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 met theoretical 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.
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-kindred types, as in (extensions of) Haskell, and with user-definable kinds, as in Ω [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
construct the intermediate structure introduced by \texttt{build} and immediately consumed by \texttt{fold}. Local program transformations [Gill et al. (1993), Johann (2002), Svenningsson (2002)] based on \texttt{fold}/\texttt{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 \texttt{fold} combinator, a single generic \texttt{build} combinator, and a single generic \texttt{fold}/\texttt{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 initial algebra packages. Furthermore, 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. 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 co-ends, 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 $C$ 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 \texttt{fold} and \texttt{build} combinators, Church encodings, and \texttt{fold}/\texttt{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 $\mathcal{C} \rightarrow \mathcal{C}$, 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 $[\mathcal{C} \rightarrow \mathcal{C}] \rightarrow \mathcal{C} \rightarrow \mathcal{C}$. Our solution to this problem lies in modeling GADTs as carriers of initial algebras of higher-order functors $[\mathcal{C} \rightarrow \mathcal{C}] \rightarrow [\mathcal{C} \rightarrow \mathcal{C}]$, where $\mathcal{C}$ is the discrete category derived from $\mathcal{C}$. The second issue is that an arbitrary GADT $\mathcal{G}$ has data constructors whose return types are not of the form $\mathcal{G} a$ for some type variable $a$, but rather are of the form $\mathcal{G} (a a)$. (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 $\mathcal{G} (a a)$ for an equivalent one all of whose constructors have return types of the form $\mathcal{G} a$. 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 \texttt{fold} and \texttt{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 extend 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 via 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 C interprets the kind *) as a target language to show that GADTs are not obviously functors C → 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 (|C| → C) → |C| → C (although, as just noted, not necessarily higher-order functors (C → C) → C → 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. Consider, for example, the data type of lists, realizable in Haskell as

```
data List a = Nil | Cons (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 :: * -> * 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 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 C → C which is the carrier of the initial algebra of a functor (C → C) → C → C. The first question which arises is thus whether or not every GADT can be interpreted as a functor C → C.

To answer this question, consider the situation in Haskell, where the category 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.
The above considerations lead us to try to model a GADT as a Haskell functor. For example, we would require a declaration:

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

The clause for terms of the form `Pair x y` would require a declaration that `fmap` should change the data in a term while preserving its structure, we should have:

```haskell
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 `C' -> C` for a category `C'` distinct from the category `C`. A natural candidate for `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 `C'` to be those of `C`.

At first glance this seems to have gotten us nowhere, but we have not yet fully specified the category `C'`. In addition to specifying the objects of `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 `C'` to have as morphisms only the identities. That is, we take the category `C'` to be the discrete category derived from `C`. Writing `|C|` for this category, we observe that every function from the objects of `|C|` to `C` is trivially a functor `|C| -> C`. We now turn our attention to showing that every GADT can be modeled as the carrier of the initial algebra of a functor `(|C| -> C) -> |C| -> C`.

#### 3.2 Basic GADTs and Higher-order Functors

To keep syntactic overhead to a minimum, and to highlight the key ideas underlyng 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:

```haskell
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:

```haskell
class HFunctor f where
  hmap :: (Functor g => a -> b) -> f g a -> f g b
  hmap :: 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 `hmap` function; that a higher-order functor maps natural transformations to natural transformations is captured by the requirement that an `HFunctor` supports an `hmap` function. The type of natural transformations can be given in Haskell by:

```haskell
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 a` to `h a`, 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 `hmap` 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`.

```haskell
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 `(|C| -> C) -> |C| -> C`, then in order for `G f h` and its data constructor `GCon` to be well-kinded we should have in Haskell:

```haskell
h :: |*| -> |*|,
f :: (|*| -> |*|) -> |*| -> *
```

We can tailor the `HFunctor` class above to accommodate higher-order functors of kind `(|*| -> *) -> |*| -> *`. This amounts to eliminating the `hmap` 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:

```haskell
class HFunctor f where
  hmap :: 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 `(|C| -> C) -> |C| -> C` such that the carrier of its initial algebra interprets `G f h`.

In a parametric model, the interpretation of the type `G f h` a -> `G f h (h a)` of `GCon` is isomorphic to the interpretation of the type ` Lan h f g a` where `data Lan h g c = forall b. Lan (Eq (b c) g b)` is the Haskell representation of the left Kan extension [MacLane (1971)] of `g` along `h`, and `Eq (b c)` is the equality GADT.

```haskell
data Eq a b where
  Refl :: Eq a a
```

The use of `Eq` in the definition of `Lan` reflects the fact that, in Haskell, the domain of `h` is the discrete category `|*|`. Con-
structing an element of type \( G f h c \) requires using \( \text{GCon} \), and thus finding a type \( b \) such that \( h b = c \) and giving an element of \( f (G f h b) \) to which \( \text{GCon} \) 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 \( \text{Lan} h (f (G f (h b))) \) 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

\[
K f h g a = \text{Lan} h (f g) a
\]

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:

\[
data K f h g a \equiv \forall b. \text{Lan} h (f g) a
\]

We can verify that \( K f h \) is indeed an \( \text{HFunctor} \) by observing that it is the composition of \( f \), which is an \( \text{HFunctor} \) by assumption, and \( \text{Lan} h \), which is an \( \text{HFunctor} \) as shown here:

\[
\text{instance \( \text{HFunctor} \)} (\text{Lan} h) \text{ where}
\]

\[
\text{hfmap } k \text{ (} \text{Lan} (p, v) \text{)} = \text{Lan} (p, k v)
\]

Composition of \( \text{HFunctors} \) can be coded in Haskell using

\[
\text{newtype \( \text{HFunctor} \) \( g \), \( \text{HFunctor} \) \( h \)} \Rightarrow
\]

\[
\text{HComp } g h k a = \text{HComp } (g (h k) a)
\]

The following instance declaration shows that the composition of two \( \text{HFunctors} \) is again an \( \text{HFunctor} \):

\[
\text{instance \( \text{HFunctor} \) \( g \), \( \text{HFunctor} \) \( h \)} \Rightarrow
\]

\[
\text{HFunctor } (g \ 'HComp' \ h) \text{ where}
\]

\[
\text{hfmap } k \text{ (} \text{HComp } t \text{)} = \text{HComp } (\text{hfmap } (\text{hfmap } k) \ t)
\]

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

\[
\text{instance \( \text{HFunctor} \) \( f \), \( \text{HFunctor} \) \( (\text{K } f h) \)} \Rightarrow
\]

\[
\text{HFunctor } (\text{K } f h \ 'HComp' \ h) \text{ where}
\]

\[
\text{hfmap } k \text{ (} \text{K } f h \text{ )} = \text{K } f h (\text{hfmap } k \ v)
\]

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

\[
data \text{NG } f h a \text{ where}
\]

\[
\text{NGCon } :: \text{K } f h \text{ (} \text{NG } f h a \text{)} a \Rightarrow \text{NG } f h a
\]

This follows from the isomorphism between the interpretation of \( \forall c. \text{Lan} h g c \Rightarrow f c \) and the interpretation of \( \forall c. \text{Lan} h g c \Rightarrow (f h c) \) which holds for all \( h, g, \) and \( f \). This isomorphism can be coded in Haskell as

\[
\text{toLan } :: (\forall c. \text{Lan} h g c \Rightarrow f c) \Rightarrow
\]

\[
\text{Lan} h g c \Rightarrow f c
\]

\[
\text{toLan } s (\text{Lan} (\text{Refl}, v)) = s v
\]

\[
\text{fromLan } :: (\forall c. \text{Lan} h g c \Rightarrow f c) \Rightarrow
\]

\[
g c \Rightarrow f (h c)
\]

\[
\text{fromLan } s t = s (\text{Lan} (\text{Refl}, t))
\]

For our left Kan extension \( K, \text{toLan} \) and \( \text{fromLan} \) specialize to the following functions giving an isomorphism between the interpretation of \( \forall c. K f h g c \Rightarrow k c \) and the interpretation of \( \forall c. f g c \Rightarrow k c \):

\[
toK :: \text{HFunctor } f \Rightarrow (\forall c. f g c \Rightarrow k (h c)) \Rightarrow K f h g c \Rightarrow k c
\]

\[
toK s (K (\text{Refl}, v)) = s v
\]

\[
\text{fromK } :: \text{HFunctor } f \Rightarrow (\forall c. K f h g c \Rightarrow k c) \Rightarrow f g c \Rightarrow k (h c)
\]

\[
\text{fromK } s t = s (K (\text{Refl}, t))
\]

Thus, giving data appropriate for input to \( \text{GCon} \) is equivalent to giving data appropriate for input to \( \text{NGCon} \). Since the data constructors of these types contain exactly the same information, they generate exactly the same \( \text{GADT} \).

In light of the above, we can write \( G f h \) as \( \text{Ma } (K f h) \) where

\[
\text{newtype } \text{Mu } f a \equiv \text{Inn} (f (\text{Mu } f a))
\]

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 \( \text{GADTs} \).

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

\[
\text{newtype } M f = \text{Inn} (f (M f))
\]

\[
\text{fold } :: \text{Functor } f \Rightarrow (f a \Rightarrow a) \Rightarrow M f \Rightarrow a
\]

\[
\text{fold } h (\text{Inn } k) = h (\text{funmap } (\text{fold } h) k)
\]

\[
\text{build } :: \text{Functor } f \Rightarrow
\]

\[
(\forall b. (f b \Rightarrow b) \Rightarrow b) \Rightarrow M f
\]

\[
\text{build } g = \text{g Inn}
\]

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 \( f a \Rightarrow a \), and \( g \) is any function of closed type \( f b \Rightarrow b \), 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:

\[
\text{fold } h (\text{build } g) = g h
\]

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

\[\text{3 Unfortunately, we cannot use the same fixed point operator here as in Section 3. This is because } f \text{ is not an } \text{HFunctor} \text{ in this case, which is problematic because Haskell lacks proper polymorphic kinding.}\]
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr c n x = case x 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)
-- Church encoding
foldr :: (a -> b -> b) -> b -> [a] -> b
foldT l b (buildT g) = g l b
-- fold/build rule
buildT g = g Leaf Branch
buildT :: (forall b. (a -> b) -> Tree a)
-- Church encoding
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 folds are true folds, that their Church encodings really en-
code 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 pa-
er 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 recon-
struction 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 $F : C \rightarrow C$ for some suitable
category $C$. In more detail, suppose we have fixed a category $C$.
An algebra for a functor $F : C \rightarrow C$ (or, simply, an $F$-algebra)
is a pair $(A, h)$ where $A$ is an object of $C$ and $h : FA \rightarrow A$ is
a morphism of $C$. 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 $f : A \rightarrow B$ such that the following diagram commutes:

\[
\begin{array}{ccc}
FA & \xrightarrow{Ff} & FB \\
\downarrow{h} & & \downarrow{g} \\
A & \xrightarrow{f} & B
\end{array}
\]

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 ini-
tial $F$-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 isomor-
phism. We write $(\mu F, \psi)$ for the initial $F$-algebra comprising the
fixed point $\mu F$ and the isomorphism $\psi : F(\mu F) \rightarrow \mu F$.

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 $F$. Intuitively 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 $h : \mu F \rightarrow A$ makes the following diagram
commute:

\[
\begin{array}{ccc}
\mu F & \xrightarrow{Ff} & FA \\
\downarrow{h} & & \downarrow{f} \\
A & \xrightarrow{fold h} & A
\end{array}
\]

From this diagram, we see that fold : $(FA \rightarrow A) \rightarrow \mu F \rightarrow A$ and
that fold $h$ satisfies fold $h$ (in $t$) = $h$ ($F$ (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 fusi-
ion 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 $\psi \cdot$ fold $h$ = fold $h'$. But note that fold fusion [Bayley (2001),
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 in-
ductive 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-
don” 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 $C$ 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 $C$, then the forgetful functor $U_F$ maps $F$-algebras to objects in $C$ by forgetting the $F$-structure. That is, $U_F$ maps an $F$-algebra $(A, h)$ to its carrier $A$, and maps an $F$-algebra homomorphism $f : A \rightarrow B$ between $F$-algebras $(A, h)$ and $(B, g)$ to the morphism $f : A \rightarrow B$ in $C$. If $C$ is an object in $C$, then a $U_F$-cone for $C$ consists, for every $F$-algebra $(A, h)$, a morphism $\nu((A, h)) : C \rightarrow A$ in $C$ such that, for every $F$-algebra map $f : A \rightarrow B$, we have $\nu((B, g)) = f \circ \nu((A, h))$.

$$
\begin{array}{c}
(A, h) \\
\downarrow f \\
\nu((B, g)) \\
(C)
\end{array}
$$

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 $\forall x. (F x \rightarrow x) \rightarrow C \rightarrow x$. A $U_F$-cone morphism $g : (C, \nu) \rightarrow (D, \mu)$ between $U_F$-cones $(C, \nu)$ and $(D, \mu)$ is a morphism $g : C \rightarrow D$ in $C$ 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 $C$ 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 fold : $(F x \rightarrow x) \rightarrow \mu F \rightarrow x$.
- The unique mediating morphism from the vertex $C$ of a $U_F$-cone to the vertex $\mu F$ of the $U_F$-limit defines build : $(\forall x. (F x \rightarrow x) \rightarrow C \rightarrow x) \rightarrow C \rightarrow \mu F$ to be build $g = g \circ m$. This justifies the definition of the build combinator given in Section 4.1.
- Correctness of the fold/build rule follows from the fact that fold $h \cdot build g = g \cdot h$, 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 $c \rightarrow M f$ and the interpretation of its generalized Church en-

coding forall x. (f x \rightarrow x) \rightarrow c \rightarrow 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 fold $\prime$ $m$ $h$ for fold $h$ $m$ we have

$$
\begin{align*}
fold' &:: \mu F \rightarrow \forall x.(F x \rightarrow x) \rightarrow x \\
build' &:: (\forall x.(F x \rightarrow x) \rightarrow x) \rightarrow \mu F
\end{align*}
$$

From this we see that correctness of the fold/build rule for inductive types codes one half of the requirement that build and 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 builds 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 :: NG 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

$$
foldNG :: (HFunctor f, Functor h) => Nat (K f h a) a -> Nat (NG f h) a
$$

doNG m = NGCon u = m (hfmap (foldNG m) u)

generalized Church encoding

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

and the build combinator

$$
buildNG :: (HFunctor f, Functor h) => Nat (K f h a) a -> Nat c a
$$

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

$$
foldNG m \cdot buildNG g = g \circ m
$$

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) => Nat (K f h a) a -> Nat (NG 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 -> 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

\[
\text{data BiK } f \ h1 \ h2 \ g \ a \ b = \text{forall c1 c2}.
\]

\[
\text{BiHFunctor } f => \text{BiK } (\text{Eql } (h1 \ c1 \ c2) \ a), \ Eql (h2 \ c1 \ c2) \ b, \ f \ g \ c1 \ c2)
\]

where \text{BiHFunctor} is the generalization

\[
\text{class BiHFunctor } f \ where
\]

\[
\text{mhfmap } :: \text{MNat } g \ h \rightarrow \text{MNat } (f \ g) \ (f \ h)
\]

\[
\text{type } \text{MNat } f \ g = \text{forall c1 c2. } f \ c1 \ c2 \rightarrow g \ c1 \ c2
\]

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

\[
\text{instance BiHFunctor } f => \text{BiK } (\text{Eql } (h1 \ c1 \ c2) \ a, \ Eql (h2 \ c1 \ c2) \ b, \ f \ g \ c1 \ c2)
\]

\[
\text{BiHFunctor is the generalization}
\]

\[
\text{class BiHFunctor } f \ where
\]

\[
\text{mhfmap } :: \text{MNat } g \ h \rightarrow \text{MNat } (f \ g) \ (f \ h)
\]

\[
\text{type } \text{MNat } f \ g = \text{forall c1 c2. } f \ c1 \ c2 \rightarrow g \ c1 \ c2
\]

The key universal property of the form of Kan extension captured by \( \text{BiK} \) is that there is an isomorphism between the interpretation of the type

\[
\text{forall a b. } f \ g \ a \ b \rightarrow g \ (h1 \ a \ b) (h2 \ a \ b)
\]

and the interpretation of the type

\[
\text{forall a b. } \text{BiK } f \ h1 \ h2 \ g \ a \ b \rightarrow g \ a \ b
\]

The GADT \( G f h1 h2 a b \) is thus equivalent to the GADT

\[
\text{data NG } f h1 h2 a where
\]

\[
\text{NGCon } :: \text{BiK } f \ h1 h2 \ (\text{NG } f \ h1 h2) \ a \ b \rightarrow 
\]

\[
\text{NG } f h1 h2 a b
\]

for which an initial algebra semantics is easily given, as the interpretation of \( \text{NG } f h1 h2 \) is the carrier of the initial algebra of the interpretation of \( \text{VK } f h1 h2 \). An example of a GADT with more than one type parameter is the \text{Expr GADT} from Example 3.

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

Consider a GADT of the form

\[
\text{data } G f h a \ where
\]

\[
\text{GCon } :: f \ (G f h) \ a \ b \rightarrow G f h \ (h a b)
\]

where the Haskell kinds of \( f, h \), and \( G f h \) are

\[
h :: *[*] \rightarrow *[*] \rightarrow *[*]
\]

\[
f :: ([*] \rightarrow [*]) \rightarrow ([*] \rightarrow [*]) \rightarrow *
\]

\[
G f h :: *[*] \rightarrow *
\]

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 \text{GCon} is the same as the number appearing in \text{GCon}'s return type. We can treat a GADT \( G f h \) of this form by rewriting the type of \text{GCon} in terms of the data type \( k \) representing left Kan extensions and the \text{Eql GADT}, although \( k \) must be generalized to allow \( h \) to take more than one type argument as input. This yields the definition

\[
\text{data VK } f h g a = \text{forall c1 c2. } \text{VHFunctor } f => \text{VK } (\text{Eql } (h1 \ c1 \ c2) \ a, \ f \ g \ c1 \ c2)
\]

where \text{VHFunctor} is the generalization

\[
\text{class VHFunctor } f \ where
\]

\[
\text{vhfmap } :: \text{Nat } g \ h \rightarrow \text{MNat } (f \ g) \ (f \ h)
\]

of the \text{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 \text{VHFunctor} class:

\[
\text{instance VHFunctor } f => \text{HFunctor } (VK \ f h) \ where
\]

\[
\text{vhfmap } k \ (\text{VK } (p, q, v)) = \text{VK } (p, \text{vhfmap } k \ v)
\]

The key universal property of the form of Kan extension captured by \( \text{VK} \) is that there is an isomorphism between the interpretation of the type

\[
\text{forall a b. } f \ g \ a \ b \rightarrow g \ (h a b)
\]

and the interpretation of the type

\[
\text{forall a. } \text{VK } f h g \ a \rightarrow g \ a
\]

The GADT \( G f h a \) is thus equivalent to the GADT

\[
\text{data } NG f h a \ where
\]

\[
\text{NGCon } :: \text{VK } f h (NG f h) \ a \rightarrow NG f h a
\]

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

In summary, our techniques extend from basic GADTs to GADTs \( G' \) of the form

\[
G f1 \ ... \ fkn \ h11 \ ... \ h1n \ ... \ hkn
\]

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

\[
fi G' a11 \ ... \ airi ->
\]

\[
G' (h11 a11 \ ... \ airi) ... (h1n a1n \ ... \ airi)
\]

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 \text{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**

\[
\text{data Z}
\]

\[
\text{data S a}
\]

\[
\text{data Fin a where}
\]

\[
\text{Fz :: Fin } (S a)
\]

\[
\text{Fs :: Fin } a \rightarrow \text{Fin } (S a)
\]

\[
\text{Note that Fin Z is empty. This GADT is equivalent to the GADT}
\]

\*An alternative way to handle a data constructor such as \text{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 \text{forall b. } f \ g \ a \ b \rightarrow g \ (h a b) \ where \( f \ a \) is a type parameterized over \( b a \) and \( g \) is a type in which \( b \) does not appear, is equivalent to the type \( (\exists b) \ (f \ g) \ a \ b \rightarrow g \) Combined with the tupling described in Section 6.1, this allows us to trade, for example, the \text{App} data constructor for the equivalent data constructor \text{App :: exists b. } (\text{Term } b \rightarrow a), \text{Term } b \rightarrow \text{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)) -> 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)) -> Nat c f
buildFin :: (forall f. (forall 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) -> 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) -> 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

foldTerm :: (forall a. a -> 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) -> Nat c f
buildTerm :: (forall f. (forall a. a -> 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.

vhfmap k (HL (u,v)) = HL (k u, k v)
The GADT NTerm has initial algebra package
foldNTerm :: (forall a. 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) -> Nat NTerm f)
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.

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, s v)
instance BiHFunctor H2ProdFloat where
  mhfmap s (H2ProdFloat (u,v)) = H2ProdFloat (s u, s 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. BiK H2Prod Fst KFloat f a b -> f a b) ->
  (forall a b. BiK H2ProdFloat Fst KFloat f 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) =
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 g = g NVar NIConst NRConst NPProd NSIMul NSRMul

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

Figure 6. Initial algebra package for NExpr.

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.

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


