h` and `f` to show that their composition is an `HFunctor`. But unfortunately, this causes a proliferation of `HComp` type and data constructors throughout our code. To avoid this — and thus purely for cosmetic reasons — we give an `HFunctor` instance declaration for the composition of `Lan h` and `f` directly. We have

```
instance HFunctor f => HFunctor (K f h) where
  hfmap k (K (p, v)) = K (p, hfmap k v)
```

The `hfmap` on the right-hand side of the definition in the instance declaration is the one for `f`.

Since `G f h` is inductively defined, we can see that the interpretation of `G f h` is the carrier of the initial algebra of the interpretation of `K f h` by establishing that the interpretation of `G f h` is isomorphic to the interpretation of the type

```
data NG f h a where
  NGCon :: K f h (NG f h) a -> NG f h a
```

This follows from the isomorphism between the interpretation of `forall c. Lan h g c -> f c` and the interpretation of `forall c. g c -> f (h c)` which holds for all `h`, `g`, and `f`. This isomorphism can be coded in Haskell as

```
toLan :: (forall c. g c -> f (h c)) ->
                              Lan h g c -> f c
toLan s (Lan (Refl, v)) =  s v

fromLan :: (forall c. Lan h g c -> f c) ->
                               g c -> f (h c)
fromLan s t = s (Lan (Refl, t))
```

For our left Kan extension `K`, `toLan` and `fromLan` specialize to the following functions giving an isomorphism between the interpretation of `forall c. K f h g c -> k c` and the interpretation of `forall c. f g c -> k (h c)`:

```
toK :: HFunctor f => (forall c. f g c -> k (h c))
                        -> K f h g c -> k c
toK s (K (Refl, v)) =  s v

fromK :: HFunctor f => (forall c. K f h g c -> k c)
                          -> f g c -> k (h c)
fromK s t = s (K (Refl, t))
```

Thus, giving data appropriate for input to `GCon` is equivalent to giving data appropriate for input to `NGCon`. Since the data constructors of these types contain exactly the same information, they generate exactly the same GADT.

In light of the above, we can write `G f h` as `Mu (K f h)` where

```
newtype Mu f a = In (f (Mu f) a)
```

represents the carrier of the initial algebra of the interpretation of `f`. Since `G f h` is interpreted as the carrier of the initial algebra of the interpretation of `K f h`, it admits an initial algebra semantics.

## 4. Semantic Foundations of Initial Algebra Packages

In this section we first recall the derivation of initial algebra packages for inductive types from [Ghani et al. (2003)], and then recap the well-known categorical ideas which underlie it. In the next section we show how to instantiate these same ideas in our higher-order setting to derive initial algebra packages for GADTs.

### 4.1 Initial Algebra Packages for Inductive Types

An *inductive data type* is a data type which can be interpreted as the carrier of the initial algebra of a functor, and an *inductive data structure* is a data structure of inductive type. It is well known [Ghani et al. (2005), Ghani et al. (2003), Takano & Meijer (1995)] that every inductive type has an associated initial algebra package. If `f` is a Haskell functor, then the associated inductive type `M f` and its associated `fold` and `build` combinators can be implemented generically in Haskell by[3]

```
newtype M f = Inn (f (M f))

fold :: Functor f => (f a -> a) -> M f -> a
fold h (Inn k) = h (fmap (fold h) k)

build :: Functor f =>
            (forall b. (f b -> b) -> b) -> M f
build g = g Inn
```

As usual, the type of the polymorphic function argument to `build` gives the Church encoding for `M f`. These `build` and `fold` combinators can be used to produce and consume inductive data structures of type `M f`. Moreover, if `f` is any Haskell functor, `h` is any function of any type `f a -> a`, and `g` is any function of closed type `forall b. (f b -> b) -> b`, then the following `fold`/`build` rule for `M f` eliminates from computations structures of type `M f` which are produced by `build` and immediately consumed by `fold`:

```
fold h (build g) = g h
```

The familiar initial algebra package for list types [Gill et al. (1993)] is an instance of this scheme, as are the corresponding packages

[3] Unfortunately, we cannot use the same fixed point operator here as in Section 3. This is because `f` is not an `HFunctor` in this case, which is problematic because Haskell lacks proper polymorphic kinding.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr c n xs = case xs of []   -> n
                          z:zs -> c z (foldr c n zs)

-- Church encoding
forall b. (a -> b -> b) -> b -> b

buildL :: (forall b. (a -> b -> b) -> b -> b) -> [a]
buildL g = g (:) []

-- fold/build rule
foldr c n (buildL g) = g c n
```

**Figure 1.** Generic initial algebra package for list types.

```
foldT :: (a -> b) -> (b -> b -> b) -> Tree a -> b
foldT l b t = case t of
                Leaf x -> l x
                Branch t1 t2 -> b (foldT l b t1)
                                  (foldT l b t2)

-- Church encoding
forall b. (a -> b) -> (b -> b -> b) -> b

buildT :: (forall b. (a -> b) ->
             (b -> b -> b) -> b) -> Tree a
buildT g = g Leaf Branch

-- fold/build rule
foldT l b (buildT g) = g l b
```

**Figure 2.** Generic initial algebra package for `Tree`.

for algebraic data types given in [Johann (2002)]. The former is given in Figure 1, while the initial algebra package for the non-list algebraic data type

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

of trees over data of type `a` is given in Figure 2.

It is the fact that inductive types are interpreted by carriers of initial algebras of functors that makes it possible to define initial algebra packages for them. In particular, this is what ensures that their `fold`s are true folds, that their Church encodings really encode them, and that their `fold`/`build` rules truly are correct. It's what allows us to program with, and reason about programs involving, inductive types. Section 4.2 explains the theory underlying this assertion and, at the same time, develops the semantic foundations for our derivation of initial algebra packages for GADTs; note that the material in Section 4.2.2 may be new even to those familiar with initial algebra semantics. Readers without the required background in category theory or whose main focus is not on the categorical foundations of initial algebra packages for advanced data types can safely omit this section and other categorical discussions in the paper since all of the relevant category-theoretic constructs used in this paper are implemented in Haskell. Readers who choose to do this will miss some of the motivations for the theory of GADTs, and some of the connections between the theory of inductive types and the theory of GADTs developed in this paper, but will miss no necessary facts. We stress that we do not attempt a complete reconstruction of all of category theory here, but instead introduce only those concepts that form the basis of our initial algebra approach to deriving principled programming tools for GADTs.

## 4.2 The Fundamental Theory

The key idea underlying initial algebra packages is the idea of *initial algebra semantics*. Within the paradigm of initial algebra semantics, every data type is interpreted as the carrier $\mu F$ of the initial algebra of a suitable functor $F : \mathcal{C} \to \mathcal{C}$ for some suitable category $\mathcal{C}$. In more detail, suppose we have fixed a category $\mathcal{C}$. An *algebra* for a functor $F : \mathcal{C} \to \mathcal{C}$ (or, simply, an $F$*-algebra*) is a pair $(A, h)$ where $A$ is an object of $\mathcal{C}$ and $h : FA \to A$ is a morphism of $\mathcal{C}$. Here, $A$ is called the *carrier* of the algebra and $h$ is called its *structure map*. As it turns out, the $F$-algebras for a given functor $F$ themselves form a category. In the category of $F$-algebras, a morphism between $F$-algebras $(A, h)$ and $(B, g)$ is a map $f : A \to B$ such that the following diagram commutes:

$$\begin{tikzcd} FA \arrow[r, "Ff"] \arrow[d, "h"'] & FB \arrow[d, "g"] \\ A \arrow[r, "f"] & B \end{tikzcd}$$

We call such a morphism an $F$*-algebra homomorphism*.

Now, if the category of $F$-algebras has an initial object — called an *initial algebra for* $F$, or, more simply, an *initial* $F$*-algebra* — then Lambek's Lemma ensures that the structure map of this initial $F$-algebra is an isomorphism, and thus that its carrier is a fixed point of $F$. If it exists, the initial $F$-algebra is unique up to isomorphism. We write $(\mu F, in)$ for the initial $F$-algebra comprising the fixed point $\mu F$ and the isomorphism $in : F(\mu F) \to \mu F$.

### 4.2.1 Folds

The standard interpretation of a type constructor is a functor $F$, and the standard interpretation of the inductive type it defines is the carrier of the initial algebra of $F$. Initiality ensures that there is a unique $F$-algebra homomorphism from the initial $F$-algebra to any other $F$-algebra. The map underlying this $F$-algebra homomorphism is exactly the *fold* for $\mu F$. Thus if $(A, h)$ is any $F$-algebra, then *fold* $h : \mu F \to A$ makes the following diagram commute:

$$\begin{tikzcd} F(\mu F) \arrow[r, "F(\mathit{fold}\ h)"] \arrow[d, "in"'] & FA \arrow[d, "h"] \\ \mu F \arrow[r, "\mathit{fold}\ h"] & A \end{tikzcd}$$

From this diagram, we see that $\mathit{fold} : (FA \to A) \to \mu F \to A$ and that *fold* $h$ satisfies $\mathit{fold}\ \ h\ (in\ \ t) = h\ (F\ (\mathit{fold}\ \ h)\ t)$. This justifies the definition of the `fold` combinator given in Section 4.1. Furthermore, the uniqueness of the mediating morphism ensures that, for every algebra $h$, the map *fold* $h$ is defined uniquely. This provides the basis for the correctness of `fold` fusion for inductive types, which derives from the fact that if $h$ and $h'$ are $F$-algebras and $\psi$ is an $F$-algebra homomorphism from $h$ to $h'$, then $\psi \,.\, \mathit{fold}\ \ h = \mathit{fold}\ \ h'$. But note that `fold` fusion [Bayley (2001), Blampied (2000), Bird & Paterson (1998), Bird & Paterson (1999), Martin et al. (2004)] is distinct from, and inherently simpler than, the `fold`/`build` fusion in this paper.

### 4.2.2 Church Encodings, builds, and fold/build Fusion Rules

Although the above discussion shows that `fold` combinators for inductive types can be derived entirely from, and understood entirely in terms of, initial algebra semantics, regrettably the standard initial algebra semantics does not provide a similar principled derivation of the `build` combinators or the correctness of the `fold`/`build` rules. In fact, `build` has been regarded as a kind of optional "add-on" which is not a fundamental part of the basic infrastructure for programming with inductive types. The practical consequence of this is that the `build` combinators have been largely overlooked, treated as poor relatives of their corresponding `fold` combinators, and regarded as unworthy of fundamental study.

This situation was rectified in [Ghani et al. (2003)], where the standard initial algebra semantics was extended to support not only `fold` combinators for inductive types, but also Church encodings and `build` combinators for them. Indeed, [Ghani et al. (2003)] considers the initial $F$-algebra to be not only the initial object of the category of $F$-algebras, but also the limit of the forgetful functor from the category of $F$-algebras to the underlying category $\mathcal{C}$ as well. We now summarize this result and its consequences, which we later apply to derive our combinators for GADTs.

If $F$ is a functor on $\mathcal{C}$, then the *forgetful functor* $U_F$ maps $F$-algebras to objects in $\mathcal{C}$ by forgetting the $F$-algebra structure. That is, $U_F$ maps an $F$-algebra $(A, h)$ to its carrier $A$, and maps an $F$-algebra homomorphism $f : A \to B$ between $F$-algebras $(A, h)$ and $(B, g)$ to the morphism $f : A \to B$ in $\mathcal{C}$. If $C$ is an object in $\mathcal{C}$, then a *$U_F$-cone for $C$* comprises, for every $F$-algebra $(A, h)$, a morphism $\nu_{(A,h)} : C \to A$ in $\mathcal{C}$ such that, for every $F$-algebra map $f : A \to B$, we have $\nu_{(B,g)} = f \circ \nu_{(A,h)}$.

$$\begin{tikzcd}
(A,h) \arrow[rr, "f"] & & (B,g) \\
& C \arrow[ul, "\nu_{(A,h)}"] \arrow[ur, "\nu_{(B,g)}"'] &
\end{tikzcd}$$

We write $(C, \nu)$ for this cone, and call $C$ its *vertex* and the morphism $\nu_{(A,h)}$ the *projection* from $C$ to $A$. A $U_F$-cone with vertex $C$ can be thought of as having the type $\forall x.(Fx \to x) \to C \to x$. A *$U_F$-cone morphism* $g : (C, \nu) \to (D, \mu)$ between $U_F$-cones $(C, \nu)$ and $(D, \mu)$ is a morphism $g : C \to D$ in $\mathcal{C}$ such that for any $F$-algebra $(A, h)$, we have $\mu_{(A,h)} \circ g = \nu_{(A,h)}$. A *$U_F$-limit* is a $U_F$-cone to which there is a unique $U_F$-cone morphism, called the *mediating morphism*, from any other $U_F$-cone. When they exist, $U_F$-limits are unique up to isomorphism. Moreover, no extra structure is required of either $F$ or $\mathcal{C}$ for the $U_F$-limit to exist — it simply comprises the carrier of the initial $F$-algebra together with the family of $F$-algebra-indexed *fold* functions.

In [Ghani et al. (2003)], the characterization of the initial $F$-algebra as both the above limit *and* the initial object in the category of $F$-algebras is called the *extended initial algebra semantics*. As shown there, an initial $F$-algebra has a different universal property as a limit from the one it inherits as an initial object. This alternate universal property ensures:

- For each $F$-algebra, the projection from the vertex of the $U_F$-limit (i.e., from $\mu F$) to the carrier of that $F$-algebra defines $\mathit{fold} : (Fx \to x) \to \mu F \to x$.
- The unique mediating morphism from the vertex $C$ of a $U_F$-cone to the vertex $\mu F$ of the $U_F$-limit defines $\mathit{build} : (\forall x. (Fx \to x) \to C \to x) \to C \to \mu F$ to be $\mathit{build}\ g = g\ \mathit{in}$. This justifies the definition of the `build` combinator given in Section 4.1.
- Correctness of the `fold`/`build` rule follows from the fact that $\mathit{fold}\ h\ .\ \mathit{build}\ g = g\, h$, i.e., that *fold* after *build* is a projection after a mediating morphism from $C$ to $\mu F$, and is thus equal to the projection from $C$ to the carrier of $h$. Taking $C$ to be the unit type proves the correctness, relative to the underlying semantics, of the `fold`/`build` rule given in Section 4.1.

The extended initial algebra semantics thus shows that, given a parametric interpretation of universal quantification of types, there is an isomorphism between the interpretation of the type `c -> M f` and the interpretation of its *generalized Church encoding* `forall x. (f x -> x) -> c -> x`. The term "generalized" reflects the presence of the parameter `c`, which is absent in other Church encodings [Takano & Meijer (1995)], but is essential to the derivation of `build` combinators for GADTs. Choosing `c` to be the unit type gives the usual isomorphism between the interpretation of an inductive type and the interpretation of its usual Church encoding. This isomorphism comprises precisely *fold* (up to order of arguments) and *build* for the interpretation of that type. Writing $\mathit{fold}'\ m\ h$ for $\mathit{fold}\ h\ m$ we have

$$\begin{array}{lcl} \mathit{fold}' & :: & \mu F \to \forall x.(Fx \to x) \to x \\ \mathit{build} & :: & (\forall x.(Fx \to x) \to x) \to \mu F \end{array}$$

From this we see that correctness of the `fold`/`build` rule for inductive types codes one half of the requirement that *build* and $\mathit{fold}'$ are mutually inverse. A parametric model guarantees the existence of the interpretation of the Church encoding of a type constructor, which, by this isomorphism, guarantees the existence of the initial algebra for the interpretation of that type constructor. Generic `build` combinators and Church encodings for inductive types are given in [Takano & Meijer (1995)], but that paper does not show how to derive *build*s for fixed points of higher-order functors or how to interpret GADTs as such fixed points. Indeed, it doesn't even mention GADTs.

## 5. Initial Algebra Packages for Basic GADTs

In this section we give Haskell implementations of the initial algebra packages for basic GADTs derived from the initial algebra semantics developed for them in Section 3. From Section 3 we have that the interpretation of the basic GADT

```
data G f h a where
  GCon :: f (G f h) a -> G f h (h a)
```

is isomorphic to the interpretation of

```
data NG f h a where
  NGCon :: K f h (NG f h) a -> NG f h a
```

which is the carrier of the initial algebra of the interpretation of `K f h`, where

```
data K f h g a = forall b. HFunctor f =>
                      K (Eql (h b) a, f g b)
```

Instantiating the fundamental theory we get the `fold` combinator

```
foldNG :: (HFunctor f, Functor h) =>
               Nat (K f h a) a -> Nat (NG f h) a
foldNG m (NGCon u) = m (hfmap (foldNG m) u)
```

the generalized Church encoding

```
(forall a. Nat (K f h a) a -> Nat c a)
```

and the `build` combinator

```
buildNG :: HFunctor f => (forall a. Nat (K f h a) a
             -> Nat c a) -> Nat c (NG f h)
buildNG g = g NGCon
```

for `NG f h`. Here, the instance of `hfmap` in the definition of `foldNG` is the one for the `HFunctor` `K f h`. We have the `fold`/`build` rule

```
foldNG m . buildNG g = g m
```

for `NG f h`. If we define

```
toNG :: HFunctor f => G f h a -> NG f h a
toNG (GCon t) = NGCon (K (Refl, hfmap toNG t))

fromNG :: (HFunctor f, Functor h) =>
                   NG f h a -> G f h a
fromNG = foldNG (toK GCon)
```

```
foldG :: HFunctor f => (forall a. f x a -> x (h a))
                          -> Nat (G f h) x
foldG m (GCon t) = m (hfmap (foldG m) t)

-- generalized Church encoding
forall y. (forall a. f y a -> y (h a)) -> Nat c y

buildG :: HFunctor f =>
             (forall y. (forall a. f y a -> y (h a))
                -> Nat c y) -> Nat c (G f h)
buildG g = g GCon

-- fold/build rule
foldG m . buildG g = g m
```

**Figure 3.** Generic initial algebra package for GADTs.

then we have that `toNG` and `fromNG` are Haskell codings of mutual inverses. We can therefore use these functions to derive from the initial algebra package for `NG f h` the one for `G f h` given in Figure 3. The derivation is based on the following definitions:

```
foldG :: HFunctor f => (forall a. f x a -> x (h a))
                            -> Nat (G f h) x
foldG m t = foldNG (toK m) (toNG t)

buildG :: HFunctor f =>
            (forall y. (forall a. f y a -> y (h a))
               -> Nat c y) -> Nat c (G f h)
buildG g = fromNG . (buildNG g')
            where g' k = g (fromK k)
```

The instance of `hfmap` in the definition of `foldG` is the one for `f`. Note that `foldG` terminates since it is structurally recursive. Unwinding these definitions justifies the definitions in Figure 3.

## 6. More General GADTs

We have seen how a basic GADT, i.e., a GADT of the form

```
data G f h a where
     GCon :: f (G f h) a -> G f h (h a)
```

can be reduced to a data type `NG f h` using only existentials and the `Eql` GADT. We considered this special case first to highlight the basic ideas underlying our approach to initial algebra semantics for GADTs, as well as to avoid the cumbersome notation associated with arbitrary GADTs. In this section we show how the basic syntactic restriction can be lifted, and thus how our approach can be extended to arbitrary GADTs. Below, the covariance restriction entails that `G` does not appear in any `hi`, and that any `fi` is a higher-order functor. The initial algebra package for the `Term` GADT from Section 2.1 appears in Section 7 below. There are four independent dimensions along which basic GADTs can be generalized.

### 6.1 GADTs with Non-Unary Data Constructors

If we have a GADT of the form

```
data G f1 f2 h a where
  GCon :: f1 (G f1 f2 h) a -> f2 (G f1 f2 h) a ->
              G f1 f2 h (h a)
```

then we can curry `GCon` to derive a data constructor which takes one tupled argument. The above GADT is thus equivalent to:

```
data G' f1 f2 h a where
  GCon' :: PrHFunctor f1 f2 (G' f1 f2 h) a ->
             G' f1 f2 h (h a)
```

where `PrHFunctor f1 f2` is the higher-order functor given by

```
newtype PrHFunctor f1 f2 g a =
    PrHFunctor (f1 g a, f2 g a)
```

It is easy to check that `PrHFunctor f1 f2` is an `HFunctor`. Thus we can reduce a GADT whose data constructor takes several inputs to one whose data constructor takes only a single input. An example of a GADT with a non-unary data constructor is `Term`.

### 6.2 GADTs with More Than One Data Constructor

The presence of the different functors `h1` and `h2` in the codomain types of the data constructors `GCon1` and `GCon2` of a GADT `G f1 f2 h1 h2` of the form

```
data G f1 f2 h1 h2 a where
  GCon1 :: f1 (G f1 f2 h1 h2) a ->
                     G f1 f2 h1 h2 (h1 a)
  GCon2 :: f2 (G f1 f2 h1 h2) a ->
                     G f1 f2 h1 h2 (h2 a)
```

entails that it is not possible to reduce a GADT with two data constructors to a GADT with one data constructor. Nevertheless, we can still show that a GADT with more than one data constructor can be reduced to a GADT using only existentials and the `Eql` GADT, and also derive an initial algebra semantics for it.

The basic idea is to treat each data constructor individually. That is, we use the same technique as we used for single data constructor GADTs to convert the type of each data constructor of a GADT `G f1 f2 h1 h2` into a type whose codomain is of the form `G f1 f2 h1 h2 a`, and thus avoid the nesting of functors in the data constructors' codomain types. Concretely, we transform the GADT `G f1 f2 h1 h2` into the following equivalent GADT:

```
data NG f1 f2 h1 h2 a where
  NGCon1 :: K f1 h1 (NG f1 f2 h1 h2) a ->
                           NG f1 f2 h1 h2 a
  NGCon2 :: K f2 h2 (NG f1 f2 h1 h2) a ->
                           NG f1 f2 h1 h2 a
```

The return types of the data constructors of `NG f1 f2 h1 h2` are the same, and this GADT uses only existentials and the `Eql` GADT. The two data constructors can now be bundled into one in the usual way, so that the interpretation of `NG f1 f2 h1 h2` is the carrier of the initial algebra of the higher-order functor interpreting

```
  newtype SumKs f1 f2 h1 h2 g a
= Inl (K f1 h1 g a) | Inr (K f2 h2 g a)
```

It is not hard to check that `SumKs f1 f2 h1 h2` is indeed an `HFunctor`. We can therefore derive an initial algebra semantics, and hence an initial algebra package, for `NG f1 f2 h1 h2`. Using `toK` and `fromK`, we can derive one for the original GADT `G f1 f2 h1 h2` as well. An example of a GADT with more than one data constructor is the `Fin` GADT from Example 1 below.

### 6.3 GADTs with More Than One Type Parameter

Consider a GADT `G f h1 h2 a b`

```
data G f h1 h2 a b where
  GCon :: f (G f h1 h2) a b ->
             G f h1 h2 (h1 a b) (h2 a b)
```

with two type parameters. In Haskell, the kinds of `h1`, `h2`, `f`, and `G f h1 h2` are

```
h1, h2 :: |*| -> |*| -> |*|
f :: (|*| -> |*| -> *) -> |*| -> |*| -> *
G f h1 h2 :: |*| -> |*| -> *
```

We can treat this GADT by rewriting the type of `GCon` in terms of the type `K` representing left Kan extensions and the `Eql` GADT, although `K` must be generalized to take two type arguments as input. This yields the definition

```
data BiK f h1 h2 g a b = forall c1 c2.
  BiHFunctor f => BiK (Eql (h1 c1 c2) a,
                       Eql (h2 c1 c2) b, f g c1 c2)
```

where `BiHFunctor` is the generalization

```
class BiHFunctor f where
   mhfmap :: MNat g h -> MNat (f g) (f h)

type MNat f g = forall c1 c2. f c1 c2 -> g c1 c2
```

of the `HFunctor` class to two type parameters which is required to capture the structure of `f` here. It is not hard to see that `BiK f h1 h2` is an instance of the `BiHFunctor` class:

```
instance BiHFunctor f =>
  BiHFunctor (BiK f h1 h2) where
    mhfmap k (BiK (p, q, v)) = BiK (p, q, mhfmap k v)
```

The key universal property of the form of Kan extension captured by `BiK` is that there is an isomorphism between the interpretation of the type

```
forall a b. f g a b -> g (h1 a b) (h2 a b)
```

and the interpretation of the type

```
forall a b. BiK f h1 h2 g a b -> g a b
```

The GADT `G f h1 h2 a b` is thus equivalent to the GADT

```
data NG f h1 h2 a b where
  NGCon :: BiK f h1 h2 (NG f h1 h2) a b ->
              NG f h1 h2 a b
```

for which an initial algebra semantics is easily given, since the interpretation of `NG f h1 h2` is the carrier of the initial algebra of the interpretation of `BiK f h1 h2`. An example of a GADT with more than one type parameter is the `Expr` GADT from Example 3.

### 6.4 GADTs Whose Data Constructors Have More Than One Type Parameter

Consider a GADT of the form

```
data G f h a where
  GCon :: f (G f h) a b -> G f h (h a b)
```

where the Haskell kinds of `h`, `f`, and `G f h` are

```
h :: |*| -> |*| -> |*|
f :: (|*| -> *) -> |*| -> |*| -> *
G f h :: |*| -> *
```

Without loss of generality, we may assume that the number of type variables appearing in the domain type `f (G f h) a b` of `GCon` is the same as the number appearing in `GCon`'s return type. We can treat a GADT `G f h` of this form by rewriting the type of `GCon` in terms of the data type `K` representing left Kan extensions and the `Eql` GADT, although `K` must be generalized to allow `h` to take more than one type argument as input. This yields the definition

```
data VK f h g a = forall c1 c2. VHFunctor f =>
                    VK (Eql (h c1 c2) a, f g c1 c2)
```

where `VHFunctor` is the generalization

```
class VHFunctor f where
   vhfmap :: Nat g h -> MNat (f g) (f h)
```

of the `HFunctor` class which allows `f` to return a type constructor parameterized over two type variables. It is not hard to see that `VK f h` is an instance of the `VHFunctor` class:

```
instance VHFunctor f => HFunctor (VK f h) where
  hfmap k (VK (p, v)) = VK (p, vhfmap k v)
```

The key universal property of the form of Kan extension captured by `VK` is that there is an isomorphism between the interpretation of the type

```
forall a b. f g a b -> g (h a b)
```

and the interpretation of the type

```
forall a. VK f h g a -> g a
```

The GADT `G f h a` is thus equivalent to the GADT

```
data NG f h a where
  NGCon :: VK f h (NG f h) a -> NG f h a
```

for which an initial algebra semantics is easily given, as the interpretation of `NG f h` is the carrier of the initial algebra of the interpretation of `VK f h`. An example of a (single parameter) GADT whose data constructor has more than one type parameter is `Term`.[4]

In summary, our techniques extend from basic GADTs to GADTs `G'` of the form

```
G f1 ... fk h11 ... h1n ... hk1 ... hkn
```

where `n` is the number of type arguments `G'` takes, and `G'` is given as a list of data constructors with types of the form

```
fi G' ai1 ... airi ->
  G' (hi1 ai1 ... airi) ... (hin ai1 ... airi)
```

As discussed above, any constructor taking several arguments can be exchanged for a constructor taking exactly one argument.

Thus for every data constructor of every GADT there is an appropriate Kan extension which can be used to trade that data constructor for one based on existentials and the `Eql` GADT, and from which an initial algebra semantics for the original GADT can be derived. Thus, the proliferation of different codings of Kan extensions, which seems a drawback at first, simply reflects the fact that Haskell does not have a proper polymorphic system of kinds. If it did, then only one polykinded Kan extension would be required and we would be able to present the general case from the start rather than a special case followed by a sketch of the general case.

## 7. Examples

In this section we show how to derive initial algebra packages for some familiar GADTs.

EXAMPLE 1. *Consider the GADT of finite sets [Sheard et al. (2005)] given by*

```
data Z
data S a

data Fin a where
     Fz :: Fin (S a)
     Fs :: Fin a -> Fin (S a)
```

*Note that* `Fin Z` *is empty. This GADT is equivalent to the GADT*

[4] An alternative way to handle a data constructor such as `App` which has more type variables in its domain type than in its return type is to wrap the "extra" type variables in the domain type in an existential quantifier. This is justified by observing that a type of the form `forall b. f b -> g`, where `f` is a type parameterized over `b` and `g` is a type in which `b` does not appear, is equivalent to the type `(exists b. f b) -> g`. Combined with the tupling described in Section 6.1, this allows us to trade, for example, the `App` data constructor for the equivalent data constructor `App :: exists b. (Term (b -> a), Term b) -> Term a` in whose domain type only the type variables appearing in the return type of the constructor appear free.

```
foldFin :: (forall a. f (S a)) ->
           (forall a. f a -> f (S a)) -> Nat Fin f
foldFin z s Fz     = z
foldFin z s (Fs t) = s (foldFin z s t)

-- generalized Church encoding
forall f. (forall a. f (S a)) ->
  (forall a. f a -> f (S a)) -> Nat c f

buildFin :: (forall f. (forall a. f (S a)) ->
                       (forall a. f a -> f (S a)) ->
                        Nat c f) -> Nat c Fin
buildFin g = g Fz Fs

-- fold/build rule
foldFin z s . (buildFin g) = g z s
```

**Figure 4.** Initial algebra package for `Fin`.

```
data NFin a where
     NFz :: Lan S One a  -> NFin a
     NFs :: Lan S NFin a -> NFin a
```

*which has initial algebra package*

```
foldNFin :: (forall a. Lan S One a -> f a) ->
              (forall a. Lan S f a -> f a) ->
                Nat NFin f
foldNFin z s (NFz k) = z k
foldNFin z s (NFs t) = s (hfmap (foldNFin z s) t)

buildNFin :: (forall f.
                (forall a. Lan S One a -> f a) ->
                (forall a. Lan S f a -> f a)
                  -> Nat c f) -> Nat c NFin
buildNFin g = g NFz NFs

foldNFin z s . (buildNFin g) = g z s
```

*The definitions of* `foldG` *and* `buildG` *in Section 5 can be instantiated to give the initial algebra package in Figure 4 relative to* `Fin`.

EXAMPLE 2. *Consider again the GADT* `Term` *[Sheard et al. (2005)]*

```
data Term a where
   Const :: a -> Term a
   Pair  :: Term b -> Term c -> Term (b,c)
   App   :: Term (b -> a) -> Term b -> Term a
```

*from Section 2.1. This GADT is equivalent to*

```
newtype Fst a b     = Fst a
newtype HProd g a b = HProd (g a, g b)
newtype Prod a b    = Prod (a, b)
data HL f a b       = HL (f (b -> a), f b)

data NTerm a where
   NConst :: a -> NTerm a
   NPair  :: VK HProd Prod NTerm a -> NTerm a
   NApp   :: VK HL Fst NTerm a -> NTerm a
```

*where* `VK` *is defined as in Section 6.4. Note that* `HProd` *and* `HL` *are instances of the* `VHFunctor` *class:*

```
instance VHFunctor HProd where
  vhfmap k (HProd (u,v)) = HProd (k u, k v)

instance VHFunctor HL where
  vhfmap k (HL (u,v)) = HL (k u, k v)
```

*The GADT* `NTerm` *has initial algebra package*

```
foldNTerm :: (forall a. a -> f a) ->
             (forall a. VK HProd Prod f a -> f a) ->
             (forall a. VK HL Fst f a -> f a) ->
                Nat NTerm f
foldNTerm c p a (NConst v) = c v
foldNTerm c p a (NPair t) =
     p (hfmap (foldNTerm c p a) t)
foldNTerm c p a (NApp t) =
     a (hfmap (foldNTerm c p a) t)

buildNTerm :: (forall f.
 (forall a. a -> f a) ->
 (forall a. VK HProd Prod f a -> f a) ->
 (forall a. VK HL Fst f a -> f a) -> Nat c f) ->
    Nat c NTerm
buildNTerm g = g NConst NPair NApp

foldNTerm c p a . (buildNTerm g) = g c p a
```

*The definitions of* `foldG` *and* `buildG` *from Section 5 can be instantiated to give the initial algebra package in Figure 5 relative to* `Term`.

```
foldTerm :: (forall a. a -> f a) ->
            (forall a b. f a -> f b -> f (a,b)) ->
            (forall a b. f (b -> a) -> f b -> f a) ->
             Nat Term f
foldTerm c p a (Const v)  = c v
foldTerm c p a (Pair u v) = p (foldTerm c p a u)
                              (foldTerm c p a v)
foldTerm c p a (App t u)  = a (foldTerm c p a t)
                              (foldTerm c p a u)

-- generalized Church encoding
forall f. (forall a. a -> f a) ->
   (forall a b. f a -> f b -> f (a,b)) ->
   (forall a b. f (b -> a) -> f b -> f a) ->
      Nat c f

buildTerm :: (forall f.
   (forall a. a -> f a) ->
   (forall a b. f a -> f b -> f (a,b)) ->
   (forall a b. f (b -> a) -> f b -> f a) ->
      Nat c f) -> Nat c Term
buildTerm g = g Const Pair App

-- fold/build rule
foldTerm c p a . (buildTerm g) = g c p a
```

**Figure 5.** Initial algebra package for `Term`.

EXAMPLE 3. *Consider the GADT of polynomial expressions with variables of type* `a` *and coefficients of type* `b` *given by*

```
data Expr a b where
  Var    :: a -> Expr a b
  IConst :: Int -> Expr a Int
  RConst :: Float -> Expr a Float
  PProd  :: Expr a b -> Expr a b -> Expr a b
  SIMul  :: Expr a b -> Int -> Expr a b
  SRMul  :: Expr a b -> Float -> Expr a Float
```

*This GADT is equivalent to the GADT*

```
data NExpr a b where
  NVar    :: a -> NExpr a b
  NIConst :: MK HInt Fst KInt NExpr a b ->
              NExpr a b
  NRConst :: MK  HFloat Fst KFloat NExpr a b ->
              NExpr a b
  NPProd  :: H2Prod NExpr a b -> NExpr a b
  NSIMul  :: H2ProdInt NExpr a b -> NExpr a b
  NSRMul  :: MK H2ProdFloat Fst KFloat NExpr a b ->
              NExpr a b
```

*where* `BiK` *is as defined in Section 6.3 and*

```
newtype KInt       a b = KInt Int
newtype KFloat     a b = KFloat Float

data HInt        g a b = HInt Int
data HFloat      g a b = HFloat Float
data H2Prod      g a b = H2Prod (g a b, g a b)
data H2ProdInt   g a b = H2ProdInt (g a b, Int)
data H2ProdFloat g a b = H2ProdFloat (g a b, Float)
```

*Note that* `HInt`, `HFloat`, `H2Prod`, `H2ProdInt`, *and* `H2ProdFloat` *are instances of the* `BiHFunctor` *class from Section 6.3. Indeed,*

```
instance BiHFunctor HInt where
  mhfmap s (HInt t) = (HInt t)

instance BiHFunctor HFloat where
  mhfmap s (HFloat t) = (HFloat t)

instance BiHFunctor H2Prod where
  mhfmap s (H2Prod (u,v)) = H2Prod (s u, s v)

instance BiHFunctor H2ProdInt where
  mhfmap s (H2ProdInt (u,v)) = H2ProdInt (s u, v)

instance BiHFunctor H2ProdFloat where
  mhfmap s (H2ProdFloat (u,v)) = H2ProdFloat (s u, v)
```

*The GADT* `NExpr` *has initial algebra package*

```
foldNExpr :: (forall a b. a -> f a b) ->
 (forall a b. BiK HInt Fst KInt f a b -> f a b) ->
 (forall a b. BiK HFloat Fst KFloat f a b ->
                  f a b) ->
 (forall a b. H2Prod f a b -> f a b) ->
 (forall a b. H2ProdInt f a b -> f a b) ->
 (forall a b. BiK H2ProdFloat Fst KFloat f a b ->
      f a b) -> NExpr a b -> f a b

foldNExpr v i r p si sr (NVar t)    = v t
foldNExpr v i r p si sr (NIConst t) =
     i (mhfmap (foldNExpr v i r p si sr) t)
foldNExpr v i r p si sr (NRConst t) =
     r (mhfmap (foldNExpr v i r p si sr) t)
foldNExpr v i r p si sr (NPProd  t) =
     p (mhfmap (foldNExpr v i r p si sr) t)
foldNExpr v i r p si sr (NSIMul t)  =
     si (mhfmap (foldNExpr v i r p si sr) t)
foldNExpr v i r p si sr (NSRMul t)  =
     sr (mhfmap (foldNExpr v i r p si sr) t)

buildNExpr :: (forall f.
 (forall a b. a -> f a b) ->
 (forall a b. BiK HInt Fst KInt f a b -> f a b) ->
 (forall a b. BiK HFloat Fst KFloat f a b ->
                 f a b) ->
 (forall a b. H2Prod f a b -> f a b) ->
 (forall a b. H2ProdInt f a b -> f a b) ->
 (forall a b. BiK H2ProdFloat Fst KFloat f a b ->
       f a b) -> MNat c f) -> MNat c NExpr
buildNExpr g =
  g NVar NIConst NRConst NPProd NSIMul NSRMul

foldNExpr v i r p si sr . buildNExpr g =
   g v i r p si sr
```

*The definitions in Section 5 give the initial algebra package in Figure 6 relative to the original GADT* `Expr`.

```
foldExpr :: (forall a b. a -> f a b) ->
  (forall a. Int -> f a Int) ->
  (forall a. Float -> f a Float) ->
  (forall a b. f a b -> f a b -> f a b) ->
  (forall a b. f a b -> Int -> f a b) ->
  (forall a b. f a b -> Float -> f a Float) ->
  forall a b. Expr a b -> f a b
foldExpr v i r p si sr (Var t)      = v t
foldExpr v i r p si sr (IConst t)   = i t
foldExpr v i r p si sr (RConst t)   = r t
foldExpr v i r p si sr (PProd t u)  =
   p (foldExpr v i r p si sr t)
     (foldExpr v i r p  si sr u)
foldExpr v i r p si sr (SIMul  t n) =
   si (foldExpr v i r p si sr t) n
foldExpr v i r p si sr (SRMul  t n) =
   sr (foldExpr v i r p si sr t) n

buildExpr :: (forall f.
  (forall a b. a -> f a b) ->
  (forall a. Int -> f a Int) ->
  (forall a. Float -> f a Float) ->
  (forall a b. f a b -> f a b -> f a b) ->
  (forall a b. f a b -> Int -> f a b) ->
  (forall a b. f a b -> Float -> f a Float) ->
        MNat c f) -> MNat c Expr
buildExpr g = g Var IConst RConst PProd SIMul SRMul

-- fold/build rule
foldExpr v i r p si sr . (buildExpr g) =
  g v i r p si sr
```

**Figure 6.** Initial algebra package for `Expr`.

## 8. Conclusion and Future Work

In this paper we have shown that the standard view of data types as carriers of initial algebras of functors can be extended from algebraic and nested data types to GADTs. We have used this observation to derive an initial algebra semantics and initial algebra packages for GADTs and, thereby, to provide expressive and principled tools for reasoning about, programming with, and improving the performance of programs involving, GADTs. We have also given a constructive demonstration that every GADT can be reduced to one which involves only the equality GADT and existential quantification. Our reduction is local, independent of any particular syntactic presentation of GADTs, and implementable in the host language. Our approach to initial algebra semantics for GADTs is based on an interpretation of them as carriers of initial algebras of higher-order functors which map functors with discrete domains to functors with discrete domains, rather than functors with possibly nondiscrete domains to functors with possibly nondiscrete domains, and on the use of left Kan extensions as a restructuring device. Our use of left Kan extensions here is reminiscent of their use in [Johann & Ghani (2007a), Johann & Ghani (2007b)] to derive initial algebra semantics for nested data types.

The foundations of GADTs and other advanced inductive constructions have been considered from the type-theoretic perspective; see, e.g., [Pfenning & Paulin-Mohring (1990)], which considers inductive types in the Calculus of Constructions. This work gives fold combinators for GADTs, as well as Church encodings which are essentially the special case of our generalized Church encodings obtained by taking the parameter `c` to be the constantly `1`-valued type constructor. The importance of generalised Church encodings is discussed in [Johann & Ghani (2007a), Johann & Ghani (2007b)]. The relationship between our work and that of Pfenning and Paulin-Mohring is the subject for future research, but we see our categorical approach as complementary to their type-theoretic one. Also, our use of left Kan extensions to trade GADT constructors for simpler nested type constructors provides a clean and concise derivation of the folklore result stating that the essence of GADTs is existential quantification coupled with the equality GADT. Our treatment of the foundations of GADTs *as GADTs*, rather than as embedded within more general type theories, is fundamental to our results.

Our main direction for future work involves extending the results of this paper from GADTs, which are indexed by types, to styles of indexed programming which allow more general indices. This paper and [Johann & Ghani (2007a), Johann & Ghani (2007b)] together make clear that the fundamental structure underlying the algebra of nested data types and GADTs is captured by functorial composition and its adjoints, namely left and right Kan extensions. The mathematical basis for generalizing this structure to encompass other forms of indexing is clear. We will therefore consider indexed programming in the context of fibrations.

## Acknowledgments

Patricia Johann is supported in part by NSF grant CCF-0700341. Neil Ghani is supported in part by EPSRC grant EP/C511964/2.

## References

[Bainbridge et al. (1990)] E. S. Bainbridge, P. J. Freyd, A. Scedrov and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science* 70(1) (1990), pp. 35–64.

[Bayley (2001)] I. Bayley. Generic Operations on Nested Datatypes. Ph.D. Dissertation, Univ. of Oxford, 2001. At `http://web.comlab.ox.ac.uk/oucl/research/areas/ap/papers/bayley-thesis.pdf`

[Blampied (2000)] P. Blampied. Structured Recursion for Non-uniform Data-types. Ph.D. Dissertation, Univ. of Nottingham, 2000. At `http://www.cs.nott.ac.uk/Research/fop/blampied-thesis.pdf`

[Bird & Meertens (1998)] Bird, R. and Meertens, L. Nested datatypes. Proc., Mathematics of Program Construction, pp. 52–67, 1998.

[Bird & Paterson (1998)] R. Bird and R. Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming* 9(1) (1998), pp. 77–91.

[Bird & Paterson (1999)] R. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing* 11(2) (1999), pp. 200–222.

[Cheney & Hinze (2003)] J. Cheney and R. Hinze. First-class phantom types. At `http://www.informatik.uni-bonn.de/~ralf/publications/Phantom.pdf`

[Dybjer (1994)] P. Dybjer. Inductive Families. *Formal Aspects of Computing* 6(4), pp. 440–465, 1994.

[Gill et al. (1993)] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. Proceedings, Functional Programming Languages and Computer Architecture, pp. 223–232, 1993.

[Ghani et al. (2005)] N. Ghani, P. Johann, T. Uustalu, and V. Vene. Monadic augment and generalised short cut fusion. Proceedings, International Conference on Functional Programming, pp. 294–305, 2005.

[Ghani et al. (2003)] N. Ghani, T. Uustalu, and V. Vene. Build, augment and destroy. Universally. Proceedings, Asian Symposium on Programming Languages, pp. 327–347, 2003.

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

[Johann & Ghani (2007b)] P. Johann and N. Ghani. Programming with Nested Types. Submitted, 2007.

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

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

[Martin et al. (2004)] C. Martin, J. Gibbons, and I. Bayley. Disciplined efficient generalised folds for nested datatypes. *Formal Aspects of Computing* 16(1) (2004), pp. 19–35.

[McBride (2004)] C. McBride. Epigram: Practical programming with dependent types. Proceedings, 5th International Summer School on Advanced Functional Programming, 2004. At `http://www.e-pig.org/downloads/epigram-notes.pdf`

[Omega] The Omega Download Page. `http://web.cecs.pdx.edu/~sheard/Omega/index.html`

[Pfenning & Paulin-Mohring (1990)] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. Proceedings, Mathematical Foundations of Programming Semantics, pp. 209-228, 1990.

[Sheard et al. (2005)] T. Sheard, J. Hook, and N. Linger. GADTs + extensible kinds = dependent programming. At `http://www.cs.pdx.edu/~sheard/papers/GADT+ ExtKinds.ps`

[Sheard & Pasalic (2004)] T. Sheard and E. Pasalic. Meta-programming with built-in type equality. Proceedings, Logical Frameworks and Meta-languages, 2004. At `http://homepage.mac.com/pasalic/p2/papers/LFM04 .pdf`

[Sulzmann & Wang (2004)] M. Sulzmann and M. Wang. A systematic translation of guarded recursive data types to existential types. At `http://www.comp .nus.edu.sg/~sulzmann/research/ms.html`

[Sulzmann & Wang (2005)] M. Sulzmann and M. Wang. Translating generalized algebraic data types to System F. Manuscript, 2005. At `http://www.comp. nus.edu.sg/~sulzmann/manuscript/simple-translate-gadts. ps`

[Svenningsson (2002)] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. Proceedings, International Conference on Functional Programming, pp. 124–132, 2002.

[Takano & Meijer (1995)] A. Takano and E. Meijer. Shortcut deforestation in calculational form. Proceedings, Functional Programming Languages and Computer Architecture, pp. 306–313, 1995.

[Xi et al. (2003)] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. Proceedings, Principles of Programming Languages, pp. 224–235, 2003.

[Xi & Pfenning (1999)] H. Xi and F. Pfenning. Dependent types in practical programming. Proceedings, Principles of Programming Languages, pp. 214–227, 1999.