(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?)

$$\begin{array}{l} \mathsf{data\ List\ (a : Set) : Set\ where} \\ \qquad \mathsf{Nil} \quad : \mathsf{List\ a} \\ \qquad \mathsf{Cons} : \mathsf{a} \to \mathsf{List\ a} \to \mathsf{List\ a} \end{array} \tag{1}$$

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*

$$\begin{array}{l} \mathsf{data\ PTree\ (a : Set) : Set\ where} \\ \qquad \mathsf{PLeaf} \quad : \mathsf{a} \to \mathsf{PTree\ a} \\ \qquad \mathsf{PNode} : \mathsf{PTree\ (a \times a)} \to \mathsf{PTree\ a} \end{array} \tag{2}$$

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 $(\mathsf{a} \times \mathsf{a})$ 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

$$\begin{array}{l} \mathsf{data\ Bush\ (a : Set) : Set\ where} \\ \qquad \mathsf{BNil} \quad : \mathsf{Bush\ a} \\ \qquad \mathsf{BCons} : \mathsf{a} \to \mathsf{Bush\ (Bush\ a)} \to \mathsf{Bush\ a} \end{array} \tag{3}$$

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

2021. 2475-1421/2021/8-ART1 $15.00
https://doi.org/

defined. In other words, all constructors of PTree a have to return an element of type PTree a, and all constructors of Bush a have to return an element of type 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 $\mathsf{a} \to \mathsf{Set}$ (where Set is the type of sets) mapping each $\mathsf{x} : \mathsf{a}$ to the set of proofs that the predicate holds for x. For example, the induction principle for List is

$$\begin{aligned}\forall(\mathsf{a} : \mathsf{Set})(\mathsf{P} : \mathsf{List\,a} \to \mathsf{Set}) \to \mathsf{P\,Nil} \to \big(\forall(\mathsf{x} : \mathsf{a})(\mathsf{xs} : \mathsf{List\,a}) \to \mathsf{P\,xs} \to \mathsf{P\,(Cons\,x\,xs)}\big)\\ \to \forall(\mathsf{xs} : \mathsf{List\,a}) \to \mathsf{P\,xs}\end{aligned} \tag{4}$$

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 necessary involve a polymorphic predicate. Thus, the induction rule for PTree is

$$\begin{aligned}\forall(\mathsf{P} : \forall(\mathsf{a} : \mathsf{Set}) \to \mathsf{PTree\,a} \to \mathsf{Set}) \to \big(\forall(\mathsf{a} : \mathsf{Set})(\mathsf{x} : \mathsf{a}) \to \mathsf{P\,a\,(PLeaf\,x)}\big)\\ \to \big(\forall(\mathsf{a} : \mathsf{Set})(\mathsf{x} : \mathsf{PTree\,(a \times a)}) \to \mathsf{P\,(a \times a)\,x} \to \mathsf{P\,a\,(PNode\,x)}\big)\\ \to \forall(\mathsf{a} : \mathsf{Set})(\mathsf{x} : \mathsf{PTree\,a}) \to \mathsf{P\,a\,x}\end{aligned} \tag{5}$$

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

$$\begin{aligned}\forall(\mathsf{P} : \forall(\mathsf{a} : \mathsf{Set}) \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{Bush\,a} \to \mathsf{Set}) \to \big(\forall(\mathsf{a} : \mathsf{Set}) \to \mathsf{P\,a\,BNil}\big)\\ \to \big(\forall(\mathsf{a} : \mathsf{Set})(\mathsf{Q} : \mathsf{a} \to \mathsf{Set})(\mathsf{x} : \mathsf{a})(\mathsf{y} : \mathsf{Bush\,(Bush\,a)})\\ \to \mathsf{Q\,x} \to \mathsf{P\,(Bush\,a)\,(Bush^{\wedge}\,a\,Q)\,y} \to \mathsf{P\,a\,Q\,(BCons\,x\,y)}\big)\\ \to \forall(\mathsf{a} : \mathsf{Set})(\mathsf{Q} : \mathsf{a} \to \mathsf{Set})(\mathsf{x} : \mathsf{Bush\,a}) \to \mathsf{Bush^{\wedge}\,a\,Q\,x} \to \mathsf{P\,a\,Q\,x}\end{aligned} \tag{6}$$

where $\mathsf{Bush}^{\wedge} : \forall(\mathsf{a} : \mathsf{Set}) \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{Bush\,a} \to \mathsf{Set}$ lifts a predicate Q on data of type a to a predicate on data of type Bush a asserting that Q holds for every element of type a contained in its argument bush. It is defined as

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

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

$$\begin{aligned}&\forall(\mathsf{a} : \mathsf{Set})(\mathsf{P} : \mathsf{List}\ \mathsf{a} \to \mathsf{Set})(\mathsf{Q} : \mathsf{a} \to \mathsf{Set})\\ &\qquad \to \mathsf{P}\ \mathsf{Nil} \to \big(\forall(\mathsf{x} : \mathsf{a})(\mathsf{xs} : \mathsf{List}\ \mathsf{a}) \to \mathsf{Q}\ \mathsf{x} \to \mathsf{P}\ \mathsf{xs} \to \mathsf{P}\ (\mathsf{Cons}\ \mathsf{x}\ \mathsf{xs})\big)\\ &\qquad\qquad \to \forall(\mathsf{xs} : \mathsf{List}\ \mathsf{a}) \to \mathsf{List}^\wedge\ \mathsf{a}\ \mathsf{Q}\ \mathsf{xs} \to \mathsf{P}\ \mathsf{xs}\end{aligned} \tag{7}$$

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

$$\begin{aligned}&\forall(\mathsf{P} : \forall(\mathsf{a} : \mathsf{Set}) \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{PTree}\ \mathsf{a} \to \mathsf{Set})\\ &\qquad \to \big(\forall(\mathsf{a} : \mathsf{Set})(\mathsf{Q} : \mathsf{a} \to \mathsf{Set})(\mathsf{x} : \mathsf{a}) \to \mathsf{Q}\ \mathsf{x} \to \mathsf{P}\ \mathsf{a}\ \mathsf{Q}\ (\mathsf{PLeaf}\ \mathsf{x})\big)\\ &\to \big(\forall(\mathsf{a} : \mathsf{Set})(\mathsf{Q} : \mathsf{a} \to \mathsf{Set})(\mathsf{x} : \mathsf{PTree}\ (\mathsf{a} \times \mathsf{a})) \to \mathsf{P}\ (\mathsf{a} \times \mathsf{a})\ (\mathsf{Pair}^\wedge\ \mathsf{a}\ \mathsf{Q})\ \mathsf{x} \to \mathsf{P}\ \mathsf{a}\ \mathsf{Q}\ (\mathsf{PNode}\ \mathsf{x})\big)\\ &\qquad\qquad \to \forall(\mathsf{a} : \mathsf{Set})(\mathsf{Q} : \mathsf{a} \to \mathsf{Set})(\mathsf{x} : \mathsf{PTree}\ \mathsf{a}) \to \mathsf{PTree}^\wedge\ \mathsf{a}\ \mathsf{Q}\ \mathsf{x} \to \mathsf{P}\ \mathsf{a}\ \mathsf{Q}\ \mathsf{x}\end{aligned} \tag{8}$$

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

Moreover, for types admitting both deep induction and structural induction, the former generalizes the latter. Indeed, structural induction rules can be derived from deep induction rules by choosing the constantly true predicate as the custom predicate traversing each data element contained in the data type. That way, deep induction only inspects the structure of the data type and not its content, just like structural induction does. A concrete example of this technique will be demonstrated in Section 4.1.

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

$$\begin{aligned}&\mathsf{data\ Equal\ (a\ b : Set) : Set\ where}\\ &\qquad \mathsf{Refl} : \mathsf{Equal\ c\ c}\end{aligned} \tag{9}$$

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

$$\begin{aligned}&\mathsf{data\ Seq\ (a : Set) : Set\ where}\\ &\qquad \mathsf{Const} : \mathsf{a} \to \mathsf{Seq\ a}\\ &\qquad \mathsf{SPair} : \mathsf{Seq\ a} \to \mathsf{Seq\ b} \to \mathsf{Seq\ (a \times b)}\end{aligned} \tag{10}$$

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

$$\begin{aligned}&\mathsf{data\ Seq\ (a : Set) : Set\ where}\\ &\qquad \mathsf{Const} : \mathsf{a} \to \mathsf{Seq\ a}\\ &\qquad \mathsf{SPair} : \exists (\mathsf{b\ c : Set}).\, \mathsf{Equal\ a\ (b \times c)} \to \mathsf{Seq\ b} \to \mathsf{Seq\ c} \to \mathsf{Seq\ a}\end{aligned} \tag{11}$$

where the requirement that the SPair constructor produces 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 some product type. 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 Section 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 2019]. 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 some 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 GADTs not featuring nesting in their definition.

4.1 (Deep) induction for Equal

As a first example, we derive the induction rule for the Equal type from Equation 9. This will provide a simple case study that will inform the investigation of more complex GADTs. Moreover, since we define GADTs using the Equal type, as for example in Equation 10, this example will be instrumental in stating and deriving the induction rule of other GADTs.

To define an induction rule for a type G we first need a predicate-lifting operation which takes predicates on a type a and lifts them to predicates on G a. the predicate-lifting function for Equal is the function

$$\mathsf{Equal}^{\wedge} : \forall (a\, b : \mathsf{Set}) \to (a \to \mathsf{Set}) \to (b \to \mathsf{Set}) \to \mathsf{Equal}\; a\; b \to \mathsf{Set}$$

defined as

$$\mathsf{Equal}^{\wedge}\; a\; a\; Q\; Q'\; \mathsf{Refl} = \forall (x : a) \to \mathsf{Equal}\; (Q\; x)(Q'\; x)$$

i.e., the function that takes two predicates on the same type and tests them for extensional equality.

Next, we need to associate each constructor of the GADT under consideration to the expression that a given predicate is preserved by such constructor. Let CRefl be the following function associated to the Refl constructor:

$$\begin{aligned}&\lambda(P : \forall (a\, b : \mathsf{Set}) \to (a \to \mathsf{Set}) \to (b \to \mathsf{Set}) \to \mathsf{Equal}\; a\; b \to \mathsf{Set})\\ &\qquad \to \forall (c : \mathsf{Set})(Q : c \to \mathsf{Set})(Q' : c \to \mathsf{Set}) \to \mathsf{Equal}^{\wedge}\; c\; c\; Q\; Q'\; \mathsf{Refl} \to P\; c\; c\; Q\; Q'\; \mathsf{Refl}\end{aligned}$$

The induction rule states that, if a predicate is preserved by all of the constructors of the GADT under consideration, then the predicate is satisfied by any element of the GADT. The induction rule for Equal is thus the type

$$\begin{aligned}&\forall (P : \forall (a\, b : \mathsf{Set}) \to (a \to \mathsf{Set}) \to (b \to \mathsf{Set}) \to \mathsf{Equal}\; a\; b \to \mathsf{Set})\\ &\qquad \to \mathsf{CRefl}\; P \to \forall (a\, b : \mathsf{Set})(Q_a : a \to \mathsf{Set})(Q_b : b \to \mathsf{Set})(e : \mathsf{Equal}\; a\; b)\\ &\qquad\qquad \to \mathsf{Equal}^{\wedge}\; a\; b\; Q_a\; Q_b\; e \to P\; a\; b\; Q_a\; Q_b\; e\end{aligned}$$

To validate the induction rule we need to provide it with a witness, i.e., we need to show that the associated type is inhabited. We define a term DIEqual of the above type as

$$\mathsf{DIEqual}\; P\; \mathsf{crefl}\; a\; a\; Q_a\; Q_a'\; \mathsf{Refl}\; L_E = \mathsf{crefl}\; a\; Q_a\; Q_a'\; L_E$$

where $\mathsf{crefl} : \mathsf{CRefl}\; P$, $Q_a : a \to \mathsf{Set}$, $Q_a' : a \to \mathsf{Set}$ and $L_E : \mathsf{Equal}^{\wedge}\; a\; a\; Q_a\; Q_a'\; \mathsf{Refl}$. Having provided a well-defined term for it, we have shown that the induction rule for Equal is sound.

The type Equal also has a standard structural induction rule SIEqual,

$$\begin{aligned}&\forall (Q : \forall (a\, b : \mathsf{Set}) \to \mathsf{Equal}\; a\; b \to \mathsf{Set})\\ &\qquad \to \big(\forall (c : \mathsf{Set}) \to P\; c\; c\; \mathsf{Refl}\big) \to \forall (a\, b : \mathsf{Set})(e : \mathsf{Equal}\; a\; b) \to P\; a\; b\; e\end{aligned}$$

As is the case for ADTs and nested types, the structural induction rule for Equal is a consequence of the deep induction rule. Indeed, we can define SIEqual as

$$\mathsf{SIEqual}\; Q\; \mathsf{srefl}\; a\; b\; e = \mathsf{DIEqual}\; P\; \mathsf{srefl}\; a\; b\; K_1^a\; K_1^b\; e\; L_E$$

where $Q : \forall (a\, b : \mathsf{Set}) \to \mathsf{Equal}\; a\; b \to \mathsf{Set}$, $\mathsf{srefl} : \forall (c : \mathsf{Set}) \to P\; c\; c\; \mathsf{Refl}$ and $e : \mathsf{Equal}\; a\; b$, and

- $P : \forall (a\, b : \mathsf{Set}) \to (a \to \mathsf{Set}) \to (b \to \mathsf{Set}) \to \mathsf{Equal}\; a\; b \to \mathsf{Set}$ is defined as $P\; a\; b\; Q_a\; Q_b\; e = Q\; a\; b\; e$;
- K_1^a and K_1^b are the constantly 1-valued predicates on, respectively, a and b;
- $L_E : \mathsf{Equal}^{\wedge}\; a\; b\; K_1^a\; K_1^b\; e$ is defined by pattern matching, i.e., in case $a = b$ and $e = \mathsf{Refl}$, it is defined as $L_E\; x = \mathsf{Refl} : \mathsf{Equal}\; a\; a$.

That the structural induction rule is a consequence of the deep induction one is also true for all the examples below, even though we will not remark it every time.

4.2 (Deep) induction for Seq

Next, we shall provide an induction rule for the Seq type defined in Equation 11. Again, the first step in deriving the induction rule for Seq consists in defining the predicate-lifting function over it,

$$\mathsf{Seq}^\wedge : \forall(a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{Seq}\, a \to \mathsf{Set}$$

which is given by pattern-matching as

$$\mathsf{Seq}^\wedge\, a\, Q_a\, (\mathsf{Const}\, x) = Q_a\, x$$

where $Q_a : a \to \mathsf{Set}$ and $x : a$, and

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

where $e : \mathsf{Equal}\, a\, (b \times c)$, $s_b : \mathsf{Seq}\, b$ and $s_c : \mathsf{Seq}\, c$. We also need to define the lifting of predicates over the polymorphic type of pairs, $\mathsf{Pair} = \forall(b\, c : \mathsf{Set}) \to b \times c$, which is

$$\mathsf{Pair}^\wedge : \forall(b\, c : \mathsf{Set}) \to (b \to \mathsf{Set}) \to (c \to \mathsf{Set}) \to \mathsf{Pair}\, b\, c \to \mathsf{Set}$$

and it is defined as

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

where $Q_b : b \to \mathsf{Set}$, $Q_c : c \to \mathsf{Set}$, $y : b$ and $z : c$.

Finally, let CConst be the function

$$\begin{aligned}&\lambda(P : \forall(a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{Seq}\, a \to \mathsf{Set})\\ &\qquad \to \forall(a : \mathsf{Set})(Q_a : a \to \mathsf{Set})(x : a) \to Q_a\, x \to P\, a\, Q_a\, (\mathsf{Const}\, x)\end{aligned}$$

associated to the Const constructor, and let CSPair be the function

$$\begin{aligned}&\lambda(P : \forall(a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{Seq}\, a \to \mathsf{Set})\\ &\qquad \to \forall(a\, b\, c : \mathsf{Set})(Q_a : a \to \mathsf{Set})(Q_b : b \to \mathsf{Set})(Q_c : c \to \mathsf{Set})\\ &\qquad (s_b : \mathsf{Seq}\, b)(s_c : \mathsf{Seq}\, c)(e : \mathsf{Equal}\, a\, (b \times c)) \to \mathsf{Equal}^\wedge a\, (b \times c)\, Q_a\, (\mathsf{Pair}^\wedge\, b\, c\, Q_b\, Q_c)\, e\\ &\qquad \to P\, b\, Q_b\, s_b \to P\, c\, Q_c\, s_c \to P a Q_a (\mathsf{SPair}\, b\, c\, e\, s_b\, s_c)\end{aligned}$$

associated to the SPair constructor,

With these tools we can formulate an induction rule for Seq,

$$\begin{aligned}&\forall(P : \forall(a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{Seq}\, a \to \mathsf{Set}) \to \mathsf{CConst}\, P \to \mathsf{CSPair}\, P\\ &\qquad \to \forall(a : \mathsf{Set})(Q_a : a \to \mathsf{Set})(s_a : \mathsf{Seq}\, a) \to \mathsf{Seq}^\wedge\, a\, Q_a\, s_a \to P\, a\, Q_a\, s_a\end{aligned}$$

To validate the induction rule, we define a term DISeq for the above type. We have to define

$$\mathsf{DISeq}\, P\, \mathsf{cconst}\, \mathsf{cspair}\, a\, Q_a\, s_a\, L_a : P\, a\, Q_a\, s_a$$

where $P : \forall(a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{Seq}\, a \to \mathsf{Set}$, $\mathsf{cconst} : \mathsf{CConst}\, P$, $\mathsf{cspair} : \mathsf{CSPair}\, P$, $Q_a : a \to \mathsf{Set}$, $s_a : \mathsf{Seq}\, a$ and $L_a : \mathsf{Seq}^\wedge\, a\, Q\, s_a$, and we proceed by pattern-matching on s_a. Let $s_a = \mathsf{Const}\, x$ for $x : a$, and define

$$\mathsf{DISeq}\, P\, \mathsf{cconst}\, \mathsf{cspair}\, a\, Q_a\, (\mathsf{Const}\, x)\, L_a = \mathsf{cconst}\, a\, Q_a\, x\, L_a$$

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

$$\mathsf{DISeq}\, P\, \mathsf{cconst}\, \mathsf{cspair}\, a\, Q_a\, (\mathsf{SPair}\, b\, c\, e\, s_b\, s_c)\, (Q_b, Q_c, L_e, L_b, L_c) = \mathsf{cspair}\, a\, b\, c\, Q_a\, Q_b\, Q_c\, s_b\, s_c\, e\, L_e\, p_b\, p_c$$

where $(Q_b, Q_c, L_e, L_b, L_c) : \mathsf{Seq}^\wedge\, a\, Q\, (\mathsf{SPair}\, b\, c\, e\, x\, y)$, i.e.,

- $Q_b : b \to \mathrm{Set}$ and $Q_c : c \to \mathrm{Set}$;
- $L_e : \mathrm{Equal}^\wedge\ a\ (b \times c)\ Q_a\ (Q_b \times Q_c)\ e$;
- $L_b : \mathrm{Seq}^\wedge\ b\ Q_b\ s_b$ and $L_c : \mathrm{Seq}^\wedge\ c\ Q_c\ s_c$;

and p_b and p_c are defined as follows:

$$p_b = \mathrm{DISeq}\ P\ \mathrm{cconst}\ \mathrm{cspair}\ b\ Q_b\ s_b\ L_b : P\ b\ Q_b\ s_b$$
$$p_c = \mathrm{DISeq}\ P\ \mathrm{cconst}\ \mathrm{cspair}\ c\ Q_c\ s_c\ L_c : P\ c\ Q_c\ s_c$$

4.3 Third example introduced above

4.4 General case

Finally, we generalize the approach taken in the previous examples and provide a general framework to derive induction rules for arbitrary GADTs. For that, we need to give a grammar for the types we will be considering. A generic GADT

$$\mathrm{data}\ G\ (\overline{a : \mathrm{Set}}) : \mathrm{Set}\ \mathrm{where}$$
$$C_i\ :\ F_i\ G\ \overline{b} \to G(\overline{K_i\ \overline{b}})$$

is defined by a finite number of constructors C_i. In the definition above, F_i is a type constructor with signature $(\mathrm{Set}^\alpha \to \mathrm{Set}) \to \mathrm{Set}^\beta \to \mathrm{Set}$ and each K_i is a type constructor with signature $\mathrm{Set}^\beta \to \mathrm{Set}$ (i.e. a type constructor of arity β). The overline notation denotes a finite list: $\overline{a}$ is a list of types of length α, so that it can be applied to the type constructor G of arity α. Each of the α-many K_i is a type constructor of arity β so that it can be applied to the list of types $\overline{b}$ of length β. Moreover, notice that the arity of G matches the number of type constructors $\overline{K_i}$. We allow each F_i to be inductively built in the following ways (and with the following restrictions):

- $F_i = F_i' \times F_i''$ where F_i' and F_i'' have the same signature as F_i and are built recursively from the same induction rules.
- $F_i = F_i' + F_i''$ where F_i' and F_i'' have the same signature as F_i and are built recursively from the same induction rules.
- $F_i = F_i' \to F_i''$ where F_i' does not contain the recursive variable, i.e., $F_i' : \mathrm{Set}^\beta \to \mathrm{Set}$ is a type constructor of arity β, and F_i'' has the same signature as F_i and is built recursively from the same induction rules.
- $F_i\ G\ \overline{b} = G(\overline{F_a\ \overline{b}})$ where none of the F_a contains the recursive variable, i.e., $F_a : \mathrm{Set}^\beta \to \mathrm{Set}$ is a type constructor of arity β for each a. Such restriction is necessary to prevent nesting, as that would break the induction rule as discussed in Section 5.
- $F_i\ G\ \overline{b} = H\ \overline{b}$ where H is a type constructor of arity β not containing the recursive variable, i.e., $H : \mathrm{Set}^\beta \to \mathrm{Set}$. Notice that this covers the case in which F_i is a closed type, so, in particular, the unit and empty types, 1 and 0.
- $F_i\ G\ \overline{b} = H(\overline{F_c\ G\ \overline{b}})$ where H is a γ-ary type constructor not containing the recursive variable, i.e., $H : \mathrm{Set}^\gamma \to \mathrm{Set}$, and F_c has the same signature as F_i and is built recursively from the same induction rules, for every $c = 1 \ldots \gamma$. Moreover, we require that H is not a GADT itself (but we allow it to be an ADT or even a nested type). This way H admits functorial semantics, and thus we have the map function of $H^\wedge$,

$$\mathrm{HLMap} : \forall(\overline{c : \mathrm{Set}})(\overline{Q_c\ Q_c' : c \to \mathrm{Set}}) \to \overline{\mathrm{PredMap}\ c\ Q_c\ Q_c'} \to \mathrm{PredMap}\ (H\ \overline{c})\ (H^\wedge\ \overline{c}\ \overline{Q_c})\ (H^\wedge\ \overline{c}\ \overline{Q_c'})$$

where $\mathrm{PredMap} : \forall(c : \mathrm{Set}) \to (c \to \mathrm{Set}) \to (c \to \mathrm{Set}) \to \mathrm{Set}$ is defined as

$$\mathrm{PredMap}\ c\ Q_c\ Q_c' = \forall(x : c) \to Q_c\ x \to Q_c'\ x$$

and represents the type of morphisms between predicates.

We can summarize the above inductive definition with the following grammar (but beware that the above restrictions and requirements still apply):

$$\mathsf{F_i\,G\,\overline{b} := F'_i\,G\,\overline{b} \times F''_i\,G\,\overline{b} \mid F'_i\,G\,\overline{b} + F''_i\,G\,\overline{b} \mid F'_i\,\overline{b} \to F''_i\,G\,\overline{b} \mid G(\overline{F_a\,\overline{b}}) \mid H\,\overline{b} \mid H(\overline{F_c\,G\,\overline{b}})}$$

A further requirement that applies to all of the types appearing above, including the types $\mathsf{K_i}$, is that every type needs to have a predicate-lifting function. This is not an overly restrictive condition, though: all types made by sums, products, arrow types and type application do, and so do GADTs as defined above. A way to concretely define the predicate-lifting function for a type is to proceed by induction on the structure of the type, and we have seen in the previous sections examples of how to do so for products and type application. We do not give here the general definition of lifting, as that would require to first present a full type calculus, and that is beyond the scope of the paper.

Consider a generic GADT as defined above,

$$\begin{array}{c}\mathsf{data\ G\ (a : Set) : Set\ where} \\ \mathsf{C\ :\ F\,G\,\overline{b} \to G(K\,\overline{b})}\end{array} \tag{12}$$

which, for ease of notation, we assume to be a unary type constructor (i.e. it depends on a single type parameter a) and to have only one constructor C. Extending the argument to GADTs of arbitrary arity and with multiple constructors presents no difficulty other than heavier notation. In the definition above, F has signature $\mathsf{(Set \to Set) \to Set^{\beta} \to Set}$ and each K has signature $\mathsf{Set^{\beta} \to Set}$. The constructor C can be rewritten using the Equal type as

$$\mathsf{C : \exists(\overline{b : Set}) \to Equal\,a\,(K\,\overline{b}) \to F\,G\,\overline{b} \to G\,a}$$

which is the form we shall use from now on.

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

$$\mathsf{G^{\wedge} : \forall(a : Set) \to (a \to Set) \to G\,a \to Set}$$

as

$$\mathsf{G^{\wedge}\,a\,Q_a\,(C\,\overline{b}\,e\,x) = \exists(\overline{Q_b : b \to Set}) \to Equal^{\wedge}\,a\,(K\,\overline{b})\,Q_a\,(K^{\wedge}\,\overline{b}\,\overline{Q_b})\,e \times F^{\wedge}\,G\,\overline{b}\,G^{\wedge}\,\overline{Q_b}\,x}$$

where $\mathsf{Q_a : a \to Set}$ and $\mathsf{C\,\overline{b}\,e\,x : G\,a}$, i.e., $\mathsf{e : Equal\,a\,(K\,\overline{b})}$ and $\mathsf{x : F\,G\,\overline{b}}$. As already mentioned before, we also assume to have liftings for F,

$$\begin{aligned}\mathsf{F^{\wedge}} &: \mathsf{\forall(G : Set^{\alpha} \to Set)(\overline{b : Set}) \to (\forall(a : Set) \to (a \to Set) \to G\,a \to Set)} \\ &\qquad \mathsf{\to (\overline{b \to Set}) \to F\,G\,\overline{b} \to Set}\end{aligned}$$

and for K,

$$\mathsf{K^{\wedge} : \forall(\overline{b : Set}) \to (\overline{b \to Set}) \to K\,\overline{b} \to Set}$$

Finally, associate the function

$$\begin{aligned}\mathsf{CC} = \lambda(&\mathsf{P : \forall(a : Set) \to (a \to Set) \to G\,a \to Set)} \\ &\mathsf{\to \forall(a : Set)(\overline{b : Set})(Q_a : a \to Set)(\overline{Q_b : b \to Set})(e : Equal\,a\,(K\,\overline{b}))(x : F\,G\,\overline{b})} \\ &\qquad \mathsf{\to Equal^{\wedge}\,a\,(K\,\overline{b})\,Q_a\,(K^{\wedge}\,\overline{b}\,\overline{Q_b})\,e \to F^{\wedge}\,G\,\overline{b}\,P\,\overline{Q_b}\,x \to P\,a\,Q_a\,(C\,\overline{b}\,e\,x)}\end{aligned}$$

to the constructor C.

The induction rule for G is

$$\begin{aligned}&\mathsf{\forall(P : \forall(a : Set) \to (a \to Set) \to G\,a \to Set) \to CC\,P} \\ &\qquad \mathsf{\to \forall(a : Set)(Q_a : a \to Set)(y : G\,a) \to G^{\wedge}\,a\,Q_a\,y \to P\,a\,Q_a\,y}\end{aligned}$$

As we already did in the previous examples, we validate the induction rule by providing a term DIG for the type above. Define

$$\mathsf{DIG}\;\mathsf{P}\;\mathsf{cc}\;\mathsf{a}\;\mathsf{Q_a}\;(\mathsf{C}\;\overline{\mathsf{b}}\;\mathsf{e}\;\mathsf{x})\;(\overline{\mathsf{Q_b}},\mathsf{L_E},\mathsf{L_F}) = \mathsf{cc}\;\mathsf{a}\;\overline{\mathsf{b}}\;\mathsf{Q_a}\;\overline{\mathsf{Q_b}}\;\mathsf{e}\;\mathsf{x}\;\mathsf{L_E}\;(\mathsf{p}\;\mathsf{x}\;\mathsf{L_F})$$

where $\mathsf{cc} : \mathsf{CC}\;\mathsf{P}$ and

- $\mathsf{C}\;\overline{\mathsf{b}}\;\mathsf{e}\;\mathsf{x} : \mathsf{G}\;\mathsf{a}$, i.e., $\mathsf{e} : \mathsf{Equal}\;\mathsf{a}\;(\mathsf{K}\;\overline{\mathsf{b}})$, and $\mathsf{x} : \mathsf{F}\;\mathsf{G}\;\overline{\mathsf{b}}$;
- $(\overline{\mathsf{Q_b}},\mathsf{L_E},\mathsf{L_F}) : \mathsf{G}^\wedge\;\mathsf{a}\;\mathsf{Q_a}(\mathsf{C}\;\overline{\mathsf{b}}\;\mathsf{e}\;\mathsf{x})$, i.e., $\mathsf{Q_b} : \mathsf{b} \to \mathsf{Set}$ for each b, $\mathsf{L_E} : \mathsf{Equal}^\wedge\;\mathsf{a}\;(\mathsf{K}\;\overline{\mathsf{b}})\;\mathsf{Q_a}\;(\mathsf{K}^\wedge\;\overline{\mathsf{b}}\;\overline{\mathsf{Q_b}})\;\mathsf{e}$, and $\mathsf{L_F} : \mathsf{F}^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{G}^\wedge\;\overline{\mathsf{Q_b}}\;\mathsf{x}$.

Finally, the morphism of predicates

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

is defined by structural induction on F as follows:

- Case $\mathsf{F} = \mathsf{F_1} \times \mathsf{F_2}$ where $\mathsf{F_1}$ and $\mathsf{F_2}$ have the same signature as F. We have that
$$\mathsf{F}^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{P}\;\overline{\mathsf{Q_b}} = \mathsf{Pair}^\wedge(\mathsf{F_1}\;\mathsf{G}\;\overline{\mathsf{b}})(\mathsf{F_2}\;\mathsf{G}\;\overline{\mathsf{b}})(\mathsf{F}_1^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{P}\;\overline{\mathsf{Q_b}})(\mathsf{F}_2^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{P}\;\overline{\mathsf{Q_b}})$$
By inductive hypothesis, there exist morphisms of predicates
$$\mathsf{p_1} : \mathsf{PredMap}\;(\mathsf{F_1}\;\mathsf{G}\;\overline{\mathsf{b}})\;((\mathsf{F_1})^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{G}^\wedge\;\overline{\mathsf{Q_b}})((\mathsf{F_1})^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{P}\;\overline{\mathsf{Q_b}})$$
$$\mathsf{p_2} : \mathsf{PredMap}\;(\mathsf{F_2}\;\mathsf{G}\;\overline{\mathsf{b}})\;((\mathsf{F_2})^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{G}^\wedge\;\overline{\mathsf{Q_b}})((\mathsf{F_2})^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{P}\;\overline{\mathsf{Q_b}})$$
Thus, we define $\mathsf{p}(\mathsf{x_1},\mathsf{x_2})(\mathsf{L_1},\mathsf{L_2}) = (\mathsf{p_1}\;\mathsf{x_1}\;\mathsf{L_1},\mathsf{p_2}\;\mathsf{x_2}\;\mathsf{L_2})$ for $\mathsf{x_1} : \mathsf{F_1}\;\mathsf{G}\;\overline{\mathsf{b}}$, $\mathsf{L_1} : \mathsf{F}_1^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{G}^\wedge\;\overline{\mathsf{Q_b}}\;\mathsf{x_1}$, $\mathsf{x_2} : \mathsf{F_2}\;\mathsf{G}\;\overline{\mathsf{b}}$ and $\mathsf{L_2} : \mathsf{F}_2^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{G}^\wedge\;\overline{\mathsf{Q_b}}\;\mathsf{x_2}$.
- Case $\mathsf{F} = \mathsf{F_1} + \mathsf{F_2}$ where $\mathsf{F_1}$ and $\mathsf{F_2}$ have the same signature as F. Analogous to case $\mathsf{F} = \mathsf{F_1} \times \mathsf{F_2}$.
- Case $\mathsf{F} = \mathsf{F_1} \to \mathsf{F_2}$ where $\mathsf{F_1}$ does not contain the recursive variable, i.e., $\mathsf{F_1} : \mathsf{Set}^\beta \to \mathsf{Set}$, and $\mathsf{F_2}$ has the same signature as F. We have that
$$\mathsf{F}^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{P}\;\overline{\mathsf{Q_b}}\;\mathsf{x} = \forall(\mathsf{z} : \mathsf{F_1}\;\overline{\mathsf{b}}) \to \mathsf{F}_1^\wedge\;\overline{\mathsf{b}}\;\overline{\mathsf{Q_b}}\;\mathsf{z} \to \mathsf{F}_2^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{P}\;\overline{\mathsf{Q_b}}\;(\mathsf{x}\;\mathsf{z})$$
where $\mathsf{x} : \mathsf{F}\;\mathsf{G}\;\overline{\mathsf{b}} = \mathsf{F_1}\;\overline{\mathsf{b}} \to \mathsf{F_2}\;\mathsf{G}\;\overline{\mathsf{b}}$. By inductive hypothesis, there exist a morphism of predicates
$$\mathsf{p_2} : \mathsf{PredMap}\;(\mathsf{F_2}\;\mathsf{G}\;\overline{\mathsf{b}})(\mathsf{F}_2^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{G}^\wedge\;\overline{\mathsf{Q_b}})(\mathsf{F}_2^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{P}\;\overline{\mathsf{Q_b}})$$
Thus, we define $\mathsf{p}\;\mathsf{x}\;\mathsf{L_F} : \mathsf{F}^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{P}\;\overline{\mathsf{Q_b}}\;\mathsf{x}$ for $\mathsf{L_F} : \mathsf{F}^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{G}^\wedge\;\overline{\mathsf{Q_b}}\;\mathsf{x}$ as $\mathsf{p}\;\mathsf{x}\;\mathsf{L_F}\;\mathsf{z}\;\mathsf{L_1} = \mathsf{p_2}(\mathsf{x}\;\mathsf{z})(\mathsf{L_F}\;\mathsf{z}\;\mathsf{L_1})$ for $\mathsf{z} : \mathsf{F_1}\;\overline{\mathsf{b}}$ and $\mathsf{L_1} : \mathsf{F}_1^\wedge\;\overline{\mathsf{b}}\;\overline{\mathsf{Q_b}}\;\mathsf{z}$. Notice that $\mathsf{F_1}$ not containing the recursive variable is a necessary restriction, as the proof relies on $\mathsf{F}^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{G}^\wedge\;\overline{\mathsf{Q_b}}\;\mathsf{x}$ and $\mathsf{F}^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{P}\;\overline{\mathsf{Q_b}}\;\mathsf{x}$ having the same domain $\mathsf{F}_1^\wedge\;\overline{\mathsf{b}}\;\overline{\mathsf{Q_b}}\;\mathsf{z}$.
- Case $\mathsf{F}\;\mathsf{G}\;\overline{\mathsf{b}} = \mathsf{G}(\mathsf{F'}\;\overline{\mathsf{b}})$ where $\mathsf{F'}$ does not contain the recursive variable, i.e., $\mathsf{F'} : \mathsf{Set}^\beta \to \mathsf{Set}$. Thus, $\mathsf{F}^\wedge\;\mathsf{G}\;\overline{\mathsf{b}}\;\mathsf{P}\;\overline{\mathsf{Q_b}} = \mathsf{P}(\mathsf{F'}\;\overline{\mathsf{b}})(\mathsf{F'}^\wedge\;\overline{\mathsf{b}}\;\overline{\mathsf{Q_b}})$. So, p is defined as
$$\mathsf{p} = \mathsf{DIG}\;\mathsf{P}\;\mathsf{cc}\;(\mathsf{F'}\;\overline{\mathsf{b}})\;(\mathsf{F'}^\wedge\;\overline{\mathsf{b}}\;\overline{\mathsf{Q_b}})$$
- Case $\mathsf{F}\;\mathsf{G}\;\overline{\mathsf{b}} = \mathsf{H}\;\overline{\mathsf{b}}$ where H is a β-ary type constructor not containing the recursive variable, i.e., $\mathsf{H} : \mathsf{Set}^\beta \to \mathsf{Set}$. In such case, $\mathsf{p} : \mathsf{PredMap}(\mathsf{H}\;\overline{\mathsf{b}})(\mathsf{H}^\wedge\;\overline{\mathsf{b}}\;\overline{\mathsf{Q_b}})(\mathsf{H}^\wedge\;\overline{\mathsf{b}}\;\overline{\mathsf{Q_b}})$ is just the identity morphism of predicates.
- Case $\mathsf{F}\;\mathsf{G}\;\overline{\mathsf{b}} = \mathsf{H}(\overline{\mathsf{F_c}\;\mathsf{G}\;\overline{\mathsf{b}}})$ where H is a γ-ary type constructor not containing the recursive variable, i.e., $\mathsf{H} : \mathsf{Set}^\gamma \to \mathsf{Set}$, and $\mathsf{F_c}$ has the same signature as F, for every $\mathsf{c} = 1 \ldots \gamma$. Moreover, we assume that H has an associated predicate-lifting function,
$$\mathsf{H}^\wedge : \forall(\overline{\mathsf{c} : \mathsf{Set}}) \to (\overline{\mathsf{c} \to \mathsf{Set}}) \to \mathsf{H}\;\overline{\mathsf{c}} \to \mathsf{Set}$$

and that this predicate-lifting function has a map function HLMap of type

$$\forall(\overline{c : \mathsf{Set}})(\overline{Q_c\ Q'_c : c \to \mathsf{Set}}) \to \mathsf{PredMap}\,\overline{c}\,\overline{Q_c}\,\overline{Q'_c} \to \mathsf{PredMap}\,(H\,\overline{c})\,(H^\wedge\,\overline{c}\,\overline{Q_c})\,(H^\wedge\,\overline{c}\,\overline{Q'_c})$$

That means that H cannot be a GADT, as GADTs have no functorial semantics [?] and incur in the issue exposed in Section 5, but it can be an ADT or even a nested type as those types have functorial semantics [Johann et al. 2021; Johann and Polonsky 2019]. Thus,

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

By induction hypothesis, there is a morphism of predicates

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

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

$$p = \mathsf{HLMap}\,(\overline{F_c\,G\,\overline{b}})(\overline{F_c^\wedge\,G\,\overline{b}\,G^\wedge\,\overline{Q_b}})(\overline{F_c^\wedge\,G\,\overline{b}\,P\,\overline{Q_b}})\,\overline{p_c}$$

5 INDUCTION FOR GADTS WITH NESTING

In the previous sections, we derive induction rules for examples of 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 show how induction for GADTs featuring nesting goes wrong by analyzing the following concrete example of such a type.

$$\begin{array}{l} \mathsf{data}\ G\ (a : \mathsf{Set}) : \mathsf{Set}\ \mathsf{where} \\ \quad C :\ G(G\,a) \to G(a \times a) \end{array} \tag{13}$$

The constructor C can be rewritten as

$$C : \exists(b : \mathsf{Set}) \to \mathsf{Equal}\,a\,(b \times b) \to G(G\,b) \to G\,a$$

which is the form we shall use from now on. The predicate-lifting function of G,

$$G^\wedge : \forall(\overline{a : \mathsf{Set}}) \to (\overline{a \to \mathsf{Set}}) \to G\,\overline{a} \to \mathsf{Set}$$

is defined as

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

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

Finally, let CC be the type

$$\begin{array}{l} \forall(P : \forall(a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to G\,a \to \mathsf{Set}) \\ \qquad \to \forall(a\ b : \mathsf{Set})(Q_a : a \to \mathsf{Set})(Q_b : b \to \mathsf{Set})(e : \mathsf{Equal}\,a\,(b \times b))(x : G\,(G\,b)) \\ \qquad\quad \to \mathsf{Equal}^\wedge\,a\,(b \times b)\,Q_a\,(\mathsf{Pair}^\wedge\,b\,b\,Q_b\,Q_b)\,e \to P\,(G\,b)\,(P\,b\,Q_b)\,x \to P\,a\,Q_a\,(C\,b\,e\,x) \end{array}$$

associated to the C constructor.

The induction rule for G is

$$\begin{array}{l} \forall(P : \forall(a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to G\,a \to \mathsf{Set}) \to CC\,P \\ \qquad\qquad \to \forall(a : \mathsf{Set})(Q_a : a \to \mathsf{Set})(y : G\,a) \to G^\wedge\,a\,Q_a\,y \to P\,a\,Q_a\,y \end{array}$$

Consistently with the previous examples, we define a term validating the induction rule, DIG, as

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

where $\mathsf{cc} : \mathsf{CC}\;\mathsf{P}$, $\mathsf{C}\;\mathsf{b}\;\mathsf{e}\;\mathsf{x} : \mathsf{G}\;\mathsf{a}$, i.e., $\mathsf{e} : \mathsf{Equal}\;\mathsf{a}\;(\mathsf{b}\times\mathsf{b})$ and $\mathsf{x} : \mathsf{G}(\mathsf{G}\;\mathsf{b})$, and $(\mathsf{Q_b},\mathsf{L_E},\mathsf{L_G}) : \mathsf{G}^\wedge\;\mathsf{a}\;\mathsf{Q_a}\;(\mathsf{C}\;\mathsf{b}\;\mathsf{e}\;\mathsf{x})$, i.e., $\mathsf{Q_b} : \mathsf{b}\to\mathsf{Set}$, $\mathsf{L_E} : \mathsf{Equal}^\wedge\;\mathsf{a}\;(\mathsf{b}\times\mathsf{b})\;\mathsf{Q_a}\;(\mathsf{Pair}^\wedge\;\mathsf{b}\;\mathsf{b}\;\mathsf{Q_b}\;\mathsf{Q_b})\;\mathsf{e}$, and $\mathsf{L_G} : \mathsf{G}^\wedge\;(\mathsf{G}\;\mathsf{b})\;(\mathsf{G}^\wedge\;\mathsf{b}\;\mathsf{Q_b})\;\mathsf{x}$. We still need to define $\mathsf{p} : \mathsf{P}\;(\mathsf{G}\;\mathsf{b})\;(\mathsf{P}\;\mathsf{b}\;\mathsf{Q_b})\;\mathsf{x}$. We do so by using the induction rule and letting

$$\mathsf{p} = \mathsf{DIG}\;\mathsf{P}\;\mathsf{cc}\;(\mathsf{G}\;\mathsf{b})\;(\mathsf{P}\;\mathsf{b}\;\mathsf{Q_b})\;\mathsf{x}\;\mathsf{q}$$

where we still need to provide $\mathsf{q} : \mathsf{G}^\wedge\;(\mathsf{G}\;\mathsf{b})\;(\mathsf{P}\;\mathsf{b}\;\mathsf{Q_b})\;\mathsf{x}$. To produce such q, we need the map function of $\mathsf{G}^\wedge$,

$$\mathsf{GLMap} : \forall(\mathsf{a}:\mathsf{Set})(\mathsf{Q_a}\;\mathsf{Q_a'} : \mathsf{a}\to\mathsf{Set}) \to \mathsf{PredMap}\;\mathsf{a}\;\mathsf{Q_a}\;\mathsf{Q_a'} \to \mathsf{PredMap}\;(\mathsf{G}\;\mathsf{a})\;(\mathsf{G}^\wedge\;\mathsf{a}\;\mathsf{Q_a})\;(\mathsf{G}^\wedge\;\mathsf{a}\;\mathsf{Q_a'})$$

where $\mathsf{PredMap} : \forall(\mathsf{a}:\mathsf{Set}) \to (\mathsf{a}\to\mathsf{Set}) \to (\mathsf{a}\to\mathsf{Set}) \to \mathsf{Set}$ is defined as

$$\mathsf{PredMap}\;\mathsf{a}\;\mathsf{Q_a}\;\mathsf{Q_a'} = \forall(\mathsf{x}:\mathsf{a}) \to \mathsf{Q_a}\;\mathsf{x} \to \mathsf{Q_a'}\;\mathsf{x}$$

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

$$\mathsf{q} = \mathsf{GLMap}\;(\mathsf{G}\;\mathsf{b})\;(\mathsf{G}^\wedge\;\mathsf{b}\;\mathsf{Q_b})\;(\mathsf{P}\;\mathsf{b}\;\mathsf{Q_b})\;(\mathsf{DIG}\;\mathsf{P}\;\mathsf{cc}\;\mathsf{b}\;\mathsf{Q_b})\;\mathsf{x}\;\mathsf{L_G}$$

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

$$\mathsf{GLMap}\;\mathsf{a}\;\mathsf{Q_a}\;\mathsf{Q_a'}\;\mathsf{M}\;(\mathsf{C}\;\mathsf{b}\;\mathsf{e}\;\mathsf{x})\;(\mathsf{Q_b},\mathsf{L_E},\mathsf{L_G}) = (\mathsf{Q_b'},\mathsf{L_E'},\mathsf{L_G'})$$

where $\mathsf{Q_a} : \mathsf{a}\to\mathsf{Set}$, $\mathsf{Q_a'} : \mathsf{a}\to\mathsf{Set}$, $\mathsf{M} : \mathsf{PredMap}\;\mathsf{a}\;\mathsf{Q_a}\;\mathsf{Q_a'}$, $\mathsf{C}\;\mathsf{b}\;\mathsf{e}\;\mathsf{x} : \mathsf{G}\;\mathsf{a}$, i.e.,

- $\mathsf{e} : \mathsf{Equal}\;\mathsf{a}\;(\mathsf{b}\times\mathsf{b})$;
- $\mathsf{x} : \mathsf{G}\;(\mathsf{G}\;\mathsf{b})$;

$(\mathsf{Q_b},\mathsf{L_E},\mathsf{L_G})$ has type $\mathsf{G}^\wedge\;\mathsf{a}\;\mathsf{Q_a}\;(\mathsf{C}\;\mathsf{b}\;\mathsf{e}\;\mathsf{x})$, i.e.,

- $\mathsf{Q_b} : \mathsf{b}\to\mathsf{Set}$;
- $\mathsf{L_E} : \mathsf{Equal}^\wedge\;\mathsf{a}\;(\mathsf{b}\times\mathsf{b})\;\mathsf{Q_a}\;(\mathsf{Pair}^\wedge\;\mathsf{b}\;\mathsf{b}\;\mathsf{Q_b}\;\mathsf{Q_b})\;\mathsf{e}$;
- $\mathsf{L_G} : \mathsf{G}^\wedge\;(\mathsf{G}\;\mathsf{b})\;(\mathsf{G}^\wedge\;\mathsf{b}\;\mathsf{Q_b})\;\mathsf{x}$;

and $(\mathsf{Q_b'},\mathsf{L_E'},\mathsf{L_G'})$ has type $\mathsf{G}^\wedge\;\mathsf{a}\;\mathsf{Q_a'}\;(\mathsf{C}\;\mathsf{b}\;\mathsf{e}\;\mathsf{x})$, i.e.,

- $\mathsf{Q_b'} : \mathsf{b}\to\mathsf{Set}$;
- $\mathsf{L_E'} : \mathsf{Equal}^\wedge\;\mathsf{a}\;(\mathsf{b}\times\mathsf{b})\;\mathsf{Q_a'}\;(\mathsf{Pair}^\wedge\;\mathsf{b}\;\mathsf{b}\;\mathsf{Q_b'}\;\mathsf{Q_b'})\;\mathsf{e}$;
- $\mathsf{L_G'} : \mathsf{G}^\wedge\;(\mathsf{G}\;\mathsf{b})\;(\mathsf{G}^\wedge\;\mathsf{b}\;\mathsf{Q_b'})\;\mathsf{x}$;

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

At a deeper level, the fundamental issue is that the Equal type does not have functorial semantics, so that having morphisms $A\to A'$ and $B\to B'$ 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, E. Ghiorzi, and D. Jeffries. 2021. Parametricity for Primitive Nested Types. In *Foundations of Software Science and Computation Structures*. 324–343.

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://chrispenner.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.