

# (Deep) Induction Rules for GADTs

Patricia Johann   Enrico Ghiorzi   Daniel Jeffries

{johannp, ghiorzie, jeffriesd}@appstate.edu  
Appalachian State University

---

## Abstract

Deep data types are those that are defined in terms of other such data types, including, possibly, themselves. In that case, they are said to be truly nested. Deep induction is an extension of structural induction that traverses *all* of the structure in a deep data type, propagating predicates on its primitive data throughout the entire structure. Deep induction can be used to prove properties of nested types, including truly nested types, that cannot be proved via structural induction. In this paper we show how to extend deep induction to deep GADTs that are not truly nested. We also show that deep induction cannot be extended to truly nested GADTs.

---

## 1 Introduction

Induction is one of the most important techniques available for working with advanced data types, so it is both inevitable and unsurprising that it plays an essential role in modern proof assistants. In the proof assistant Coq [7], for example, functions and predicates over advanced types are defined inductively, and almost all non-trivial proofs of their properties are either proved by induction outright or rely on lemmas that are. Every time a new inductive type is declared in Coq an induction rule is automatically generated for it.

The inductive data types handled by Coq include (possibly mutually inductive) polynomial algebraic data types (ADTs), and the induction rules Coq generates for them are the expected ones for standard structural induction. It has long been understood, however, that these rules are too weak to be genuinely useful for deep ADTs [15], i.e., ADTs that are (possibly mutually inductively) defined in terms of (other) such ADTs.<sup>1</sup> The following data type of rose trees, here coded in Agda and defined in terms of the standard type List of lists (see Section 2), is a deep ADT:

```
data Rose : Set → Set where
  empty : ∀{A : Set} → Rose A
  node  : ∀{A : Set} → A → List (Rose A) → Rose A
```

The induction rule Coq automatically generates for (the analogous Coq definition of) rose trees is

$$\begin{aligned} &\forall (A : \text{Set}) (P : \text{Rose } A \rightarrow \text{Set}) \rightarrow P \text{ empty} \rightarrow \\ &(\forall (a : A) (ts : \text{List } (\text{Rose } A)) \rightarrow P (\text{node } a \text{ ts})) \rightarrow \forall (x : \text{Rose } A) \rightarrow P x \end{aligned}$$

Unfortunately, this is neither the induction rule we intuitively expect, nor is it expressive enough to prove even basic properties of rose trees that ought to be amenable to inductive proof. What is needed here is an enhanced notion of induction that, when specialized to rose trees, will propagate the predicate  $P$  through the outer list structure and to the rose trees sitting inside `node`'s list argument. More generally, this enhanced notion of induction should traverse *all* of the layers present in a data structure, propagating suitable predicates to *all* of the data it contains. With data types becoming ever more advanced, and with deeply structured such types becoming increasingly ubiquitous in formalizations, it is critically important that proof assistants are able to automatically generate genuinely useful induction rules for data types that go well beyond traditional ADTs. Such data types include not just deep ADTs, but also (truly<sup>2</sup>) nested types [3], generalized algebraic data types (GADTs) [4,18,20,23], more richly indexed families [5], and deep variants of all of these.

---

<sup>1</sup> A data type is *deep* if it is (possibly mutually inductively) defined in terms of other such data types (including, possibly, itself).

<sup>2</sup> A truly nested type is a nested type that is defined over itself. The data type Bush in Section 2 provides a concrete example.

*Deep induction* [15] is a generalization of structural induction that fits this bill exactly. Whereas structural induction rules induct over only the top-level structure of data, leaving any data internal to the top-level structure untouched, deep induction rules induct over *all* of the structured data present. The key idea is to parameterize induction rules not just over a predicate over the top-level data type being considered, but also over additional custom predicates on the types of primitive data they contain. These custom predicates are then lifted to predicates on any internal structures 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.

Deep induction was shown in [15] to deliver induction rules appropriate to deep nested types, including deep ADTs. For the deep ADT of rose trees, for example, it gives the following genuinely useful induction rule:

$$\begin{aligned} & \forall (A : \text{Set}) (P : \text{Rose } A \rightarrow \text{Set}) (Q : A \rightarrow \text{Set}) \rightarrow P \text{ empty} \rightarrow \\ & (\forall (a : A) (ts : \text{List } (\text{Rose } A)) \rightarrow Q a \rightarrow \text{List}^\wedge (\text{Rose } A) P ts \rightarrow P (\text{node } a ts)) \rightarrow \\ & \forall (x : \text{Rose } A) \rightarrow \text{Rose}^\wedge A Q x \rightarrow P x \end{aligned} \tag{1}$$

Here,  $\text{List}^\wedge$  (resp.,  $\text{Rose}^\wedge$ ) lifts its predicate argument  $P$  (resp.,  $Q$ ) on data of type  $\text{Rose } A$  (resp.,  $A$ ) to a predicate on data of type  $\text{List } (\text{Rose } A)$  (resp.,  $\text{Rose } A$ ) asserting that  $P$  (resp.,  $Q$ ) holds for every element of its list (resp., rose tree) argument.<sup>3</sup> Deep induction was also shown in [15] to deliver the first-ever induction rules — structural or otherwise — for the *Bush* data type [3] and other truly nested types. Deep induction for ADTs and nested types is reviewed in Section 2 below.

This paper shows how to extend deep induction to proper GADTs, i.e., to GADTs that are not simply nested types, and thus are not ADTs. A constructor for such a GADT  $G$  may, like a constructor for a nested type, take as arguments data whose types involve instances of  $G$  other than the one being defined — including instances that involve  $G$  itself. But if  $G$  is a proper GADT then at least one of its constructors will also have such a structured instance of  $G$  — albeit one not involving  $G$  itself — as its codomain. For example, the constructor *pair* for the GADT

$$\begin{aligned} & \text{data Seq} : \text{Set} \rightarrow \text{Set} \text{ where} \\ & \text{const} : \forall \{A : \text{Set}\} \rightarrow A \rightarrow \text{Seq } A \\ & \text{pair} : \forall \{A B : \text{Set}\} \rightarrow \text{Seq } A \rightarrow \text{Seq } B \rightarrow \text{Seq } (A \times B) \end{aligned} \tag{2}$$

of sequences<sup>4</sup> only constructs sequences of pairs, rather than sequences of arbitrary type, as does *const*. If one or more of the constructors for a GADT  $G$  return structured instances of  $G$ , then the GADT will have two distinct, but equally natural, semantics: a functorial semantics interpreting it as a left Kan extension [16], and a parametric semantics interpreting it as the interpretation of its Church encoding [1,22]. As detailed in [13], a key difference in the two semantics is that the former views GADTs as their *functorial completions* [14], and thus as containing more data than just those expressible in syntax. By contrast, the latter views them as what might be called *syntax-only* GADTs. Happily, these two views of GADTs coincide for those GADTs that are ADTs or other nested types. However, both they and their attendant properties differ greatly for proper GADTs. In fact, the functorial and parametric semantics for proper GADTs are sufficiently disparate that, by contrast with the semantics customarily given for ADTs and other nested types [2,9,12], it is not actually possible to define a functorial parametric semantics for them [13].

This observation seems, at first, to be a death knell for the prospect of extending deep induction to GADTs. Indeed, induction can be seen as unary parametricity, so we quickly realize that GADTs viewed as their functorial completions cannot possibly support induction rules. This makes sense intuitively: induction is a syntactic proof technique, so of course it cannot be used to prove properties of those elements of a GADT's functorial completion that are not expressible in syntax. All is not lost, however. As we show below, the syntax-only view of GADTs determined by their Church encodings *does* support induction rules — including deep induction rules — for GADTs. Indeed, this paper gives the first-ever induction rules — deep or otherwise — for proper GADTs. But it actually delivers far more: it gives a general framework for deriving deep induction rules for deep GADTs directly from their syntax. This framework can serve as a basis for extending modern proof assistants' automatic generation of structural induction rules for ADTs to automatic generation of deep induction rules for GADTs. In addition, as for ADTs and other nested types, the structural induction rule for

<sup>3</sup> Predicate liftings such as  $\text{List}^\wedge$  and  $\text{Rose}^\wedge$  can either be supplied as primitives or generated automatically from their associated data type definitions as described in Section 2 below. The predicate lifting for a container type like  $\text{List } t$  or  $\text{Rose } t$  simply traverses containers of that type and applies its predicate argument pointwise to the constituent data of type  $t$ .

<sup>4</sup> The type of *Seq* is actually  $\text{Set} \rightarrow \text{Set}_1$ , but to aid readability we elide the explicit tracking of Agda universe levels in this paper.

any GADT can be recovered from its deep induction rule simply by taking the custom predicates in its deep induction rule to be constantly `True`-valued (i.e., constantly  $\top$ -valued) predicates.

Significantly, deep induction rules for GADTs cannot be derived by somehow extending the approach of [15] to syntax-only GADTs. Indeed, the approach taken there makes crucial use of the functoriality of data types' interpretations from [14], and functoriality is precisely what interpreting GADTs as the interpretations of their Church encodings fails to deliver. Our approach is to instead first give a predicate lifting styled after those of [15], together with a (deep) induction rule, for the simplest — and arguably most important — GADT, namely the equality GADT. (See Section 4.1.) We then derive the deep induction rule for a more complex GADT  $G$  by *i*) using the equality GADT to represent  $G$  as its so-called *Henry Ford encoding* [4,10,17,19,20], and *ii*) using the predicate liftings for the equality GADT, and any other GADTs appearing in the definition of  $G$ , to appropriately thread the custom predicates for the primitive types appearing in  $G$  throughout  $G$ 's structure. This two-step process delivers deep induction rules for a very general class of deep GADTs. In Section 3 we introduce a series of increasingly complex GADTs as running examples, and in Section 4 we derive a deep induction rule for each of them. In particular, we derive the deep induction rule for the `Seq` data type in (2) in Section 4.2. We present our general framework for deriving (deep) induction rules for (deep) GADTs in Section 5, and observe that the derivations in Section 4 are all instances of it. In Section 6 we show that, by contrast with truly nested types, which do have a functorial semantics, syntax-only GADTs' lack of functoriality means that it is not possible to extend induction — deep or otherwise — to truly nested GADTs. This does not appear to be much of a restriction, however, since GADTs defined over themselves do not, to our knowledge, appear in practice or in the literature. Section 7 comprises a case study in using deep induction. All of the deep induction rules appearing in this paper have been derived by instantiating our general framework. Our Agda code implementing them is available at [11].

**Additional Related Work** Various techniques for deriving induction rules for data types that go beyond ADTs have been studied. For example, Fu and Selinger [8] show, via examples, how to derive induction rules for arbitrary nested types. Unfortunately, however, their technique is rather *ad hoc*, so is unclear how to generalize it to nested types other than the specific ones studied there. Moreover, [8] actually derives induction rules for data types *related* to the original nested types rather than for the original nested types themselves, and it is unclear whether or not the derived rules are sufficiently expressive to prove all results about the original nested types that we would expect to be provable by induction. This latter point echoes the issue with Coq-derived induction rule for rose trees raised in Section 1, which has the unfortunate effect of forcing users to manually write induction (and other) rules for such types for use in that system. Tassi [21] derives induction rules for data type definitions in Coq using unary parametricity. His technique seems to be essentially equivalent to that of [14] for nested types, although he does not permit true nesting. More recently, Ullrich [24] has implemented a plugin in MetaCoq to generate induction rules for nested types. This plugin is also based on unary parametricity and true nesting still is not permitted. As far as we know, no attempts have been made to extend either implementation to truly nested types or proper GADTs or their deep variants. In fact, we know of no work other than that reported here that specifically addresses induction rules for such data types.

## 2 Deep induction for ADTs and nested types

A structural induction rule for a data type allows us to prove that if a predicate holds for every element inductively produced by the data type's constructors then it holds for every element of the data type. In this paper, we are interested in induction rules for proof-relevant predicates. A proof-relevant predicate on  $A : \text{Set}$  is a function  $P : A \rightarrow \text{Set}$  mapping each  $a : A$  to the set of proofs that  $P a$  holds. For example, the structural induction rule for the list type

```
data List : Set → Set where
  nil   : ∀{A : Set} → List A
  cons  : ∀{A : Set} → A → List A → List A
```

is

$$\forall (A : \text{Set}) (P : \text{List } A \rightarrow \text{Set}) \rightarrow P \text{ nil} \rightarrow (\forall (a : A) (as : \text{List } A) \rightarrow P a \rightarrow P (\text{cons } a \text{ as})) \rightarrow \forall (as : \text{List } A) \rightarrow P as$$

As in Coq's induction rule for rose trees, the data inside a structure of type `List` is treated monolithically (i.e., is ignored) by this structural induction rule. By contrast, the deep induction rule for lists is parameterized over a custom predicate  $Q$  on  $A$ . For `List^` as described in the introduction the deep induction rule for lists is

$$\forall (A : \text{Set}) (P : \text{List } A \rightarrow \text{Set}) (Q : A \rightarrow \text{Set}) \rightarrow P \text{ nil} \rightarrow (\forall (a : A) (as : \text{List } A) \rightarrow Q a \rightarrow P a \rightarrow P (\text{cons } a \text{ as})) \rightarrow \forall (as : \text{List } A) \rightarrow \text{List}^{\wedge} A Q as \rightarrow P as$$

Structural induction can be extended to nested types, such as the following type of perfect trees [3]:

```

data PTree : Set → Set where
  pleaf  : ∀{A : Set} → A → PTree A
  pnode  : ∀{A : Set} → PTree (A × A) → PTree A

```

Perfect trees can be thought of as lists constrained to have lengths that are powers of 2. In the above code, the constructor `pnode` uses data of type `PTree (A × A)` to construct data of type `PTree A`. Thus, it is clear that the instances of `PTree` at various indices cannot be defined independently, and that the entire inductive family of types must therefore be defined at once. This intertwinedness of the instances of nested types is reflected in their structural induction rules, which, as explained in [15], must necessarily involve polymorphic predicates rather than the monomorphic predicates appearing in structural induction rules for ADTs. The structural induction rule for perfect trees, for example, is

$$\begin{aligned} & \forall (P : \forall (A : \text{Set}) \rightarrow \text{PTree } A \rightarrow \text{Set}) \rightarrow (\forall (A : \text{Set}) (a : A) \rightarrow P \ A \ (pleaf \ a)) \rightarrow \\ & (\forall (A : \text{Set}) (pp : \text{PTree } (A \times A)) \rightarrow P \ (A \times A) \ pp \rightarrow P \ A \ (pnode \ pp)) \rightarrow \forall (A : \text{Set}) (p : \text{PTree } A) \rightarrow P \ A \ p \end{aligned}$$

The deep induction rule for perfect trees similarly uses polymorphic predicates but otherwise follows the now-familiar pattern:

$$\begin{aligned} & \forall (P : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{PTree } A \rightarrow \text{Set}) \rightarrow (\forall (A : \text{Set}) (Q : A \rightarrow \text{Set}) (a : A) \rightarrow Q \ a \rightarrow P \ A \ Q \ (pleaf \ a)) \rightarrow \\ & (\forall (A : \text{Set}) (Q : A \rightarrow \text{Set}) (pp : \text{PTree } (A \times A)) \rightarrow P \ (A \times A) \ (Pair^{\wedge} \ A \ A \ Q \ Q) \ pp \rightarrow P \ A \ Q \ (pnode \ pp)) \rightarrow \\ & \forall (A : \text{Set}) (Q : A \rightarrow \text{Set}) (p : \text{PTree } A) \rightarrow PTree^{\wedge} \ A \ Q \ p \rightarrow P \ A \ Q \ p \end{aligned}$$

Here,  $Pair^{\wedge} : \forall (A \ B : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow (B \rightarrow \text{Set}) \rightarrow A \times B \rightarrow \text{Set}$  lifts predicates  $Q_A$  on data of type  $A$  and  $Q_B$  on data of type  $B$  to a predicate on pairs of type  $A \times B$  in such a way that  $Pair^{\wedge} \ A \ B \ Q_A \ Q_B \ (a, b) = Q_A \ a \times Q_B \ b$ . Similarly,  $PTree^{\wedge} : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{PTree } A \rightarrow \text{Set}$  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 perfect tree argument.

It is not possible to extend structural induction to *truly* nested types, i.e., to nested types whose recursive occurrences appear below themselves. The quintessential example of such a type is that of bushes<sup>5</sup> [3]:

```

data Bush : Set → Set where
  bnil   : ∀{A : Set} → Bush A
  bcons  : ∀{A : Set} → A → Bush (Bush A) → Bush A

```

Even defining a structural induction rule for bushes requires that we be able to lift the rule's polymorphic predicate argument to `Bush` itself. The more general observation that an induction rule for any truly nested type must therefore necessarily be a deep induction rule was, in fact, the original motivation for the development of deep induction in [15]. The deep induction rule for bushes is

$$\begin{aligned} & \forall (P : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{Bush } A \rightarrow \text{Set}) \rightarrow (\forall (A : \text{Set}) (Q : A \rightarrow \text{Set}) \rightarrow P \ A \ Q \ bnil) \rightarrow \\ & (\forall (A : \text{Set}) (Q : A \rightarrow \text{Set}) (a : A) (bb : \text{Bush } (\text{Bush } A)) \rightarrow Q \ a \rightarrow P \ (\text{Bush } A) \ (P \ A \ Q) \ bb \rightarrow P \ A \ Q \ (bcons \ a \ bb)) \rightarrow \\ & \forall (A : \text{Set}) (Q : A \rightarrow \text{Set}) (b : \text{Bush } A) \rightarrow Bush^{\wedge} \ A \ Q \ b \rightarrow P \ A \ Q \ b \end{aligned}$$

Here,  $Bush^{\wedge} : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{Bush } A \rightarrow \text{Set}$  is the following lifting of 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`:

$$\begin{aligned} & Bush^{\wedge} \ A \ Q \ bnil = \top \\ & Bush^{\wedge} \ A \ Q \ (bcons \ a \ bb) = Q \ a \times Bush^{\wedge} \ (\text{Bush } A) \ (Bush^{\wedge} \ A \ Q) \ bb \end{aligned}$$

Although a truly nested type admits only a single induction rule, it is worth noting that for those nested types that do admit distinct structural induction and deep induction rules, the latter always generalize the former. Indeed, the structural induction rule for each such nested type is recoverable from its deep induction rule by taking the custom predicates on its data of primitive types to be constantly  $\top$ -valued predicates. This instantiation ensures that the resulting induction rule only inspects the top-level structure of its argument, rather than the contents of that structure, which is exactly what structural induction should do.

<sup>5</sup> To define truly nested types in Agda we must use the `NO_POSITIVITY_CHECK` flag. A similar flag is required in Coq.

### 3 (Deep) GADTs

While a data constructor for a nested type can take *as arguments* data whose types involve instances of that type at indices other than the one being defined, its return type must still be at the (variable) type instance being defined. For example, each of `pleaf` and `pnode` returns an element of type `PTreeA` regardless of the instances of `PTree` appearing in the types of its arguments. GADTs relax this restriction, allowing their data constructors both to take as arguments *and return as results* data whose types involve instances other than the one being defined. That is, GADTs' constructors' return type instances can, like that of `pair` in (2), be structured.

GADTs are used in precisely those situations in which different behaviors at different instances of data types are desired. This is achieved by allowing the programmer to give the type signatures of the GADT's data constructors independently, and then taking advantage of pattern matching to force the desired type refinement. For example, the *equality* GADT

$$\begin{aligned} \text{data Equal} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} \text{ where} \\ \text{refl} : \forall \{A : \text{Set}\} \rightarrow \text{Equal } A \ A \end{aligned} \quad (3)$$

is parameterized by two type indices, but it is only possible to construct data elements of type `Equal A B` if `A` and `B` are instantiated at the same type. If the types `A` and `B` are syntactically identical then the type `Equal A B` contains the single data element `refl`. It contains no data elements otherwise.

The importance of the equality GADT lies in the fact that we can understand other GADTs in terms of it. For example, the GADT `Seq` from (2) comprises constant sequences of data of any type `A` and sequences obtained by pairing the data in two already existing sequences. This GADT can be rewritten as its Henry Ford encoding [4,10,17,19,20], which makes critical use of the equality GADT, as follows:

$$\begin{aligned} \text{data Seq} : \text{Set} \rightarrow \text{Set} \text{ where} \\ \text{const} : \forall \{A : \text{Set}\} \rightarrow A \rightarrow \text{Seq } A \\ \text{pair} : \forall \{A : \text{Set}\} \rightarrow \forall (B C : \text{Set}) \rightarrow \text{Equal } A (B \times C) \rightarrow \text{Seq } B \rightarrow \text{Seq } C \rightarrow \text{Seq } A \end{aligned} \quad (4)$$

Here, the requirement that `pair` produce data at an instance of `Seq` that is a product type is replaced with the requirement that `pair` produce data at an instance of `Seq` that is *equal* to a product type. As we will see in Section 4, this encoding in terms of the equality GADT is key to deriving deep induction rules for GADTs.

Neither `Equal` nor `Seq` is a deep GADT, but the following GADT `LTerm`, inspired by [6], is. It encodes terms of a simply typed lambda calculus. More robust variations on `LTerm` are, of course, possible, but this variation is rich enough to illustrate all essential aspects of deep GADTs — and later, in Section 4.3, their deep induction rules — while still being small enough to ensure clarity of exposition.

Types are either booleans, arrow types, or list types. They are represented by the Henry Ford GADT

$$\begin{aligned} \text{data LType} : \text{Set} \rightarrow \text{Set} \text{ where} \\ \text{bool} : \forall \{A : \text{Set}\} \rightarrow \forall (B : \text{Set}) \rightarrow \text{Equal } A \text{ Bool} \rightarrow \text{LType } A \\ \text{arr} : \forall \{A : \text{Set}\} \rightarrow \forall (B C : \text{Set}) \rightarrow \text{Equal } A (B \rightarrow C) \rightarrow \text{LType } B \rightarrow \text{LType } C \rightarrow \text{LType } A \\ \text{list} : \forall \{A : \text{Set}\} \rightarrow \forall (B : \text{Set}) \rightarrow \text{Equal } A (\text{List } B) \rightarrow \text{LType } B \rightarrow \text{LType } A \end{aligned} \quad (5)$$

Terms are either variables, abstractions, applications, or lists of terms. They are similarly represented by

$$\begin{aligned} \text{data LTerm} : \text{Set} \rightarrow \text{Set} \text{ where} \\ \text{var} : \forall \{A : \text{Set}\} \rightarrow \text{String} \rightarrow \text{LType } A \rightarrow \text{LTerm } A \\ \text{abs} : \forall \{A : \text{Set}\} \rightarrow \forall (B C : \text{Set}) \rightarrow \text{Equal } A (B \rightarrow C) \rightarrow \text{String} \rightarrow \text{LType } B \rightarrow \text{LTerm } C \rightarrow \text{LTerm } A \\ \text{app} : \forall \{A : \text{Set}\} \rightarrow \forall (B : \text{Set}) \rightarrow \text{LTerm } (B \rightarrow A) \rightarrow \text{LTerm } B \rightarrow \text{LTerm } A \\ \text{list} : \forall \{A : \text{Set}\} \rightarrow \forall (B : \text{Set}) \rightarrow \text{Equal } A (\text{List } B) \rightarrow \text{List } (\text{LTerm } B) \rightarrow \text{LTerm } A \end{aligned} \quad (6)$$

The type parameter for `LTerm` tracks the types of simply typed lambda calculus terms. For example, `LTerm A` is the type of simply typed lambda terms of type `A`. Variables are tagged with their types by the data constructors `var` and `abs`, whose `LType` arguments ensure that their type tags are legal types. This ensures that all lambda terms produced by `var`, `abs`, `app`, and `list` are well-typed. We will revisit these GADTs in Sections 4 and 7.

## 4 (Deep) induction for GADTs

The equality constraints engendered by GADTs' data constructors makes deriving (deep) induction rules for them more involved than for ADTs and other nested types. Nevertheless, we show in this section how to do so. We first illustrate the key components of our approach by deriving deep induction rules for the three specific GADTs introduced in Section 3. Then, in Section 5, we abstract these to a general framework that can be applied to any deep GADT that is not truly nested. As hinted above, the predicate lifting for the equality GADT plays a central role in deriving both structural and deep induction rules for more general GADTs.

### 4.1 (Deep) induction for Equal

To define the (deep) induction rule for any (deep) GADT  $G$  we first need to define a predicate lifting that maps a predicate on a type  $A$  to a predicate on  $GA$ . Such a predicate lifting  $\text{Equal}^\wedge : \forall (A B : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow (B \rightarrow \text{Set}) \rightarrow \text{Equal } A B \rightarrow \text{Set}$  for  $\text{Equal}$  is defined by  $\text{Equal}^\wedge A A Q Q' \text{refl} = \forall (a : A) \rightarrow \text{Equal } (Q a) (Q' a)$ . It takes two predicates on the same type as input and tests them for extensional equality. Next, we need to associate with each data constructor  $c$  of  $G$  an *induction hypothesis* asserting that, if the custom predicate arguments to a predicate  $P$  on  $G$  can be lifted to  $G$  itself, then  $c$  respects  $P$ , i.e.,  $c$  constructs data satisfying the instance of  $P$  at those custom predicates. The following induction hypothesis  $\text{dIndRefl}$  is thus associated with the  $\text{refl}$  constructor for  $\text{Equal}$ :

$$\begin{aligned} \lambda(P : \forall (A B : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow (B \rightarrow \text{Set}) \rightarrow \text{Equal } A B \rightarrow \text{Set}) \rightarrow \\ \forall (C : \text{Set}) (Q Q' : C \rightarrow \text{Set}) \rightarrow \text{Equal}^\wedge C C Q Q' \text{refl} \rightarrow P C C Q Q' \text{refl} \end{aligned}$$

The deep induction rule for  $G$  now states that, if all of  $G$ 's data constructors respect a predicate  $P$ , then  $P$  is satisfied by every element of  $G$  to which the custom predicate arguments to  $P$  can be successfully lifted. The deep induction rule for  $\text{Equal}$  is thus

$$\begin{aligned} \forall (P : \forall (A B : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow (B \rightarrow \text{Set}) \rightarrow \text{Equal } A B \rightarrow \text{Set}) \rightarrow \text{dIndRefl } P \rightarrow \\ \forall (A B : \text{Set}) (Q_A : A \rightarrow \text{Set}) (Q_B : B \rightarrow \text{Set}) (e : \text{Equal } A B) \rightarrow \text{Equal}^\wedge A B Q_A Q_B e \rightarrow P A B Q_A Q_B e \end{aligned} \quad (7)$$

To prove that this rule is sound we must provide a witness  $\text{dIndEqual}$  inhabiting the type in (7). By pattern matching, we need only consider the case where  $A = B$  and  $e = \text{refl}$ , so we can define  $\text{dIndEqual}$  by  $\text{dIndEqual } P \text{crefl } A A Q_A Q'_A \text{refl liftE} = \text{crefl } A Q_A Q'_A \text{liftE}$ . To recover  $\text{Equal}$ 's structural induction rule

$$\forall (Q : \forall (A B : \text{Set}) \rightarrow \text{Equal } A B \rightarrow \text{Set}) \rightarrow (\forall (C : \text{Set}) \rightarrow Q C C \text{refl}) \rightarrow \forall (A B : \text{Set}) (e : \text{Equal } A B) \rightarrow Q A B e \quad (8)$$

we define a term  $\text{indEqual}$  of the type in (8) by  $\text{indEqual } Q \text{srefl } A B \text{refl} = \text{dIndEqual } P \text{srefl}' A B K_T^A K_T^B \text{refl sliftE}$ . Here,  $P : \forall (A B : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow (B \rightarrow \text{Set}) \rightarrow \text{Equal } A B \rightarrow \text{Set}$  is defined by  $P A B Q_A Q_B e = Q A B e$ ,  $K_T^A$  and  $K_T^B$  are the constantly  $\top$ -valued predicates on  $A$  and  $B$ , respectively,  $\text{sliftE} : \text{Equal}^\wedge A B K_T^A K_T^B \text{refl}$  is defined by  $\text{sliftE } a = \text{refl} : \text{Equal } \top \top$  for every  $a : A$ , and  $\text{srefl}' : \forall (C : \text{Set}) (Q_C Q'_C : C \rightarrow \text{Set}) \rightarrow \text{Equal}^\wedge C C Q_C Q'_C \text{refl} \rightarrow Q C C \text{refl}$  is defined by  $\text{srefl}' C Q_C Q'_C \text{liftE}' = \text{srefl } C$ . The structural induction rule for any GADT  $G$  that is not truly nested can similarly be recovered from its deep induction rule by instantiating every custom predicate by the appropriate constantly  $\top$ -valued predicate.

### 4.2 (Deep) induction for Seq

To derive the deep induction rule for the GADT  $\text{Seq}$  we use its Henry Ford encoding from (4). We first define its predicate lifting  $\text{Seq}^\wedge : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{Seq } A \rightarrow \text{Set}$  by

$$\begin{aligned} \text{Seq}^\wedge A Q_A (\text{const } a) &= Q_A a \\ \text{Seq}^\wedge A Q_A (\text{pair } B C e_S s_C) &= \exists [Q_B] \exists [Q_C] \text{Equal}^\wedge A (B \times C) Q_A (\text{Pair}^\wedge B C Q_B Q_C) e \times \text{Seq}^\wedge B Q_B s_B \times \text{Seq}^\wedge C Q_C s_C \end{aligned}$$

Here,  $a : A$ ,  $Q_B : B \rightarrow \text{Set}$ ,  $Q_C : C \rightarrow \text{Set}$ ,  $e : \text{Equal } A (B \times C)$ ,  $s_B : \text{Seq } B$ ,  $s_C : \text{Seq } C$ , and  $\exists [x] F x$  is syntactic sugar for the type of dependent pairs  $(x, b)$ , where  $x : A$ ,  $b : F x$ , and  $F : A \rightarrow \text{Set}$ .

Next, let  $\text{dIndConst}$  be the induction hypothesis

$$\lambda(P : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{Seq } A \rightarrow \text{Set}) \rightarrow \forall (A : \text{Set}) (Q_A : A \rightarrow \text{Set}) (a : A) \rightarrow Q_A a \rightarrow P A Q_A (\text{const } a)$$

associated with the constructor  $\text{const}$ , and let  $\text{dIndPair}$  be the induction hypothesis



$LType^A A Q_A (bool B e)$	$= \exists [Q_B] Equal^A A B Q_A K_T^{Bool} e$
$LType^A A Q_A (arr B C e T_B T_C)$	$= \exists [Q_B] \exists [Q_C] Equal^A A (B \rightarrow C) Q_A (Arr^A B C Q_B Q_C) e \times LType^A B Q_B T_B \times LType^A C Q_C T_C$
$LType^A A Q_A (list B e T_B)$	$= \exists [Q_B] Equal^A A (List B) Q_A (List^A B Q_B) e \times LType^A B Q_B T_B$
$LTerm^A A Q_A (vars T_A)$	$= LType^A A Q_A T_A$
$LTerm^A A Q_A (abs B C e s T_B t_C)$	$= \exists [Q_B] \exists [Q_C] Equal^A A (B \rightarrow C) Q_A (Arr^A B C Q_B Q_C) e \times LType^A B Q_B T_B \times LTerm^A C Q_C t_C$
$LTerm^A A Q_A (app B t_{BA} t_B)$	$= \exists [Q_B] LTerm^A (B \rightarrow A) (Arr^A B A Q_B Q_A) t_{BA} \times LTerm^A B Q_B t_B$
$LTerm^A A Q_A (list B e ts)$	$= \exists [Q_B] Equal^A A (List B) Q_A (List^A B Q_B) e \times List^A (LTerm B) (LTerm^A B Q_B) ts$

Fig. 1. Predicate liftings for  $LType$  and  $LTerm$ 

$$\begin{aligned} & \lambda(P : \forall(A : Set) \rightarrow (A \rightarrow Set) \rightarrow Seq A \rightarrow Set) \rightarrow \\ & \quad \forall(ABC : Set)(Q_A : A \rightarrow Set)(Q_B : B \rightarrow Set)(Q_C : C \rightarrow Set)(s_B : Seq B)(s_C : Seq C)(e : Equal A (B \times C)) \rightarrow \\ & \quad Equal^A A (B \times C) Q_A (Pair^A B C Q_B Q_C) e \rightarrow P B Q_B s_B \rightarrow P C Q_C s_C \rightarrow P A Q_A (pair B C e s_B s_C) \end{aligned}$$

associated with the constructor `pair`. Then the deep induction rule for `Seq` is

$$\begin{aligned} & \forall(P : \forall(A : Set) \rightarrow (A \rightarrow Set) \rightarrow Seq A \rightarrow Set) \rightarrow dIndConst P \rightarrow dIndPair P \rightarrow \\ & \quad \forall(A : Set)(Q_A : A \rightarrow Set)(s_A : Seq A) \rightarrow Seq^A A Q_A s_A \rightarrow P A Q_A s_A \end{aligned} \quad (9)$$

To prove that this rule is sound we provide a witness `dIndSeq` inhabiting the type in (9) as follows:

$$\begin{aligned} dIndSeq P cconst cpair A Q_A (const a) liftA &= cconst A Q_A a liftA \\ dIndSeq P cconst cpair A Q_A (pair B C e s_B s_C) (Q_B, Q_C, liftE, liftB, liftC) &= cpair A B C Q_A Q_B Q_C s_B s_C e liftE p_B p_C \end{aligned}$$

In the first clause,  $a : A$ ,  $Q_A : A \rightarrow Set$ , and  $liftA : Seq^A A Q_A (const a) = Q_A a$ . In the second clause we also have  $Q_B : B \rightarrow Set$ ,  $Q_C : C \rightarrow Set$ ,  $e : Equal A (B \times C)$ ,  $s_B : Seq B$ ,  $s_C : Seq C$ ,  $liftE : Equal^A A (B \times C) Q_A (Pair^A B C Q_B Q_C) e$ ,  $liftB : Seq^A B Q_B s_B$ , and  $liftC : Seq^A C Q_C s_C$  — which together ensure that  $(Q_B, Q_C, liftE, liftB, liftC) : Seq^A A Q (pair B C e s_B s_C)$  — and  $p_B = dIndSeq P cconst cpair B Q_B s_B liftB : P B Q_B s_B$  and  $p_C = dIndSeq P cconst cpair C Q_C s_C liftC : P C Q_C s_C$ .

#### 4.3 (Deep) induction for $LTerm$

To derive the deep induction rule for the GADT  $LTerm$  we use its Henry Ford encoding from (5) and (6). We first define the predicate lifting  $Arr^A : \forall(AB : Set) \rightarrow (A \rightarrow Set) \rightarrow (B \rightarrow Set) \rightarrow (A \rightarrow B) \rightarrow Set$  for arrow types, since arrow types appear in  $LType$  and  $LTerm$ . It is given by  $Arr^A A B Q_A Q_B f = \forall(a : A) \rightarrow Q_A a \rightarrow Q_B (f a)$ . The predicate liftings  $LType^A : \forall(A : Set) \rightarrow (A \rightarrow Set) \rightarrow LType A \rightarrow Set$  for  $LType$  and  $LTerm^A : \forall(A : Set) \rightarrow (A \rightarrow Set) \rightarrow LTerm A \rightarrow Set$  for  $LTerm$  are defined in Figure 1. There,  $s : String$ ,  $Q_A : A \rightarrow Set$ ,  $Q_B : B \rightarrow Set$ ,  $Q_C : C \rightarrow Set$ ,  $K_T^{Bool}$  is the constantly  $\top$ -valued predicate on `Bool`,  $T_A : LType A$ ,  $T_B : LType B$ ,  $T_C : LType C$ ,  $t_B : LTerm B$ ,  $t_C : LTerm C$ , and  $t_{BA} : LTerm (B \rightarrow A)$ . Moreover,  $e : Equal A Bool$  in the first clause,  $e : Equal A (B \rightarrow C)$  in the second,  $e : Equal A (List B)$  in the third,  $e : Equal A (B \rightarrow C)$  in the fifth, and  $e : Equal A (List B)$ ,  $ts : List (LTerm B)$ , and  $List^A$  is the predicate lifting for lists from (1) in the seventh.

With these liftings in hand we can define the induction hypotheses `dIndVar`, `dIndAbs`, `dIndApp`, and `dIndList` associated with  $LTerms$ 's data constructors. These are, respectively,

$$\begin{aligned} & \lambda(P : \forall(A : Set) \rightarrow (A \rightarrow Set) \rightarrow LTerm A \rightarrow Set) \rightarrow \\ & \quad \forall(A : Set)(Q_A : A \rightarrow Set)(s : String)(T_A : LType A) \rightarrow LType^A A Q_A T_A \rightarrow P A Q_A (var s T_A) \\ & \lambda(P : \forall(A : Set) \rightarrow (A \rightarrow Set) \rightarrow LTerm A \rightarrow Set) \rightarrow \\ & \quad \forall(ABC : Set)(Q_A : A \rightarrow Set)(Q_B : B \rightarrow Set)(Q_C : C \rightarrow Set)(e : Equal A (B \rightarrow C))(s : String) \rightarrow \\ & \quad (T_B : LType B) \rightarrow (t_C : LTerm C) \rightarrow Equal^A A (B \rightarrow C) Q_A (Arr^A B C Q_B Q_C) e \rightarrow \\ & \quad LType^A B Q_B T_B \rightarrow P C Q_C t_C \rightarrow P A Q_A (abs B C e s T_B t_C) \end{aligned}$$

$\text{dIndLTerm } P \text{ cvar cabs capp clist } A \ Q_A \ (\text{var } s \ T_A) \ \text{lift}_A$	$= \text{cvar } A \ Q_A \ s \ T_A \ \text{lift}_A$
$\text{dIndLTerm } P \text{ cvar cabs capp clist } A \ Q_A \ (\text{abs } B \ C \ \text{es } T_B \ t_C) \ (Q_B, Q_C, \text{lift}_E, \text{lift}_{T_B}, \text{lift}_{t_C})$	$= \text{cabs } A \ B \ C \ Q_A \ Q_B \ Q_C \ \text{es } T_B \ t_C \ \text{lift}_E \ \text{lift}_{T_B} \ P_C$
$\text{dIndLTerm } P \text{ cvar cabs capp clist } A \ Q_A \ (\text{app } B \ t_{BA} \ t_B) \ (Q_B, \text{lift}_{t_{BA}}, \text{lift}_{t_B})$	$= \text{capp } A \ B \ Q_A \ Q_B \ t_{BA} \ t_B \ P_{BA} \ P_B$
$\text{dIndLTerm } P \text{ cvar cabs capp clist } A \ Q_A \ (\text{list } B \ \text{ets}) \ (Q_B, \text{lift}_{E'}, \text{lift}_{\text{List}})$	$= \text{clist } A \ B \ Q_A \ Q_B \ \text{ets} \ \text{lift}_{E'} \ P_{\text{List}}$

Fig. 2.  $\text{dIndLTerm}$ 

$$\begin{aligned}
& \lambda(P : \forall(A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{LTerm } A \rightarrow \text{Set}) \rightarrow \\
& \quad \forall(A B : \text{Set})(Q_A : A \rightarrow \text{Set})(Q_B : B \rightarrow \text{Set})(t_{BA} : \text{LTerm } (B \rightarrow A))(t_B : \text{LTerm } B) \rightarrow \\
& \quad P(B \rightarrow A) (\text{Arr}^\wedge B A Q_B Q_A) t_{BA} \rightarrow P B Q_B t_B \rightarrow P A Q_A (\text{app } B t_{BA} t_B) \\
& \lambda(P : \forall(A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{LTerm } A \rightarrow \text{Set}) \rightarrow \\
& \quad \forall(A B : \text{Set})(Q_A : A \rightarrow \text{Set})(Q_B : B \rightarrow \text{Set})(e : \text{Equal } A (\text{List } B))(ts : \text{List } (\text{LTerm } B)) \rightarrow \\
& \quad \text{Equal}^\wedge A (\text{List } B) Q_A (\text{List}^\wedge B Q_B) e \rightarrow \text{List}^\wedge (\text{LTerm } B) (P B Q_B) ts \rightarrow P A Q_A (\text{list } B \ \text{ets})
\end{aligned}$$

The deep induction rule for  $\text{LTerm}$  is thus

$$\begin{aligned}
& \forall(P : \forall(A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{LTerm } A \rightarrow \text{Set}) \rightarrow \text{dIndVar } P \rightarrow \text{dIndAbs } P \rightarrow \text{dIndApp } P \rightarrow \text{dIndList } P \rightarrow \\
& \quad \forall(A : \text{Set})(Q_A : A \rightarrow \text{Set})(t_A : \text{LTerm } A) \rightarrow \text{LTerm}^\wedge A Q_A t_A \rightarrow P A Q_A t_A
\end{aligned} \tag{10}$$

To prove this rule is sound we define a witness  $\text{dIndLTerm}$  inhabiting the type in (10) as in Figure 2. There,  $s : \text{String}$ ,  $Q_A : A \rightarrow \text{Set}$ ,  $Q_B : B \rightarrow \text{Set}$ ,  $Q_C : C \rightarrow \text{Set}$ ,  $T_A : \text{LType } A$ ,  $T_B : \text{LType } B$ ,  $t_B : \text{LTerm } B$ ,  $t_C : \text{LTerm } C$ ,  $t_{BA} : \text{LTerm } (B \rightarrow A)$ ,  $\text{lift}_A : \text{LTerm}^\wedge A Q_A (\text{var } s \ T_A) = \text{LType}^\wedge A Q_A T_A$ ,  $\text{lift}_E : \text{Equal}^\wedge A (B \rightarrow C) Q_A (\text{Arr}^\wedge B C Q_B Q_C) e$ ,  $\text{lift}_{T_B} : \text{LType}^\wedge B Q_B T_B$ ,  $\text{lift}_{t_C} : \text{LTerm}^\wedge C Q_C T_C$ ,  $\text{lift}_{t_{BA}} : \text{LTerm}^\wedge (B \rightarrow A) (\text{Arr}^\wedge B A Q_B Q_A) t_{BA}$ ,  $\text{lift}_{t_B} : \text{LTerm}^\wedge B Q_B t_B$ ,  $\text{lift}_{E'} : \text{Equal}^\wedge A (\text{List } B) Q_A (\text{List}^\wedge B Q_B) e$ , and  $\text{lift}_{\text{List}} : \text{List}^\wedge (\text{LTerm } B) (\text{LTerm}^\wedge B Q_B) ts$ . Moreover,

$$\begin{aligned}
p_C &= \text{dIndLTerm } P \text{ cvar cabs capp clist } C \ Q_C \ t_C \ \text{lift}_{t_C} : P C Q_C t_C \\
p_B &= \text{dIndLTerm } P \text{ cvar cabs capp clist } B \ Q_B \ t_B \ \text{lift}_{t_B} : P B Q_B t_B \\
p_{BA} &= \text{dIndLTerm } P \text{ cvar cabs capp clist } (B \rightarrow A) (\text{Arr}^\wedge B A Q_B Q_A) t_{BA} \ \text{lift}_{t_{BA}} : P (B \rightarrow A) (\text{Arr}^\wedge B A Q_B Q_A) t_{BA} \\
p_{\text{List}} &= \text{liftListMap } (\text{LTerm } B) (\text{LTerm}^\wedge B Q_B) (P B Q_B) p_{ts} \ \text{lift}_{\text{List}} : \text{List}^\wedge (\text{LTerm } B) (P B Q_B) ts \\
p_{ts} &= \text{dIndLTerm } P \text{ cvar cabs capp clist } B \ Q_B : \text{PredMap } (\text{LTerm } B) (\text{LTerm}^\wedge B Q_B) (P B Q_B)
\end{aligned}$$

where, in the final clause,  $\text{PredMap} : \forall(A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{Set}$  is the type constructor producing the type of morphisms between predicates defined by  $\text{PredMap } A \ Q \ Q' = \forall(a : A) \rightarrow Q a \rightarrow Q' a$  and  $\text{liftListMap} : \forall(A : \text{Set}) \rightarrow (Q \ Q' : A \rightarrow \text{Set}) \rightarrow \text{PredMap } A \ Q \ Q' \rightarrow \text{PredMap } (\text{List } A) (\text{List}^\wedge A \ Q) (\text{List}^\wedge A \ Q')$ , which takes a morphism  $f$  of predicates and produces a morphism of lifted predicates, is defined by  $\text{liftListMap } A \ Q \ Q' m \ \text{nil} \ \text{tt} = \text{tt}$  (since  $x : \text{List}^\wedge A \ Q \ \text{nil}$  must necessarily be the sole inhabitant  $\text{tt}$  of  $\top$ ), and by  $\text{liftListMap } A \ Q \ Q' m \ (\text{cons } a \ l') (y, x') = (m \ a \ y, \text{liftListMap } A \ Q \ Q' m \ l' \ x')$  (since  $x : \text{List}^\wedge A \ Q \ (\text{cons } a \ l')$  must be of the form  $x = (y, x')$  where  $y : Q a$  and  $x' : \text{List}^\wedge A \ Q \ l'$ ).

## 5 The general framework

We can generalize the approach taken in Section 4 to a general framework for deriving deep induction rules for deep GADTs. We will treat deep GADTs of the form

$$\begin{aligned}
& \text{data } G : \text{Set}^\alpha \rightarrow \text{Set} \text{ where} \\
& \quad c : \forall \{ \overline{B} : \text{Set} \} \rightarrow F \ G \ \overline{B} \rightarrow G(\overline{K} \ \overline{B})
\end{aligned} \tag{11}$$

For brevity and clarity we indicate only one constructor  $c$  in (11), even though a GADT can have any finite number of them, each with a type of the same form as  $c$ 's. In (11),  $F$  and each  $K$  in  $\overline{K}$  are type constructors with signatures  $(\text{Set}^\alpha \rightarrow \text{Set}) \rightarrow \text{Set}^\beta \rightarrow \text{Set}$  and  $\text{Set}^\beta \rightarrow \text{Set}$ , respectively. If  $T$  is a type constructor with signature  $\text{Set}^\gamma \rightarrow \text{Set}$  then  $T$  has *arity*  $\gamma$ . The overline notation denotes a finite list whose length is exactly the arity of the type constructor being applied to it. The number of type constructors in  $\overline{K}$  (resp.,  $\overline{B}$ ) is thus  $\alpha$  (resp.,  $\beta$ ). In addition, the type constructor  $F$  must be constructed inductively according to the following grammar:

$$F \ G \ \overline{B} := F_1 \ G \ \overline{B} \times F_2 \ G \ \overline{B} \mid F_1 \ G \ \overline{B} + F_2 \ G \ \overline{B} \mid F_1 \ \overline{B} \rightarrow F_2 \ G \ \overline{B} \mid G(\overline{F_1} \ \overline{B}) \mid H \ \overline{B} \mid H(\overline{F_1} \ G \ \overline{B})$$



This grammar is subject to the following restrictions. In the third clause the type constructor  $F_1$  does not contain  $G$ . In the fourth clause, none of the  $\alpha$ -many type constructors in  $\overline{F_1}$  contains  $G$ . This prevents nesting, which would make it impossible to give an induction rule for  $G$ ; see Section 6 below. In the fifth clause,  $H$  does not contain  $G$ . This clause therefore subsumes the cases in which  $F G \overline{B}$  is a closed type or one of the  $B_i$ . In the sixth clause,  $H : \text{Set}^\gamma \rightarrow \text{Set}$  does not contain  $G$  (although  $F_1 G \overline{B}$  can). Moreover, although  $H$  can construct any (truly) nested type, it must not construct a GADT. This ensures that  $H$  admits functorial semantics [15], and thus has an associated map function. From the map function for  $H$  we can also construct a map function

$$H^\wedge \text{Map} : \forall (\overline{A : \text{Set}}) (\overline{Q Q' : A \rightarrow \text{Set}}) \rightarrow \overline{\text{PredMap } A Q Q'} \rightarrow \text{PredMap } (H \overline{A}) (H^\wedge \overline{A Q}) (H^\wedge \overline{A Q'}) \quad (12)$$

for  $H^\wedge$ . A concrete way to define  $H^\wedge \text{Map}$  is by induction on the structure of the type  $H$ , but we omit such details since they are not essential to the present discussion. A further requirement that applies to all of the type constructors appearing in the right-hand side of the above grammar, including those in  $\overline{K}$ , is that they must all admit predicate liftings. This is not an overly restrictive condition, though: all types constructed from sums, products, arrow types and type application admit predicate liftings, and so do GADTs constructed from the grammar. In fact, we have seen such liftings for products and type application in Section 4. A concrete way to define more general predicate liftings is, again, by induction on the structure of the types. We do not give a general definition of predicate liftings here, though, since that would require us to first design a full type calculus, which is beyond the scope of the present paper.

We assume in the development below that  $G$  is a unary type constructor, i.e., that  $\alpha = 1$  in (11). Extending the argument to GADTs of arbitrary arity presents no difficulty other than heavier notation. In this case the type of  $G$ 's single data constructor  $c$  can be rewritten as  $c : \forall (\overline{B : \text{Set}}) \rightarrow \text{Equal } A (K \overline{B}) \rightarrow F G \overline{B} \rightarrow G A$ . The predicate lifting  $G^\wedge : \forall (\overline{A : \text{Set}}) \rightarrow (\overline{A \rightarrow \text{Set}}) \rightarrow G A \rightarrow \text{Set}$  for  $G$  is therefore

$$G^\wedge A Q_A (c \overline{B} e x) = \exists [\overline{Q_B}] \text{Equal}^\wedge A (K \overline{B}) Q_A (K^\wedge \overline{B} \overline{Q_B}) e \times F^\wedge G \overline{B} G^\wedge \overline{Q_B} x$$

where  $Q_A : A \rightarrow \text{Set}$ ,  $\overline{Q_B} : \overline{B} \rightarrow \overline{\text{Set}}$ ,  $e : \text{Equal } A (K \overline{B})$ , and  $x : F G \overline{B}$ . If we have predicate liftings  $F^\wedge : \forall (G : \text{Set}^\alpha \rightarrow \text{Set}) (\overline{B : \text{Set}}) \rightarrow (\forall (\overline{A : \text{Set}}) \rightarrow (\overline{A \rightarrow \text{Set}}) \rightarrow G A \rightarrow \text{Set}) \rightarrow (\overline{B \rightarrow \text{Set}}) \rightarrow F G \overline{B} \rightarrow \text{Set}$  for  $F$  and  $K^\wedge : \forall (\overline{B : \text{Set}}) \rightarrow (\overline{B \rightarrow \text{Set}}) \rightarrow K \overline{B} \rightarrow \text{Set}$  for  $K$ , then the induction hypothesis  $\text{dIndC}$  associated with  $c$  is

$$\begin{aligned} \text{dIndC} &= \lambda (P : \forall (\overline{A : \text{Set}}) \rightarrow (\overline{A \rightarrow \text{Set}}) \rightarrow G A \rightarrow \text{Set}) \rightarrow \\ &\quad \forall (\overline{A : \text{Set}}) (\overline{B : \text{Set}}) (Q_A : A \rightarrow \text{Set}) (\overline{Q_B} : \overline{B} \rightarrow \overline{\text{Set}}) (e : \text{Equal } A (K \overline{B})) (x : F G \overline{B}) \rightarrow \\ &\quad \text{Equal}^\wedge A (K \overline{B}) Q_A (K^\wedge \overline{B} \overline{Q_B}) e \rightarrow F^\wedge G \overline{B} P \overline{Q_B} x \rightarrow P A Q_A (c \overline{B} e x) \end{aligned}$$

and the induction rule for  $G$  is

$$\forall (P : \forall (\overline{A : \text{Set}}) \rightarrow (\overline{A \rightarrow \text{Set}}) \rightarrow G A \rightarrow \text{Set}) \rightarrow \text{dIndC } P \rightarrow \forall (\overline{A : \text{Set}}) (Q_A : A \rightarrow \text{Set}) (y : G A) \rightarrow G^\wedge A Q_A y \rightarrow P A Q_A y$$

To prove that this rule is sound we define a witness  $\text{dIndG}$  inhabiting this type by

$$\text{dIndG } P \text{ cc } A Q_A (c \overline{B} e x) (\overline{Q_B}, \text{liftE}, \text{liftF}) = \text{cc } A \overline{B} Q_A \overline{Q_B} e x \text{ liftE } (p \times \text{liftF})$$

Here,  $\text{cc} : \text{dIndC } P$ ,  $e : \text{Equal } A (K \overline{B})$ ,  $x : F G \overline{B}$ ,  $Q_A : A \rightarrow \text{Set}$ ,  $\overline{Q_B} : \overline{B} \rightarrow \overline{\text{Set}}$ ,  $\text{liftE} : \text{Equal}^\wedge A (K \overline{B}) Q_A (K^\wedge \overline{B} \overline{Q_B}) e$ , and  $\text{liftF} : F^\wedge G \overline{B} G^\wedge \overline{Q_B} x$ . As a result,  $(\overline{Q_B}, \text{liftE}, \text{liftF}) : G^\wedge A Q_A (c \overline{B} e x)$  as expected. Finally, the morphism of predicates  $p : \text{PredMap } (F G \overline{B}) (F^\wedge G \overline{B} G^\wedge \overline{Q_B}) (F^\wedge G \overline{B} P \overline{Q_B})$  is defined by structural induction on  $F$  as follows:

- If  $F G \overline{B} = F_1 G \overline{B} \times F_2 G \overline{B}$  then  $F^\wedge G \overline{B} P \overline{Q_B} = \text{Pair}^\wedge (F_1 G \overline{B}) (F_2 G \overline{B}) (F_1^\wedge G \overline{B} P \overline{Q_B}) (F_2^\wedge G \overline{B} P \overline{Q_B})$ . The induction hypothesis ensures morphisms of predicates  $p_1 : \text{PredMap } (F_1 G \overline{B}) (F_1^\wedge G \overline{B} G^\wedge \overline{Q_B}) (F_1^\wedge G \overline{B} P \overline{Q_B})$  and  $p_2 : \text{PredMap } (F_2 G \overline{B}) (F_2^\wedge G \overline{B} G^\wedge \overline{Q_B}) (F_2^\wedge G \overline{B} P \overline{Q_B})$ . For  $x_1 : F_1 G \overline{B}$ ,  $\text{liftF}_1 : F_1^\wedge G \overline{B} G^\wedge \overline{Q_B} x_1$ ,  $x_2 : F_2 G \overline{B}$  and  $\text{liftF}_2 : F_2^\wedge G \overline{B} G^\wedge \overline{Q_B} x_2$  we therefore define  $p(x_1, x_2) (\text{liftF}_1, \text{liftF}_2) = (p_1 x_1 \text{ liftF}_1, p_2 x_2 \text{ liftF}_2)$ .
- The case  $F G \overline{B} = F_1 G \overline{B} + F_2 G \overline{B}$  is analogous.
- If  $F G \overline{B} = F_1 \overline{B} \rightarrow F_2 G \overline{B}$  then  $F^\wedge G \overline{B} P \overline{Q_B} x = \forall (z : F_1 \overline{B}) \rightarrow F_1^\wedge \overline{B} \overline{Q_B} z \rightarrow F_2^\wedge G \overline{B} P \overline{Q_B} (x z)$ , where  $x : F G \overline{B}$ . The induction hypothesis ensures a morphism of predicates  $p_2 : \text{PredMap } (F_2 G \overline{B}) (F_2^\wedge G \overline{B} G^\wedge \overline{Q_B}) (F_2^\wedge G \overline{B} P \overline{Q_B})$ . We define  $p \times \text{liftF} : F^\wedge G \overline{B} P \overline{Q_B} x$ , where  $\text{liftF} : F^\wedge G \overline{B} G^\wedge \overline{Q_B} x$ , to be  $p \times \text{liftF } z \text{ liftF}_1 = p_2 (x z) (\text{liftF } z \text{ liftF}_1)$  for  $z : F_1 \overline{B}$  and  $\text{liftF}_1 : F_1^\wedge \overline{B} \overline{Q_B} z$ . Note that  $F_1$  not containing  $G$  is a necessary restriction since the proof relies on  $F^\wedge G \overline{B} G^\wedge \overline{Q_B} x$  and  $F^\wedge G \overline{B} P \overline{Q_B} x$  having the same domain  $F_1^\wedge \overline{B} \overline{Q_B} z$ .

- If  $F G \bar{B} = G(F_1 \bar{B})$  and  $F_1$  does not contain  $G$ , then  $F^\wedge G \bar{B} P \bar{Q}_B = P(F_1 \bar{B})(F_1^\wedge \bar{B} \bar{Q}_B)$  for all  $P : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow GA \rightarrow \text{Set}$ . We therefore define  $p = \text{dIndG P cc}(F_1 \bar{B})(F_1^\wedge \bar{B} \bar{Q}_B)$ .
- If  $F G \bar{B} = H \bar{B}$  and  $H$  does not contain  $G$ , then  $F^\wedge G \bar{B} P \bar{Q}_B = H^\wedge \bar{Q}_B$  for all  $P : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow GA \rightarrow \text{Set}$ . We therefore define  $p : \text{PredMap}(H \bar{B})(H^\wedge \bar{B} \bar{Q}_B)(H^\wedge \bar{B} \bar{Q}_B)$  to be the identity morphism on predicates.
- If  $F G \bar{B} = H(F_k G \bar{B})$  and  $H$  does not contain  $G$ , then  $F^\wedge G \bar{B} P \bar{Q}_B = H^\wedge(F_k G \bar{B})(F_k^\wedge G \bar{B} P \bar{Q}_B)$  for all  $P : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow GA \rightarrow \text{Set}$ . Since  $H$  is not a GADT,  $H^\wedge$  has a map function  $H^\wedge \text{Map}$  as in (12). The induction hypothesis ensures morphisms of predicates  $\bar{p}_k : \text{PredMap}(F_k G \bar{B})(F_k^\wedge G \bar{B} G^\wedge \bar{Q}_B)(F_k^\wedge G \bar{B} P \bar{Q}_B)$ . We therefore define  $p = H^\wedge \text{Map}(F_k G \bar{B})(F_k^\wedge G \bar{B} G^\wedge \bar{Q}_B)(F_k^\wedge G \bar{B} P \bar{Q}_B) \bar{p}_k$ .

## 6 Truly Nested GADTs Need Not Admit Deep Induction Rules

In Sections 4 and 5 we derived deep induction rules for GADTs that are not truly nested. Since both nested types and GADTs without true nesting admit deep induction rules, we might expect truly nested GADTs to admit them as well. Surprisingly, however, they do not. That is, our results from the previous section are the strongest possible. Indeed, the induction rule for a data type generally relies on (unary) parametricity of the model interpreting it, and deep induction for a truly nested type or a truly nested GADT also relies on this interpretation being functorial. But, whereas ADTs and nested types both admit functorial parametric semantics, proper GADTs cannot admit both functorial and parametric semantics at the same time [13]. In this section we show how deep induction for truly nested GADTs goes wrong by analyzing the following very simple nested proper GADT:

data  $G : \text{Set} \rightarrow \text{Set}$  where  
 $c : \forall \{A : \text{Set}\} \rightarrow G(GA) \rightarrow G(A \times A)$

The constructor  $c$  can be rewritten as  $c : \forall (B : \text{Set}) \rightarrow \text{Equal } A (B \times B) \rightarrow G(GB) \rightarrow GA$ , so the predicate lifting  $G^\wedge : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow GA \rightarrow \text{Set}$  for  $G$  is

$$G^\wedge A Q (c B e x) = \exists [Q_A] \text{Equal}^\wedge A (B \times B) Q_A (\text{Pair}^\wedge B B Q_B Q_B) e \times G^\wedge (GB) (G^\wedge B Q_B) x$$

where  $Q_A : A \rightarrow \text{Set}$ ,  $Q_B : B \rightarrow \text{Set}$ ,  $e : \text{Equal } A (B \times B)$ , and  $x : G(GB)$ . The induction hypothesis  $\text{dIndC}$  for  $c$  is

$$\begin{aligned} & \lambda (P : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow GA \rightarrow \text{Set}) \rightarrow \\ & \quad \forall (A B : \text{Set}) (Q_A : A \rightarrow \text{Set}) (Q_B : B \rightarrow \text{Set}) (e : \text{Equal } A (B \times B)) (x : G(GB)) \rightarrow \\ & \quad \text{Equal}^\wedge A (B \times B) Q_A (\text{Pair}^\wedge B B Q_B Q_B) e \rightarrow P(GB) (P B Q_B) x \rightarrow P A Q_A (c B e x) \end{aligned}$$

so the deep induction rule for  $G$  is

$$\forall (P : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow GA \rightarrow \text{Set}) \rightarrow \text{dIndC } P \rightarrow \forall (A : \text{Set}) (Q : A \rightarrow \text{Set}) (y : Ga) \rightarrow G^\wedge A Q y \rightarrow P A Q y$$

But if we now try to show that this rule is sound by constructing a witness  $\text{dIndG}$  inhabiting this type we run into problems. We can define  $\text{dIndG P cc A Q}(c B e x) (Q', \text{liftE}, \text{liftG}) = \text{cc A B Q Q'} e x \text{liftE } p$ , where  $\text{cc} : \text{dIndC } P$ ,  $Q : A \rightarrow \text{Set}$ ,  $Q' : B \rightarrow \text{Set}$ ,  $e : \text{Equal } A (B \times B)$ ,  $x : G(GB)$ ,  $\text{liftG} : G^\wedge (GB) (G^\wedge B Q') x$ , and  $\text{liftE} : \text{Equal}^\wedge A (B \times B) Q (\text{Pair}^\wedge B B Q' Q') e$ , but we still need to define  $p : P(GB) (P B Q') x$ . For this we can use the induction rule and let  $p = \text{dIndG P cc}(GB) (P B Q') x q$ , but we still need to define  $q : G^\wedge (GB) (P B Q') x$ . If we had the map function  $G^\wedge \text{Map} : \forall (A : \text{Set}) (Q Q' : A \rightarrow \text{Set}) \rightarrow \text{PredMap } A Q Q' \rightarrow \text{PredMap}(GA) (G^\wedge A Q) (G^\wedge A Q')$  for  $G^\wedge$ , then we could define  $q = G^\wedge \text{Map}(GB) (G^\wedge B Q') (P B Q') (\text{dIndG P cc B Q'}) x \text{liftG}$ . Unfortunately, however, we cannot define  $G^\wedge \text{Map}$ . Indeed, its definition would have to be  $G^\wedge \text{Map } A Q Q' m (c B e x) (Q_B, \text{liftE}, \text{liftG}) = (Q'_B, \text{liftE}', \text{liftG}')$  for some  $Q'_B : B \rightarrow \text{Set}$ ,  $\text{liftE}' : \text{Equal}^\wedge A (B \times B) Q' (\text{Pair}^\wedge B B Q'_B Q'_B) e$ , and  $\text{liftG}' : G^\wedge (GB) (G^\wedge B Q'_B) x$ , where  $Q : A \rightarrow \text{Set}$ ,  $Q' : A \rightarrow \text{Set}$ ,  $Q_B : B \rightarrow \text{Set}$ ,  $m : \text{PredMap } A Q Q'$ ,  $e : \text{Equal } A (B \times B)$ ,  $x : G(GB)$ ,  $\text{liftE} : \text{Equal}^\wedge A (B \times B) Q (\text{Pair}^\wedge B B Q_B Q_B) e$ , and  $\text{liftG} : G^\wedge (GB) (G^\wedge B Q_B) x$ . In other words, we would need to produce a proof  $\text{liftE}'$  of the (extensional) equality of the predicates  $Q'$  and  $\text{Pair}^\wedge B B Q'_B Q'_B$  from just a proof  $\text{liftE}$  of the (extensional) equality of the predicates  $Q$  and  $\text{Pair}^\wedge B B Q_B Q_B$  and a morphism of predicates  $m$  from  $Q$  to  $Q'_B$ . But this will not be possible in general: the facts that  $Q$  is equal to  $\text{Pair}^\wedge B B Q_B Q_B$  and that there is a morphism of predicates  $m$  from  $Q$  to  $Q'_B$  do not guarantee that there exists a predicate  $Q'_B$  such that  $Q'$  is equal to  $\text{Pair}^\wedge B B Q'_B Q'_B$ .

At a deeper level, the fundamental issue is that the `Equal` type does not have functorial semantics [13], so that having morphisms  $A \rightarrow A'$  and  $B \rightarrow B'$  (for any type  $A, A', B$  and  $B'$ ) and a proof that  $A$  is equal to  $A'$  does not provide a proof that  $B$  is equal to  $B'$ . And not being able to define  $\text{Equal}^\wedge \text{Map} : \forall (A B : \text{Set}) (Q_A Q'_A : A \rightarrow \text{Set}) (Q_B Q'_B : B \rightarrow \text{Set}) \rightarrow \text{PredMap } A Q_A Q'_A \rightarrow \text{PredMap } B Q_B Q'_B \rightarrow \text{PredMap } (\text{Equal}^\wedge A B) Q_A Q_B (\text{Equal}^\wedge A B Q_A Q_B)$  of course makes it impossible to define  $G^\wedge \text{Map}$  for more general  $G$ .

## 7 Case Study: Extracting Types of Lambda Terms

In this section we use deep induction for the `LTerm` GADT from (6) to extract the type from a lambda term. The following predicate either returns the type of its input lambda term if that type can be inferred or indicates that the type inference fails:

$$\begin{aligned} \text{GetType} &: \forall (A : \text{Set}) \rightarrow \text{LTerm } A \rightarrow \text{Set} \\ \text{GetType } A \, t &= \text{Maybe } (\text{LType } A) \end{aligned}$$

The predicate `GetType` uses the standard `Maybe` data type to represent failure of type inference. It is defined by:

$$\begin{aligned} \text{data } \text{Maybe} &: \text{Set} \rightarrow \text{Set} \text{ where} \\ \text{nothing} &: \forall \{A : \text{Set}\} \rightarrow \text{Maybe } A \\ \text{just} &: \forall \{A : \text{Set}\} \rightarrow A \rightarrow \text{Maybe } A \end{aligned} \tag{13}$$

We want to show that `GetType` is satisfied by every element  $t$  in `LTerm`, i.e., we want to prove:

$$\text{getTypeProof} : \forall (A : \text{Set}) (t : \text{LTerm } A) \rightarrow \text{GetType } A \, t$$

This property can be proved with deep induction, which is used to apply the induction hypothesis to the individual terms in the list of terms that the data constructor `list` takes as an argument. Indeed, using the deep induction rule `dIndLTerm` from Section 4.3 we can define `getTypeProof` by

$$\text{getTypeProof } A \, t = \text{dIndLTerm } P \, \text{cvar} \, \text{cabs} \, \text{capp} \, \text{clist} \, A \, K_\top \, t \, (\text{LTerm}^\wedge K_\top A \, t)$$

where  $t : \text{LTerm } A$ ,  $P$  is the polymorphic predicate  $\lambda (A : \text{Set}) (Q : A \rightarrow \text{Set}) (t : \text{LTerm } A) \rightarrow \text{Maybe } (\text{LType } A)$ ,  $K_\top$  is the constantly  $\top$ -valued predicate on  $A$ , and  $\text{LTerm}^\wedge K_\top : \forall (A : \text{Set}) (t : \text{LTerm } A) \rightarrow \text{LTerm}^\wedge A \, K_\top \, t$  is a term, to be defined below, witnessing that  $K_\top$  can be lifted to all terms. We also need the applications to  $P$  of each of the induction hypotheses from Section 4.3. These are:

$$\begin{aligned} \text{cvar} &: \forall (A : \text{Set}) (Q_A : A \rightarrow \text{Set}) (s : \text{String}) (T_A : \text{LType } A) \rightarrow \text{LType}^\wedge A \, Q_A \, T_A \rightarrow \text{Maybe } (\text{LType } A) \\ \text{cabs} &: \forall (A B C : \text{Set}) (Q_A : A \rightarrow \text{Set}) (Q_B : B \rightarrow \text{Set}) (Q_C : C \rightarrow \text{Set}) \\ &\quad (e : \text{Equal } A (B \rightarrow C)) (s : \text{String}) (T_B : \text{LType } B) (t_C : \text{LTerm } C) \rightarrow \\ &\quad \text{Equal}^\wedge A (B \rightarrow C) \, Q_A (\text{Arr}^\wedge B \, C \, Q_B \, Q_C) \, e \rightarrow \text{LType}^\wedge B \, Q_B \, T_B \rightarrow \text{Maybe } (\text{LType } C) \rightarrow \text{Maybe } (\text{LType } A) \\ \text{capp} &: \forall (A B : \text{Set}) (Q_A : A \rightarrow \text{Set}) (Q_B : B \rightarrow \text{Set}) (t_{BA} : \text{LTerm } (B \rightarrow A)) (t_B : \text{LTerm } B) \rightarrow \\ &\quad \text{Maybe } (\text{LType } (B \rightarrow A)) \rightarrow \text{Maybe } (\text{LType } B) \rightarrow \text{Maybe } (\text{LType } A) \\ \text{clist} &: \forall (A B : \text{Set}) (Q_A : A \rightarrow \text{Set}) (Q_B : B \rightarrow \text{Set}) (e : \text{Equal } A (\text{List } B)) (ts : \text{List } (\text{LTerm } B)) \rightarrow \\ &\quad \text{Equal}^\wedge A (\text{List } B) \, Q_A (\text{List}^\wedge B \, Q_B) \, e \rightarrow \text{List}^\wedge (\text{LTerm } B) (\text{GetType } B) \, ts \rightarrow \text{Maybe } (\text{LType } A) \end{aligned}$$

In the first clause, `cvar` returns `just`  $T_A$ . In the second clause, `cabs` returns `nothing` if its final argument is `nothing` and `cabs`  $A B C \, Q_A \, Q_B \, Q_C \, e \, s \, T_B \, t_C \, \text{liftE } \text{lift}_{T_B} (\text{just } T_C) = \text{just } (\text{arr } B \, C \, e \, T_B \, T_C)$  otherwise. In the third clause, `capp`  $A B \, Q_A \, Q_B \, t_{BA} \, t_A (\text{just } (\text{arr } B \, A \, \text{refl } T_B \, T_A)) \, \text{mb} = \text{just } T_A$  and `capp` returns `nothing` otherwise. In the fourth clause, we must use  $\text{List}^\wedge (\text{LTerm } B) (\text{GetType } B) \, ts$  to extract the type of the head of  $ts$  (from which we can deduce the type of the list). When  $ts = \text{nil}$  we define  $\text{clist } A B \, Q \, Q' \, e \, \text{nil} \, \text{liftE } \text{lift}_{ts} = \text{nothing}$ , where  $\text{liftE} : \text{Equal}^\wedge A (\text{List } B) \, Q (\text{List}^\wedge B \, Q') \, e$ , and  $\text{lift}_{ts} : \text{List}^\wedge (\text{LTerm } B) (\text{GetType } B) \, ts$ . When  $ts = \text{const } ts'$  the type of  $\text{list}_{ts}$  becomes  $\text{List}^\wedge (\text{LTerm } B) (\text{GetType } B) (\text{const } ts') = \text{GetType } B \, t \times \text{List}^\wedge (\text{LTerm } B) (\text{GetType } B) \, ts' = \text{Maybe } (\text{LType } B) \times \text{List}^\wedge (\text{LTerm } B) (\text{GetType } B) \, ts'$ . We pattern match on the first component of the pair to define

$$\begin{aligned} \text{clist } A B \, Q \, Q' \, e (\text{const } ts') \, \text{liftE } (\text{nothing}, \text{lift}_{ts'}) &= \text{nothing} \\ \text{clist } A B \, Q \, Q' \, e (\text{const } ts') \, \text{liftE } (\text{just } T', \text{lift}_{ts'}) &= \text{just } (\text{list } B \, e \, T') \end{aligned}$$

Here  $e : \text{Equal } A (\text{List } B)$ ,  $T' : \text{LType } B$ , and  $\text{lift}_{ts'} : \text{List}^\wedge (\text{LTerm } B) (\text{GetType } B) \, ts'$ .

To finish defining `getTypeProof` we still need a proof

$$\text{LTerm}^\wedge \text{KT} : \forall (A : \text{Set}) (t : \text{LTerm } A) \rightarrow \text{LTerm}^\wedge A \text{K}_T t$$

Since  $\text{LTerm}^\wedge$  is defined in terms of  $\text{LType}^\wedge$  and  $\text{Arr}^\wedge$ , and since  $\text{LType}^\wedge$  is also defined in terms of  $\text{List}^\wedge$ , we need analogous functions  $\text{LType}^\wedge \text{KT}$ ,  $\text{Arr}^\wedge \text{KT}$  and  $\text{List}^\wedge \text{KT}$ , respectively, for each of these liftings as well. We only give the definition of  $\text{LTerm}^\wedge \text{KT}$  here since  $\text{LType}^\wedge \text{KT}$ ,  $\text{Arr}^\wedge \text{KT}$ , and  $\text{List}^\wedge \text{KT}$  are defined analogously. We have:

- If  $s : \text{String}$  and  $T : \text{LType } A$  we define  $\text{LTerm}^\wedge \text{KT } A (\text{vars } T) = \text{LType}^\wedge \text{KT } A T$ .
- If  $e : \text{Equal } A (B \rightarrow C)$ ,  $s : \text{String}$ ,  $T : \text{LType } B$ , and  $t' : \text{LTerm } C$  we need to define  $\text{LTerm}^\wedge \text{KT } A (\text{abs } B C e s T t')$  of type

$$\begin{aligned} & \text{LTerm}^\wedge A \text{K}_T (\text{abs } B C e s T t') \\ &= \exists [Q_B][Q_C] \text{Equal}^\wedge A (B \rightarrow C) \text{K}_T (\text{Arr}^\wedge B C Q_B Q_C) e \times \text{LType}^\wedge B Q_B T \times \text{LTerm}^\wedge C Q_C t' \end{aligned}$$

where  $\text{K}_T : A \rightarrow \text{Set}$ ,  $Q_B : B \rightarrow \text{Set}$ , and  $Q_C : C \rightarrow \text{Set}$ . The only reasonable choice is to let both  $Q_B$  and  $Q_C$  be  $\text{K}_T$ , which means we need proofs of  $\text{Equal}^\wedge A (B \rightarrow C) \text{K}_T (\text{Arr}^\wedge B C \text{K}_T \text{K}_T) e$ ,  $\text{LType}^\wedge B \text{K}_T T$  and  $\text{LTerm}^\wedge C \text{K}_T t'$ . We take  $\text{LType}^\wedge \text{KT } B T$  and  $\text{LTerm}^\wedge \text{KT } C t'$  for the latter two proofs. For the former we note that, since we are working with proof-relevant predicates, the lifting  $\text{Arr}^\wedge B C \text{K}_T \text{K}_T$  of  $\text{K}_T$  to arrow types is not identical to  $\text{K}_T$  on arrow types but rather (extensionally) isomorphic. We discuss this issue in more detail at the end of the section, but for now we simply assume a proof  $\text{Equal}^\wedge \text{ArrKT} : \text{Equal}^\wedge A (B \rightarrow C) \text{K}_T (\text{Arr}^\wedge B C \text{K}_T \text{K}_T) e$  and define  $\text{LTerm}^\wedge \text{KT } A (\text{abs } B C e s T t') = (\text{K}_T, \text{K}_T, \text{Equal}^\wedge \text{ArrKT}, \text{LType}^\wedge \text{KT } B T, \text{LTerm}^\wedge \text{KT } C t')$ .

- If  $t_1 : \text{LTerm } (B \rightarrow A)$  and  $t_2 : \text{LTerm } B$  then, by the same reasoning as in the previous case, we need to define  $\text{LTerm}^\wedge \text{KT } A (\text{app } B t_1 t_2) : \text{LTerm}^\wedge (B \rightarrow A) (\text{Arr}^\wedge B A \text{K}_T \text{K}_T) t_1 \times \text{LTerm}^\wedge B \text{K}_T t_2$ . We define the second component of the pair to be  $\text{LTerm}^\wedge \text{KT } B t_2$ . We can define the first component from a proof of  $\text{LTerm}^\wedge (B \rightarrow A) \text{K}_T t_1$  and the function  $\text{LTerm}^\wedge \text{EqualMap} : \forall (A : \text{Set}) (Q Q' : A \rightarrow \text{Set}) \rightarrow \text{Equal}^\wedge A A Q Q' \text{refl} \rightarrow \text{PredMap } (\text{LTerm } A) (\text{LTerm}^\wedge A Q) (\text{LTerm}^\wedge A Q')$  that takes two (extensionally) equal predicates with the same carrier and produces a morphism of predicates between their liftings. The definition of  $\text{LTerm}^\wedge \text{EqualMap}$  is straightforwardly given by pattern matching on the first two arguments to  $\text{PredMap}$  in its return type, using transitivity and symmetry of the type constructor  $\text{Equal}$ , together with the analogously defined functions  $\text{LType}^\wedge \text{EqualMap}$  and  $\text{Arr}^\wedge \text{EqualMap}$  in the cases when the first argument to  $\text{PredMap}$  is constructed using  $\text{var}$  and  $\text{app}$ , respectively. Taking  $\text{L}_{\text{K}_T} : \text{LTerm}^\wedge (B \rightarrow A) \text{K}_T t_1$  to be the proof  $\text{L}_{\text{K}_T} = \text{LTerm}^\wedge \text{KT } (B \rightarrow A) t_1$  and taking  $\text{LTerm}^\wedge \text{Arr} : \text{LTerm}^\wedge (B \rightarrow A) (\text{Arr}^\wedge B A \text{K}_T \text{K}_T) t_1$  to be the proof  $\text{LTerm}^\wedge \text{Arr} = \text{LTerm}^\wedge \text{EqualMap } \text{K}_T (\text{Arr}^\wedge B A \text{K}_T \text{K}_T) \text{Equal}^\wedge \text{ArrKT } t_1 \text{L}_{\text{K}_T}$ , we define  $\text{LTerm}^\wedge \text{KT } A (\text{app } B t_1 t_2) = (\text{K}_T, \text{LTerm}^\wedge \text{Arr}, \text{LTerm}^\wedge \text{KT } B t_2)$ .
- If  $e : \text{Equal } A (\text{List } B)$  and  $ts : \text{List } (\text{LTerm } B)$  then, as above, we need to define  $\text{LTerm}^\wedge \text{KT } A (\text{list } B \text{ets}) : \text{Equal}^\wedge A (\text{List } B) \text{K}_T (\text{List}^\wedge B \text{K}_T) e \times \text{List}^\wedge (\text{LTerm } B) (\text{LTerm}^\wedge B \text{K}_T) ts$ . As in that case we assume a proof  $\text{Equal}^\wedge \text{ListKT} : \text{Equal}^\wedge A (\text{List } B) \text{K}_T (\text{List}^\wedge B \text{K}_T) e$  for the first component. We can define the second component using  $\text{liftListMap}$  from Section 4.3 to map a morphism  $\text{PredMap } (\text{LTerm } B) (\text{K}_T) (\text{LTerm}^\wedge B \text{K}_T)$  of predicates to a morphism  $\text{PredMap } (\text{List } (\text{LTerm } B)) (\text{List}^\wedge (\text{LTerm } B) \text{K}_T) (\text{List}^\wedge (\text{LTerm } B) (\text{LTerm}^\wedge B \text{K}_T))$  of lifted predicates. Taking  $\text{m}_{\text{K}_T} : \text{PredMap } (\text{LTerm } B) (\text{K}_T) (\text{LTerm}^\wedge B \text{K}_T)$  to be the proof  $\text{m}_{\text{K}_T} t' \text{tt} = \text{LTerm}^\wedge \text{KT } B t'$ , where  $t' : \text{LTerm } B$  and  $\text{tt}$  is the single element of  $\text{K}_T t'$ , and taking  $\text{L}_{\text{List}^\wedge \text{LTerm}^\wedge \text{KT}} : \text{List}^\wedge (\text{LTerm } B) (\text{LTerm}^\wedge B \text{K}_T) ts$  to be the proof  $\text{L}_{\text{List}^\wedge \text{LTerm}^\wedge \text{KT}} = \text{liftListMap } (\text{LTerm}^\wedge \text{KT } B) (\text{LTerm}^\wedge B \text{K}_T) \text{m}_{\text{K}_T} ts (\text{List}^\wedge \text{KT } (\text{LTerm } B) ts)$ , we define  $\text{LTerm}^\wedge \text{KT } A (\text{list } B \text{ets}) = (\text{K}_T, \text{Equal}^\wedge \text{ListKT}, \text{L}_{\text{List}^\wedge \text{LTerm}^\wedge \text{KT}})$ .

The above techniques can be used to define a function  $\text{G}^\wedge \text{KT} : \forall (A : \text{Set}) (x : \text{GA}) \rightarrow \text{G}^\wedge A \text{K}_T x$  for an arbitrary GADT  $G$  as defined in Section 5. To provide a proof of  $\text{G}^\wedge A \text{K}_T x$  for every term  $x : \text{GA}$ , we need to know that, if  $G$  has a constructor  $c : H (\overline{FG \overline{B}}) \rightarrow G (\overline{K \overline{B}})$ , then  $H$  cannot construct a GADT so the generalization  $H^\wedge \text{Map}$  of  $\text{listLiftMap}$  in the final bullet point above is guaranteed to exist. We also need to know that the lifting of  $\text{K}_T$  to types constructed by any nested type constructor  $F$  is extensionally equal to  $\text{K}_T$  on the types it constructs. For example, we might need a proof that  $\text{Pair}^\wedge A B \text{K}_T \text{K}_T$  is equal to  $\text{K}_T$  on  $A \times B$ . Given a pair  $(a, b) : A \times B$ , we have that  $\text{Pair}^\wedge A B \text{K}_T \text{K}_T (a, b) = \text{K}_T a \times \text{K}_T b = \top \times \top$ , whereas  $\text{K}_T (a, b) = \top$ . While these types are not equal, they are clearly isomorphic. Similar isomorphisms between  $F^\wedge A \text{K}_T$  and  $\text{K}_T$  hold for all other nested type constructors  $F$  as well. These isomorphisms can either be proved on an as-needed basis or, since  $F^\wedge A \text{K}_T = \text{K}_T$  is the unary analogue of the Identity Extension Lemma, be obtained at the meta-level as a consequence of unary parametricity. At the object level, our Agda code [11] simply postulates each isomorphism needed since an Agda implementation of full parametricity for some relevant calculus is beyond the scope of the present paper.

## 8 Conclusion

This paper extends deep induction to deep GADTs that are not truly nested. It also shows that truly nested GADTs, deep or not, do not admit (deep) induction rules. Our development is implemented in Agda, as is a case study showing how deep induction can prove properties of GADTs that are not provable by structural induction.

## References

- [1] Atkey, R. *Relational parametricity for higher kinds*. Proceedings, Computer Science Logic, pp. 46-61, 2012.
- [2] Bainbridge, E. S., Freyd, P., Scedrov, A., and Scott, P. J. *Functorial polymorphism*. Theoretical Computer Science 70(1), pp. 35-64, 1990.
- [3] Bird, R. and Meertens, L. *Nested datatypes*. Proceedings, Mathematics of Program Construction, pp. 52-67, 1998.
- [4] Cheney, J. and Hinze, R. *First-class phantom types*. CUCIS TR2003-1901, Cornell University, 2003.
- [5] Coquand, T. and Huet, G. *The calculus of constructions*. Information and Computation 76(2/3), 1988.
- [6] Zilberstein, N. CIS194 homepage. <https://www.seas.upenn.edu/~cis194/spring15/lectures/11-stlc.html>
- [7] The Coq Development Team. *The Coq Proof Assistant*, version 8.11.0, January 2020. <https://doi.org/10.5281/zenodo.3744225>
- [8] Fu, P. and Selinger, P. Dependently typed folds for nested data types, 2018. <https://arxiv.org/abs/1806.05230>
- [9] Ghani, N., Johann, P., Nordvall Forsberg, F., Orsanigo, F., and Revell, T. *Bifibrational functorial semantics for parametric polymorphism*. Proceedings, Mathematical Foundations of Program Semantics, pp. 165-181, 2015.
- [10] Hinze, R. *Fun with phantom types*. Proceedings, The Fun of Programming, pp. 245-262, 2003.
- [11] Johann, P., Ghiorzi, E., and Jeffries, D. Accompanying Agda code for this paper. <https://cs.appstate.edu/~johannp/FoSSaCS21Code.html>
- [12] Johann, P., Ghiorzi, E., and Jeffries, D. *Parametricity for primitive nested types*. Proceedings, Foundations of Software Science and Computation Structures, pp. 324-343, 2021.
- [13] Johann, P., Ghiorzi, E., and Jeffries, D. *GADTs, Functoriality, Parametricity: Pick Two*. Submitted, 2021.
- [14] Johann, P. and Polonsky, A. *Higher-kinded data types: Syntax and semantics* Proceedings, Logic in Computer Science, pp. 1-13, 2019.
- [15] Johann, P. and Polonsky, A. *Deep induction: Induction rules for (truly) nested types*. Proceedings, Foundations of Software Science and Computation Structures, pp. 339-358, 2020.
- [16] MacLane, S. *Categories for the Working Mathematician*. Springer, 1971.
- [17] McBride, C. *Dependently Typed Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.
- [18] Peyton Jones, S., Vytiniotis, D., Weirich, S., and Washburn, G. *Simple unification-based type inference for GADTs*. Proceedings, International Conference on Functional Programming, pp. 50-61, 2006.
- [19] Schrijvers, T., Peyton Jones, S. L., Sulzmann, M., and Vytiniotis, D. *Complete and decidable type inference for GADTs*. Proceedings, International Conference on Functional Programming, pp. 341-352, 2009.
- [20] Sheard, T., and Pasalic, E. *Meta-programming with built-in type equality*. Proceedings, Workshop on Logical Frameworks and Meta-languages, pp. 106-124, 2004.
- [21] Tassi, E.: Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. Proceedings, Interactive Theorem Proving, pp. 1-18, 2019.
- [22] Vytiniotis, D., and Weirich, S. *Parametricity, type equality, and higher-order polymorphism*. Journal of Functional Programming 20(2), pp. 175-210, 2010.
- [23] Xi, H., Chen, C. and Chen, G. *Guarded recursive datatype constructors*. Proceedings, Principles of Programming Languages, pp. 224-235, 2003.
- [24] Ullrich, M. *Generating Induction Principles for Nested Induction Types in MetaCoq*. PhD thesis, Saarland University, 2020.