

(Deep) Induction for GADTs

ANONYMOUS AUTHOR(S)

Abstract

1 INTRODUCTION

2 DEEP INDUCTION FOR ADTs AND NESTED TYPES

2.1 Syntax of ADTs and nested types

(Polynomial) algebraic data types (ADTs), both built-in and user-defined, have long been at the core of functional languages such as Haskell, ML, Agda, Epigram, and Idris. ADTs are used extensively in functional programming to structure computations, to express invariants of the data over which computations are defined, and to ensure the type safety of programs specifying those computations. ADTs include unindexed types, such as the type of natural numbers, and types indexed over other types, such as the quintessential example of an ADT, the type of lists (here coded in Agda) ([Ask Daniel which flavor of syntax, paper as literate Agda, naming conventions?](#))

$$\begin{aligned} \text{data List } (a : \text{Set}) : \text{Set where} \\ \text{Nil} &: \text{List } a \\ \text{Cons} &: a \rightarrow \text{List } a \rightarrow \text{List } a \end{aligned} \quad (1)$$

Notice that all occurrences of List in the above encoding are instantiated at the same index a . Thus, the instances of List at various indices are defined independently from one another. That is a defining feature of ADTs: an ADT defines a *family of inductive types*, one for each index type.

Over time, there has been a notable trend toward data types whose non-regular indexing can capture invariants and other sophisticated properties that can be used for program verification and other applications. A simple example of such a type is given by Bird and Meertens' [[Bird and Meertens 1998](#)] prototypical *nested type*

$$\begin{aligned} \text{data PTree } (a : \text{Set}) : \text{Set where} \\ \text{PLeaf} &: a \rightarrow \text{PTree } a \\ \text{PNode} &: \text{PTree } (a \times a) \rightarrow \text{PTree } a \end{aligned} \quad (2)$$

of perfect trees, which can be thought of as constraining lists to have lengths that are powers of 2. In the above code, the constructor PNode uses data of type PNode $(a \times a)$ to construct data of type PNode a . Thus, it is clear that the instantiations of PNode at various indices cannot be defined independently, so that the entire family of types must actually be defined at once. A nested type thus defines not a family of inductive types, but rather an *inductive family of types*.

Nested types include simple nested types, like perfect trees, none of whose recursive occurrences occur below another type constructor, and *truly* nested types, such as the nested type

$$\begin{aligned} \text{data Bush } (a : \text{Set}) : \text{Set where} \\ \text{BNil} &: \text{Bush } a \\ \text{BCons} &: a \rightarrow \text{Bush } (\text{Bush } a) \rightarrow \text{Bush } a \end{aligned} \quad (3)$$

of bushes, whose recursive occurrences appear below their own type constructors. Note that, while the constructors of a nested type can contain occurrences of the type instantiated at any index, the return types of its constructors still have to be the same type instance of the type being

defined. In other words, all constructors of $\text{PTree } a$ have to return an element of type $\text{PTree } a$, and all constructors of $\text{Bush } a$ have to return an element of type $\text{Bush } a$.

2.2 Induction principles for ADTs and nested types

An induction principle for a data type allows proving that a predicate holds for every element of that data type, provided that it holds for every element inductively produced by the type's constructors. In this paper, we are interested in induction principles for proof-relevant predicates. A proof-relevant predicate on a type a is a function $a \rightarrow \text{Set}$ (where Set is the type of sets) mapping each $x : a$ to the set of proofs that the predicate holds for x . For example, the induction principle for List is

$$\begin{aligned} \forall (a : \text{Set}) (P : \text{List } a \rightarrow \text{Set}) \rightarrow P \text{ Nil} \rightarrow (\forall (x : a) (xs : \text{List } a) \rightarrow P \text{ xs} \rightarrow P (\text{Cons } x \text{ xs})) \\ \rightarrow \forall (xs : \text{List } a) \rightarrow P \text{ xs} \quad (4) \end{aligned}$$

Note that the data inside a structure of type List is treated monolithically (i.e., ignored) by this induction rule. Indeed, the induction rule inducts over only the top-level structures of data types, leaving any data internal to the top-level structure untouched. Since this kind of induction principle is only concerned with the structure of the type, and unconcerned with the contained data, we will then refer to it as *structural induction*.

We can extend such a structural induction principle to some nested types, such as PTree . The only difference from the induction principle for ADTs is that, since a nested type is defined as a whole inductive family of types at once, its induction rule has to necessarily involve a polymorphic predicate. Thus, the induction rule for PTree is

$$\begin{aligned} \forall (P : \forall (a : \text{Set}) \rightarrow \text{PTree } a \rightarrow \text{Set}) \rightarrow (\forall (a : \text{Set}) (x : a) \rightarrow P a (P \text{Leaf } x)) \\ \rightarrow (\forall (a : \text{Set}) (x : \text{PTree } (a \times a)) \rightarrow P (a \times a) x \rightarrow P a (P \text{Node } x)) \\ \rightarrow \forall (a : \text{Set}) (x : \text{PTree } a) \rightarrow P a x \quad (5) \end{aligned}$$

Structural induction principles cannot be extended to truly nested types, such as Bush . Instead, for such data types it is necessary to use a *deep induction* principle [Johann and Polonsky 2020]. Such a principle, unlike structural induction, inducts over all of the structured data present, by traversing not just the outer structure with a predicate P , but also each data element contained in the data type with a custom predicate Q . This additional predicate is lifted to predicates on any internal structure containing these data, and the resulting predicates on these internal structures are lifted to predicates on any internal structures containing structures at the previous level, and so on, until the internal structures at all levels of the data type definition, including the top level, have been so processed. Satisfaction of a predicate by the data at one level of a structure is then conditioned upon satisfaction of the appropriate predicates by all of the data at the preceding level.

For example, the deep induction rule for Bush is

$$\begin{aligned} \forall (P : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow \text{Bush } a \rightarrow \text{Set}) \rightarrow (\forall (a : \text{Set}) \rightarrow P a B \text{Nil}) \\ \rightarrow (\forall (a : \text{Set}) (Q : a \rightarrow \text{Set}) (x : a) (y : \text{Bush } (\text{Bush } a)) \\ \rightarrow Q x \rightarrow P (\text{Bush } a) (\text{Bush}^{\wedge} a Q) y \rightarrow P a Q (B \text{Cons } x y)) \\ \rightarrow \forall (a : \text{Set}) (Q : a \rightarrow \text{Set}) (x : \text{Bush } a) \rightarrow \text{Bush}^{\wedge} a Q x \rightarrow P a Q x \quad (6) \end{aligned}$$

where $\text{Bush}^{\wedge} : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow \text{Bush } a \rightarrow \text{Set}$ lifts a predicate Q on data of type a to a predicate on data of type $\text{Bush } a$ asserting that Q holds for every element of type a contained in its

argument bush. It is defined as

$$\begin{aligned} \text{Bush}^{\wedge} a Q \text{ BNil} &= 1 \\ \text{Bush}^{\wedge} a Q (\text{BCons } x y) &= Q x \times \text{Bush}^{\wedge} (\text{Bush } a) (\text{Bush}^{\wedge} a Q) y \end{aligned}$$

Despite deep induction being motivated by the need to produce an induction principle for truly nested types, it can equally be applied to all other ADTs and nested types. For example, the deep induction principle for List is

$$\begin{aligned} \forall (a : \text{Set}) (P : \text{List } a \rightarrow \text{Set}) (Q : a \rightarrow \text{Set}) \\ \rightarrow P \text{ Nil} \rightarrow (\forall (x : a) (xs : \text{List } a) \rightarrow Q x \rightarrow P xs \rightarrow P (\text{Cons } x xs)) \\ \rightarrow \forall (xs : \text{List } a) \rightarrow \text{List}^{\wedge} a Q xs \rightarrow P xs \quad (7) \end{aligned}$$

where $\text{List}^{\wedge} : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow \text{List } a \rightarrow \text{Set}$ lifts a predicate Q on data of type a to a predicate on data of type $\text{List } a$ asserting that Q holds for every element of its argument list. Finally, the deep induction rule for PTree is

$$\begin{aligned} \forall (P : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow \text{PTree } a \rightarrow \text{Set}) \\ \rightarrow (\forall (a : \text{Set}) (Q : a \rightarrow \text{Set}) (x : a) \rightarrow Q x \rightarrow P a Q (\text{PLeaf } x)) \\ \rightarrow (\forall (a : \text{Set}) (Q : a \rightarrow \text{Set}) (x : \text{PTree } (a \times a)) \rightarrow P (a \times a) (\text{Pair}^{\wedge} a Q) x \rightarrow P a Q (\text{PNode } x)) \\ \rightarrow \forall (a : \text{Set}) (Q : a \rightarrow \text{Set}) (x : \text{PTree } a) \rightarrow \text{PTree}^{\wedge} a Q x \rightarrow P a Q x \quad (8) \end{aligned}$$

where $\text{Pair}^{\wedge} : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow a \times a \rightarrow \text{Set}$ lifts a predicate Q on a to a predicate on pairs of type $a \times a$, so that $\text{Pair}^{\wedge} a Q (x, y) = Q x \times Q y$, and $\text{PTree}^{\wedge} : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow \text{PTree } a \rightarrow \text{Set}$ lifts a predicate Q on data of type a to a predicate on data of type $\text{PTree } a$ asserting that Q holds for every element of type a contained in its argument perfect tree.

3 INTRODUCING GADTS

As noted in Subsection 2.1, the return types of the constructors of a nested type have to be the same type instance of the type being defined. As a further generalization of ADTs and nested types, *generalized algebraic data types* (GADTs) [Cheney and Hinze 2003; Sheard and Pasalic 2004; Xi et al. 2003] relax the restriction on the type instances appearing in a data type definition by allowing their constructors both to take as arguments *and return as results* data whose types involve type instances of the GADT other than the one being defined.

GADTs are used in precisely those situations in which different behaviors at different instances of a data type are desired. This is achieved by allowing the programmer to give the type signatures of the GADT's data constructors independently, and then using pattern matching to force the desired type refinement. Applications of GADTs include generic programming, modeling programming languages via higher-order abstract syntax, maintaining invariants in data structures, and expressing constraints in embedded domain-specific languages. GADTs have also been used, e.g., to implement tagless interpreters [Pasalic and Linger 2004; Peyton Jones et al. 2006; Pottier and Régis-Gianas 2006], to improve memory performance [Minsky 2015], and to design APIs [Penner 2020].

As a first and notable example of GADT, we consider the the Equal type. This GADT is parametrized by two type indices, but it is only possible to construct a data element if the two indices are instantiated at the same type. In Agda, we code it as

$$\begin{aligned} \text{data Equal } (a b : \text{Set}) : \text{Set where} \\ \text{Refl} : \text{Equal } c c \end{aligned} \quad (9)$$

Equal has thus a single data element when its two type arguments are the same and no data elements otherwise.

A more complex example for a GADT is

$$\begin{aligned} \text{data Seq } (a : \text{Set}) : \text{Set where} \\ \text{Const} : a \rightarrow \text{Seq } a \\ \text{SPair} : \text{Seq } a \rightarrow \text{Seq } b \rightarrow \text{Seq } (a \times b) \end{aligned} \quad (10)$$

which comprises sequences of any type a and sequences obtained by pairing the data in two already existing sequences. Such GADTs can be understood in terms of the Equal data type [Cheney and Hinze 2003; Sheard and Pasalic 2004]. For example, we can rewrite the Seq type as

$$\begin{aligned} \text{data Seq } (a : \text{Set}) : \text{Set where} \\ \text{Const} : a \rightarrow \text{Seq } a \\ \text{SPair} : \exists (b : \text{Set}) (c : \text{Set}). \text{Equal } a (b \times c) \rightarrow \text{Seq } b \rightarrow \text{Seq } c \rightarrow \text{Seq } a \end{aligned} \quad (11)$$

where the requirement that the SPair constructor produce an instance of Seq at a product type has been replaced with the requirement that the instance of Seq returned by SPair is *equal* to a product. This encoding is particularly convenient when representing GADTs as Church encodings [Atkey 2012; Vytiniotis and Weirich 2010].

Add third example here.

4 (DEEP) INDUCTION FOR GADTS

As we have seen in Subsection 2.2, truly nested types do not support a structural induction rule, which is the reason why it was necessary to introduce a deep induction rule supporting them. Consequently, GADTs do not support a structural induction rule either, as they generalize nested types. Still, there is hope for GADTs to support a deep induction rule, like nested types do.

Induction rules, and specifically deep induction rules for nested types, are traditionally derived using the functorial semantics of data types in the setting of a parametric model [Johann and Polonsky 2020]. In particular, relational parametricity is used to validate the induction principle because induction is, itself, a form of unary parametricity, where binary relations have been replaced with predicates, which are essentially unary relations.

Unfortunately, this approach cannot possibly be employed to prove a deep induction rule for GADTs, as these types do not allow for a functorial interpretation, at least in a parametric model [?].

Nevertheless, this paper shows how to extend deep induction to GADTs. We will first demonstrate how to derive the deep induction rule for some example GADTs, and then provide a general principle that works for all GADTs.

4.1 (Deep) induction for Equal

We first derive the induction rule for the Equal type. This will provide a simple case study that will help with the derivation of the induction rule of more complex GADTs. More importantly, since we define GADTs using the Equal type, as for example in Equation 10, this example will be instrumental in the induction rule of other GADTs.

To define the induction rule for Equal we first need a lifting operation

$$\text{Equal}^\wedge : \forall (a : \text{Set}) (b : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow (b \rightarrow \text{Set}) \rightarrow \text{Equal } a b \rightarrow \text{Set}$$

defined as

$$\text{Equal}^\wedge a a Q Q' \text{Refl} = \forall (x : a) \rightarrow \text{Equal } (Q x) (Q' x)$$

The induction rule for Equal has type

$$\begin{aligned} \text{DIEqual} : & \forall (P : \forall (a b : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow (b \rightarrow \text{Set}) \rightarrow \text{Equal } a b \rightarrow \text{Set}) \\ & \rightarrow (\forall (c : \text{Set}) (Q : c \rightarrow \text{Set}) (Q' : c \rightarrow \text{Set}) \rightarrow \text{Equal}^{\wedge} c c Q Q' \text{Refl} \rightarrow P c c Q Q' \text{Refl}) \\ & \rightarrow (\forall (a : \text{Set}) (b : \text{Set}) (Q_a : a \rightarrow \text{Set}) (Q_b : b \rightarrow \text{Set}) (e : \text{Equal } a b) \\ & \rightarrow \text{Equal}^{\wedge} a b Q_a Q_b e \rightarrow P a b Q_a Q_b e) \quad (12) \end{aligned}$$

and it is defined as

$$\text{DIEqual } P \text{ PRefl } a a Q Q' \text{Refl } I = \text{PRefl } a Q Q' I$$

where $\text{PRefl} : \forall (c : \text{Set}) (Q : c \rightarrow \text{Set}) (Q' : c \rightarrow \text{Set}) \rightarrow \text{Equal}^{\wedge} c c Q Q' \text{Refl} \rightarrow P c c Q Q' \text{Refl}$, $Q : a \rightarrow \text{Set}$, $Q' : a \rightarrow \text{Set}$ and $I : \text{Equal}^{\wedge} a a Q Q' \text{Refl}$.

4.2 (Deep) induction for Seq

Next we shall consider the Seq type defined in Equation 11. Again, the first step in deriving its induction rule consists in defining the lifting

$$\text{Seq}^{\wedge} : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow \text{Seq } a \rightarrow \text{Set}$$

which is given by

$$\text{Seq}^{\wedge} a Q (\text{Const } x) = Q x$$

and

$$\begin{aligned} & \text{Seq}^{\wedge} a Q_a (\text{SPair } b c e s_b s_c) \\ = & \exists (Q_b : b \rightarrow \text{Set}) (Q_c : c \rightarrow \text{Set}) \rightarrow \text{Equal}^{\wedge} a (b \times c) Q_a (Q_b \times Q_c) e \times \text{Seq}^{\wedge} b Q_b s_b \times \text{Seq}^{\wedge} c Q_c s_c \end{aligned}$$

We also need the lifting of the type $\text{Pair} = \forall (b : \text{Set}) (c : \text{Set}) \rightarrow b \times c$, which is

$$\text{Pair}^{\wedge} : \forall (b : \text{Set}) (c : \text{Set}) \rightarrow (b \rightarrow \text{Set}) \rightarrow (c \rightarrow \text{Set}) \rightarrow b \times c \rightarrow \text{Set}$$

defined as

$$\text{Pair}^{\wedge} b c Q_b Q_c (y, z) = Q_b y \times Q_c z$$

With these tools, we have an induction rule for Seq

$$\begin{aligned} \text{DISeq} : & \forall (P : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow \text{Seq } a \rightarrow \text{Set}) \\ & \rightarrow (\forall (a : \text{Set}) (Q_a : a \rightarrow \text{Set}) (x : a) \rightarrow Q_a x \rightarrow P a Q_a (\text{Const } x)) \\ & \rightarrow (\forall (a : \text{Set}) (b : \text{Set}) (c : \text{Set}) (Q_a : a \rightarrow \text{Set}) (Q_b : b \rightarrow \text{Set}) (Q_c : c \rightarrow \text{Set}) \\ & (s_b : \text{Seq } b) (s_c : \text{Seq } c) (e : \text{Equal } a (b \times c)) \rightarrow \text{Equal}^{\wedge} a (b \times c) Q_a (\text{Pair}^{\wedge} b c Q_b Q_c) e \\ & \rightarrow P b Q_b s_b \rightarrow P c Q_c s_c \rightarrow P a Q_a (\text{SPair } b c e s_b s_c)) \\ & \rightarrow \forall (a : \text{Set}) (Q_a : a \rightarrow \text{Set}) (s_a : \text{Seq } a) \rightarrow \text{Seq}^{\wedge} a Q_a s_a \rightarrow P a Q_a s_a \quad (13) \end{aligned}$$

We define $\text{DISeq } P \text{ PConst } \text{PSPair } a Q_a s_a I_a$, where $P : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow \text{Seq } a \rightarrow \text{Set}$, $Q_a : a \rightarrow \text{Set}$, $s_a : \text{Seq } a$ and $I_a : \text{Seq}^{\wedge} a Q_a s_a$, by pattern-matching on s_a . Let $s_a = \text{Const } x$ for $x : a$, and define

$$\text{DISeq } P \text{ PConst } \text{PSPair } a Q_a (\text{Const } x) I_a = \text{PConst } a Q_a x I_a$$

Notice that $\text{Seq}^{\wedge} a Q_a (\text{Const } x) = Q_a x$, and thus $I_a : Q_a x$, making the right-hand-side in the above expression type-check. Now, let $s_a = \text{SPair } b c e s_b s_c$ for $e : \text{Equal } a (b \times c)$, $s_b : \text{Seq } b$ and $s_c : \text{Seq } c$, and define

$$\text{DISeq } P \text{ PConst } \text{PSPair } a Q_a (\text{SPair } b c e s_b s_c) (Q_b, Q_c, I_e, I_b, I_c) = \text{PSPair } a b c Q_a Q_b Q_c s_b s_c e I_e p_b p_c$$

where $p_b : P \ b \ Q_b \ s_b$, $p_c : P \ c \ Q_c \ s_c$, $l_e : \text{Equal}^a \ a \ (b \times c) \ Q_a \ (Q_b \times Q_c) \ e$, $l_b : \text{Seq}^a \ b \ Q_b \ s_b$, $l_c : \text{Seq}^a \ c \ Q_c \ s_c$ (notice that $(Q_b, Q_c, l_e, l_b, l_c) : \text{Seq}^a \ a \ Q \ (\text{SPair } b \ c \ e \ x \ y)$, making the right-hand-side in the above expression type-check) and p_b and p_c are defined as follows:

$$\begin{aligned} p_b &= \text{DlSeq } P \ \text{PConst } \text{PSPair } b \ Q_b \ s_b \ l_b : P \ b \ Q_b \ s_b \\ p_c &= \text{DlSeq } P \ \text{PConst } \text{PSPair } c \ Q_c \ s_c \ l_c : P \ c \ Q_c \ s_c \end{aligned}$$

4.3 Third example introduced above

4.4 General case

In this section we will generalize the approach taken in the previous examples and provide a general framework to derive induction rules for GADTs. We restrict our scope to GADTs not featuring nesting, in the sense that their constructors contain no nested calls of the recursive variable, as truly nested types do. The reason for this restriction will be discussed in Section 5.

Consider a generic GADT

$$\begin{aligned} \text{data } G \ (\overline{a : \text{Set}}) : \text{Set} \text{ where} \\ C : F \ G \ \overline{b} \rightarrow G(\overline{K \ \overline{b}}) \end{aligned} \tag{14}$$

which, for ease of notation, we assume to have only one constructor C . Extending the argument to the case of a GADT with multiple constructors presents no difficulty. In the definition above, F has type $(\text{Set}^\alpha \rightarrow \text{Set}) \rightarrow \text{Set}^\beta \rightarrow \text{Set}$ and each K has type $\text{Set}^\beta \rightarrow \text{Set}$. The overline notation denotes a finite list: G depends on a finite list of types \overline{a} of length α , each K depends on a finite list of types \overline{b} of length β , and notice that the length of the list \overline{a} matches that of the list \overline{K} . With that in mind, the constructor C can be rewritten as

$$C : \exists (\overline{b : \text{Set}}) \rightarrow \overline{\text{Equal } a \ (K \ \overline{b})} \rightarrow F \ G \ \overline{b} \rightarrow G \ \overline{a}$$

which is the form we shall use from now on.

In order to state the induction rule for G , we first need to define its associated predicate-lifting function

$$G^\wedge : \forall (\overline{a : \text{Set}}) \rightarrow (\overline{a \rightarrow \text{Set}}) \rightarrow G \ \overline{a} \rightarrow \text{Set}$$

as

$$G^\wedge \ \overline{a} \ \overline{Q_a} \ (C \ \overline{b} \ \overline{e_a} \ x) = \exists (\overline{Q_b : b \rightarrow \text{Set}}) \rightarrow \overline{\text{Equal}^a \ a \ (K \ \overline{b}) \ Q_a \ (K^\wedge \ \overline{b} \ \overline{Q_b}) \ e_a} \times F^\wedge \ G \ \overline{b} \ G^\wedge \ \overline{Q_b} \ x$$

where $Q_a : a \rightarrow \text{Set}$, $e_a : \text{Equal } a \ (K \ \overline{b})$ and $x : F \ G \ \overline{b}$. Notice that we need a type $\text{Equal}^a \ a \ (K \ \overline{b}) \ Q_a \ (K^\wedge \ \overline{b} \ \overline{Q_b}) \ e_a$ for each type a . We also assume to have liftings for F ,

$$\begin{aligned} F^\wedge : \forall (G : \text{Set}^\alpha \rightarrow \text{Set}) (\overline{b : \text{Set}}) \rightarrow (\forall (\overline{a : \text{Set}}) \rightarrow (\overline{a \rightarrow \text{Set}}) \rightarrow G \ \overline{a} \rightarrow \text{Set}) \\ \rightarrow (\overline{b \rightarrow \text{Set}}) \rightarrow F \ G \ \overline{b} \rightarrow \text{Set} \end{aligned}$$

and for each K ,

$$K^\wedge : \forall (\overline{b : \text{Set}}) \rightarrow (\overline{b \rightarrow \text{Set}}) \rightarrow K \ \overline{b} \rightarrow \text{Set}$$

as we could not possibly define an induction rule otherwise. A way to concretely define the predicate-lifting function for a type is to proceed by induction on the structure of the type.

Define the induction rule for G

$$\begin{aligned} \text{DIG} : \forall (P : \forall (a : \text{Set}) \rightarrow (\overline{a \rightarrow \text{Set}}) \rightarrow G \overline{a} \rightarrow \text{Set}) \\ \rightarrow \left(\overline{\forall (a : \text{Set}) (\overline{b : \text{Set}}) (\overline{Q_a : a \rightarrow \text{Set}}) (\overline{Q_b : b \rightarrow \text{Set}}) (\overline{e_a : \text{Equal } a (K \overline{b})}) (x : F G \overline{b})} \right. \\ \left. \rightarrow \overline{\text{Equal}^{\wedge} a (K \overline{b}) Q_a (K^{\wedge} \overline{b} \overline{Q_b}) e_a} \rightarrow F^{\wedge} G \overline{b} P \overline{Q_b} x \rightarrow P \overline{a} \overline{Q_a} (C \overline{b} \overline{e_a} x) \right) \\ \rightarrow \forall (\overline{a : \text{Set}}) (\overline{Q_a : a \rightarrow \text{Set}}) (x : G \overline{a}) \rightarrow G^{\wedge} \overline{a} \overline{Q_a} x \rightarrow P \overline{a} \overline{Q_a} x \quad (15) \end{aligned}$$

as

$$\text{DIG } P \text{ PC } \overline{a} \overline{Q_a} (C \overline{b} \overline{e_a} x) (\overline{Q_b}, \overline{E_a}, L_F) = \text{PC } \overline{a} \overline{b} \overline{Q_a} \overline{Q_b} \overline{e_a} x \overline{E_a} (p \times L_F)$$

where we have $e_a : \text{Equal } a (K \overline{b})$ for each $a, x : F G \overline{b}$, $Q_b : b \rightarrow \text{Set}$ for each b , $E_a : \text{Equal}^{\wedge} a (K \overline{b}) Q_a (K^{\wedge} \overline{b} \overline{Q_b})$ for each a , $L_F : F^{\wedge} G \overline{b} G^{\wedge} \overline{Q_b} x$, and

$$p : \text{PredMap } (F G \overline{b}) (F^{\wedge} G \overline{b} G^{\wedge} \overline{Q_b}) (F^{\wedge} G \overline{b} P \overline{Q_b})$$

is defined by structural induction on F as follows:

- Case $F = F_1 \times F_2$ where F_1 and F_2 have the same type as F . We have that

$$F^{\wedge} G \overline{b} P \overline{Q_b} = \text{Pair}^{\wedge} (F_1 G \overline{b}) (F_2 G \overline{b}) (F_1^{\wedge} G \overline{b} P \overline{Q_b}) (F_2^{\wedge} G \overline{b} P \overline{Q_b})$$

By inductive hypothesis, there exist morphisms of predicates

$$p_1 : \text{PredMap } (F_1 G \overline{b}) ((F_1)^{\wedge} G \overline{b} G^{\wedge} \overline{Q_b}) ((F_1)^{\wedge} G \overline{b} P \overline{Q_b})$$

$$p_2 : \text{PredMap } (F_2 G \overline{b}) ((F_2)^{\wedge} G \overline{b} G^{\wedge} \overline{Q_b}) ((F_2)^{\wedge} G \overline{b} P \overline{Q_b})$$

Thus, we define $p(x_1, x_2)(L_1, L_2) = (p_1 x_1 L_1, p_2 x_2 L_2)$ for $x_1 : F_1 G \overline{b}$, $x_2 : F_2 G \overline{b}$, $L_1 : F_1^{\wedge} G \overline{b} G^{\wedge} \overline{Q_b} x_1$ and $L_2 : F_2^{\wedge} G \overline{b} G^{\wedge} \overline{Q_b} x_2$.

- Case $F = F_1 + F_2$ where F_1 and F_2 have the same type as F . Analogous to case $F = F_1 \times F_2$.
- Case $F = F_1 \rightarrow F_2$ where F_1 and F_2 have the same type as F . We have that

$$F^{\wedge} G \overline{b} G^{\wedge} \overline{Q_b} x = \forall (y : F_1 G \overline{b}) \rightarrow F_2^{\wedge} G \overline{b} G^{\wedge} \overline{Q_b} (x y)$$

$$F^{\wedge} G \overline{b} P \overline{Q_b} x = \forall (y : F_1 G \overline{b}) \rightarrow F_2^{\wedge} G \overline{b} P \overline{Q_b} (x y)$$

By inductive hypothesis, there exist a morphism of predicates

$$p_2 : \text{PredMap } (F_2 G \overline{b}) (F_2^{\wedge} G \overline{b} G^{\wedge} \overline{Q_b}) (F_2^{\wedge} G \overline{b} P \overline{Q_b})$$

Thus, we define $p \times L_F y = p_2(x y)(L_F y)$. [Revise this case!](#)

- Case $F G \overline{b} = G(F_a \overline{b})$ where none of the F_a contains the recursive variable, i.e., $F_a : \text{Set}^{\beta} \rightarrow \text{Set}$ for each a . Thus, $F^{\wedge} G \overline{b} P \overline{Q_b} = P(F_a \overline{b}) (F_a^{\wedge} \overline{b} \overline{Q_b})$. So, p is defined as

$$p = \text{DIG } P \text{ PC } (\overline{F_a \overline{b}}) (\overline{F_a \overline{b} \overline{Q_b}})$$

Notice that the fact that the types F_a do not contain the recursive variable means that G does not feature nesting. As mentioned before, that is a fundamental assumption to make the argument work. Why the argument fails in presence of truly nested GADTs will be shown in the next section.

- Case $F G \overline{b} = H \overline{b}$ where H is a type constructor not containing the recursive variable, i.e., $H : \text{Set}^{\beta} \rightarrow \text{Set}$. In such case, $p : \text{PredMap}(H \overline{b})(H^{\wedge} \overline{b} \overline{Q_b})(H^{\wedge} \overline{b} \overline{Q_b})$ is just the identity morphism of predicates.

- Case $F G \bar{b} = H(\overline{F_c G \bar{b}})$ where H is a type constructor not containing the recursive variable, i.e., $H : \text{Set}^\gamma \rightarrow \text{Set}$, and F_c has the same type as F , for every $c = 1 \dots \gamma$. Moreover, we assume that H has an associated predicate-lifting function,

$$H^\wedge : \forall(\bar{c} : \text{Set}) \rightarrow (\bar{c} \rightarrow \text{Set}) \rightarrow H \bar{c} \rightarrow \text{Set}$$

and that this predicate-lifting function has a map function

$$\text{HLMa} : \forall(\bar{c} : \text{Set})(\overline{Q_c Q'_c} : \bar{c} \rightarrow \text{Set}) \rightarrow \text{PredMap } \bar{c} \overline{Q_c Q'_c} \rightarrow \text{PredMap } (H \bar{c}) (H^\wedge \bar{c} \overline{Q_c}) (H^\wedge \bar{c} \overline{Q'_c})$$

That means that H cannot be a GADT, as those do not have functorial semantics [?], but it can be an ADT or even a nested type [Johann and Polonsky 2019; ?]. Thus,

$$F^\wedge G \bar{b} P \overline{Q_b} = H^\wedge(\overline{F_c G \bar{b}})(\overline{F_c^\wedge G \bar{b} P \overline{Q_b}})$$

By induction hypothesis, there is a morphism of predicates

$$p_c : \text{PredMap } (F_c G \bar{b})(F_c^\wedge G \bar{b} G^\wedge \overline{Q_b})(F_c^\wedge G \bar{b} P \overline{Q_b})$$

for every $c = 1 \dots \gamma$. So, p is defined as

$$p = \text{HLMa } (\overline{F_c G \bar{b}})(\overline{F_c^\wedge G \bar{b} G^\wedge \overline{Q_b}})(\overline{F_c^\wedge G \bar{b} P \overline{Q_b}}) \overline{p_c}$$

Notice that, by proceeding by structural induction as above, we have implicitly defined a grammar for GADTs. In particular, we allow F to feature sums, products, arrow types and (limited) type application.

5 INDUCTION FOR GADTS WITH NESTING

In the previous sections, we derive induction rules for GADTs that do not feature nesting, in the sense that their constructors contain no nested calls of the recursive variable, as truly nested types do. Since both nested types [Johann and Polonsky 2020] and GADTs without nesting admit induction rules, it is just natural to expect that GADTs with nesting would as well. Surprisingly, that is not the case: indeed, induction rules for nested types rely on functorial semantics, but GADTs cannot admit both functorial and parametric semantics at the same time [?]. In this section we will show how induction for GADTs featuring nesting goes wrong by analyzing a concrete example.

Consider the type

$$\begin{aligned} \text{data } G(a : \text{Set}) : \text{Set} \text{ where} \\ C : G(G a) \rightarrow G(a \times a) \end{aligned} \tag{16}$$

whose constructor C can be rewritten as

$$C : \exists(b : \text{Set}) \rightarrow \text{Equal } a (b \times b) \rightarrow G(G b) \rightarrow G a$$

which is the form we shall use from now on. Its predicate-lifting function

$$G^\wedge : \forall(\bar{a} : \text{Set}) \rightarrow (\bar{a} \rightarrow \text{Set}) \rightarrow G \bar{a} \rightarrow \text{Set}$$

is defined as

$$G^\wedge a Q_a (C b e x) = \exists(Q_b : b \rightarrow \text{Set}) \rightarrow \text{Equal}^\wedge a (b \times b) Q_a (\text{Pair}^\wedge b b Q_b Q_b) e \times G^\wedge (G b) (G^\wedge b Q_b) x$$

where $Q_a : a \rightarrow \text{Set}$, $e : \text{Equal } a (b \times b)$ and $x : G(G b)$.

We need to define an induction rule for G . That has to be a term

$$\begin{aligned} \text{DIG} : & \forall (P : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow G a \rightarrow \text{Set}) \\ & \rightarrow \left(\forall (a b : \text{Set}) (Q_a : a \rightarrow \text{Set}) (Q_b : b \rightarrow \text{Set}) (e : \text{Equal } a (b \times b)) (x : G (G b)) \right. \\ & \left. \rightarrow \text{Equal}^{\wedge} a (b \times b) Q_a (\text{Pair}^{\wedge} b b Q_b Q_b) e \rightarrow P (G b) (P b Q_b) x \rightarrow P a Q_a (C b e x) \right) \\ & \rightarrow \forall (a : \text{Set}) (Q_a : a \rightarrow \text{Set}) (x : G a) \rightarrow G^{\wedge} a Q_a x \rightarrow P a Q_a x \quad (17) \end{aligned}$$

Consistently with the examples in the previous sections, we define DIG as

$$\text{DIG } P \text{ pc } a Q_a (C b e x) (Q_b, L_E, L_G) = \text{pc } a b Q_a Q_b e x L_E p$$

where (Q_b, L_E, L_G) , the last term applied to DIG, has type $G^{\wedge} a Q_a (C b e x)$, i.e., $Q_b : b \rightarrow \text{Set}$, $L_E : \text{Equal}^{\wedge} a (b \times b) Q_a (\text{Pair}^{\wedge} b b Q_b Q_b) e$, and $L_G : G^{\wedge} (G b) (G^{\wedge} b Q_b) x$. We still need to define $p : P (G b) (P b Q_b) x$. We do so by using the induction rule, letting $p = \text{DIG } P \text{ pc } (G b) (P b Q_b) x q$ where we still need to provide $q : G^{\wedge} (G b) (P b Q_b) x$. To produce such q , we need the map function of G^{\wedge} , with type

$$\text{GLMap} : \forall (a : \text{Set}) (Q_a Q'_a : a \rightarrow \text{Set}) \rightarrow \text{PredMap } a Q_a Q'_a \rightarrow \text{PredMap } (G a) (G^{\wedge} a Q_a) (G^{\wedge} a Q'_a)$$

where $\text{PredMap} : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow \text{Set}$ is defined as

$$\text{PredMap } a Q_a Q'_a = \forall (x : a) \rightarrow Q_a x \rightarrow Q'_a x$$

and represents the type of morphisms between predicates. If we had GLMap, then we would be able to define

$$q = \text{GLMap } (G b) (G^{\wedge} b Q_b) (P b Q_b) (\text{DIG } P \text{ pc } b Q_b) x L_G$$

Unfortunately, we cannot define such a GLMap. Indeed, its definition would have to be

$$\text{GLMap } a Q_a Q'_a M (C b e x) (Q_b, L_E, L_G) = (Q'_b, L'_E, L'_G)$$

where (Q_b, L_E, L_G) has type $G^{\wedge} a Q_a (C b e x)$ and (Q'_b, L'_E, L'_G) has type $G^{\wedge} a Q'_a (C b e x)$, i.e.,

$$Q_a Q'_a : a \rightarrow \text{Set}$$

$$Q_b Q'_b : b \rightarrow \text{Set}$$

$$L_E : \text{Equal}^{\wedge} a (b \times b) Q_a (\text{Pair}^{\wedge} b b Q_b Q_b) e$$

$$L_G : G^{\wedge} (G b) (G^{\wedge} b Q_b) x$$

$$L'_E : \text{Equal}^{\wedge} a (b \times b) Q'_a (\text{Pair}^{\wedge} b b Q'_b Q'_b) e$$

$$L'_G : G^{\wedge} (G b) (G^{\wedge} b Q'_b) x$$

In other words, we have a proof L_E of the (extensional) equality of the predicates Q_a and $\text{Pair}^{\wedge} b b Q_b Q_b$ and a morphism of predicates M from Q_a to Q'_a , and we need to use those to deduce a proof of the (extensional) equality of the predicates Q'_a and $\text{Pair}^{\wedge} b b Q'_b Q'_b$. But that is not generally possible: the facts that Q_a is equal to $\text{Pair}^{\wedge} b b Q_b Q_b$ and that there is a morphism of predicates M from Q_a to Q'_a do not guarantee that Q'_a is equal to $\text{Pair}^{\wedge} b b Q'_b Q'_b$ for some Q'_b .

At a deeper level, the fundamental issue at play here is that the Equal type does not have functorial semantics, so that having morphisms $A \rightarrow A'$ and $B \rightarrow B'$ and a proof that A is equal to A' does not provide a proof that B is equal to B' . This is because GADTs can either have a syntax-only semantics or a functorial-completion semantics. Since we are interested in induction rules, we considered GADTs with their syntax-only semantics, which is parametric but it is not functorial. Had we considered the functorial-completion semantics, instead, we would have forfeited parametricity [?].

In both cases, thus, we cannot derive an induction rule for generic GADTs when they feature nesting.

6 APPLICATIONS

Lambda normal form example.

7 PRIMITIVE REPRESENTATION FOR GADTS

No induction with primitive representation (reference Haskell Symposium paper and [Johann and Polonsky 2019] and paper Patricia Neil Clement 2010)

8 CONCLUSION

Mention Patricia/Neil2008 paper

9 TODO

- reference (correctly) Haskell Symposium paper

REFERENCES

- R. Atkey. 2012. Relational parametricity for higher kinds. In *Computer Science Logic*. 46–61.
- R. Bird and L. Meertens. 1998. Nested datatypes. In *Mathematics of Program Construction*.
- J. Cheney and R. Hinze. 2003. *First-class phantom types*. Technical Report. Cornell University.
- P. Johann and A. Polonsky. 2019. Higher-kinded data types: Syntax and semantics. In *Logic in Computer Science*. 1–13.
- P. Johann and A. Polonsky. 2020. Deep Induction: Induction Rules for (Truly) Nested Types. In *Foundations of Software Science and Computation Structures*. 339–358.
- Y. Minsky. 2015. Why GADTs matter for performance. <https://blog.janestreet.com/why-gadts-matter-for-performance/>. (2015).
- E. Pasalic and N. Linger. 2004. Meta-programming with typed object-language representations. In *Generic Programming and Component Engineering*. 136–167.
- C. Penner. 2020. Simpler and safer API design using GADTs. <https://chrispensner.ca/posts/gadt-design>. (2020).
- S. L. Peyton Jones, D. Vytiniotis, G. Washburn, and S. Weirich. 2006. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming*. 50–61.
- F. Pottier and Y. Régis-Gianas. 2006. Stratified type inference for generalized algebraic data types. In *Principles of Programming Languages*. 232–244.
- T. Sheard and E. Pasalic. 2004. Meta-programming with built-in type equality. In *Fourth International Workshop on Logical Frameworks and Meta-Languages*.
- D. Vytiniotis and S. Weirich. 2010. Parametricity, type equality, and higher-order polymorphism. (2010).
- H. Xi, C. Chen, and G. Chen. 2003. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 224–235.