(Deep) Induction Rules for GADTs

Patricia Johann Enrico Ghiorzi Daniel Jeffries

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

Abstract

Deep data types those that are defined in terms of other 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 was recently shown 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 algebraic data type (ADT) is declared in Coq [1], an induction rule is automatically generated for it.

The data types handled by Coq are (possibly mutually inductive) polynomial 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 so-called *deep ADTs* [15], i.e., ADTs that are (possibly mutually inductively) defined in terms of (other) such ADTs. Consider, for example, the following type of rose trees, here coded in Agda and defined in terms of the standard type List of lists (see Section 2):

$$\begin{aligned} \mathsf{data\,Rose} &: \mathsf{Set} \to \mathsf{Set\,where} \\ \mathsf{empty} &: \mathsf{Rose\,A} \\ \mathsf{node} &: \mathsf{A} \to \mathsf{List\,(Rose\,A)} \to \mathsf{Rose\,A} \end{aligned}$$

The induction rule Coq automatically generates for rose trees is

$$\begin{aligned} &\forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{P} : \mathsf{Rose\,A} \to \mathsf{Set}) \to \mathsf{P\,empty} \to \\ &\quad (\forall\,(\mathsf{a} : \mathsf{A})\,(\mathsf{ts} : \mathsf{List\,(Rose\,A)}) \to \mathsf{P\,(node\,a\,ts)}) \to \forall\,(\mathsf{x} : \mathsf{Rose\,A}) \to \mathsf{P\,x} \end{aligned}$$

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 nested types [3], such as Rose above, as well as truly nested types [2], generalized algebraic data types (GADTs) [4,18,20,23], more richly indexed families [5], and deep variants of all of these.

[1] Algebraic data types are called inductive types in Coq.

[2] A truly nested type is a nested type that is defined over itself. The data type Bush in Section 2 provides a concrete example.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

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 be the form of induction most appropriate to nested types (including ADTs) that are defined over, or mutually recursively with, other such types (including, possibly, themselves). Deep induction delivers the following genuinely useful induction rule for rose trees:

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

Here, $\mathsf{List}^{\wedge}$ (resp., $\mathsf{Rose}^{\wedge}$) lifts its predicate argument P (resp., Q) on data of type $\mathsf{Rose}\,\mathsf{A}$ (resp., A) to a predicate on data of type $\mathsf{List}\,(\mathsf{Rose}\,\mathsf{A})$ (resp., $\mathsf{Rose}\,\mathsf{A}$) asserting that P (resp., Q) holds for every element of its list (resp., rose tree) argument.[3] 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 (truly) 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

$$\begin{aligned}&\mathsf{data}\,\mathsf{Seq}\,(\mathsf{A}:\mathsf{Set}):\mathsf{Set}\,\mathsf{where}\\&\qquad\mathsf{const}:\mathsf{A}\to\mathsf{Seq}\,\mathsf{A}\\&\qquad\mathsf{pair}\ \ :\mathsf{Seq}\,\mathsf{A}\to\mathsf{Seq}\,\mathsf{B}\to\mathsf{Seq}\,(\mathsf{A}\times\mathsf{B})\end{aligned}\tag{2}$$

of sequences only constructs sequences of pairs, rather than sequences of arbitrary type, as does const. If all of the constructors for a GADT G return structured instances of G, then some of G's instances might not be inhabited. GADTs therefore have two distinct, but equally natural, semantics: a functorial semantics interpreting them as left Kan extensions [16], and a parametric semantics interpreting them as their Church encodings [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 that are ADTs or other nested types. However, both they and their attendant properties differ greatly for proper GADTs. In fact, the views deriving from the functorial and parametric semantics for proper GADTs are sufficiently distinct that, by contrast with the situation 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. Perhaps surprisingly, 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. As for ADTs and other nested types, the structural induction

[3] Predicate liftings such as $\mathsf{List}^{\wedge}$ and $\mathsf{Rose}^{\wedge}$ 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 $\mathsf{List}\,\mathsf{t}$ or $\mathsf{Rose}\,\mathsf{t}$ simply traverses containers of that type and applies its predicate argument pointwise to the constituent data of type t.

rule for 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 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, and for the simplest — and arguably most important — GADT, namely the equality GADT. (See Section 4.1.) We then derive the deep induction rule 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 Seq 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 applications or 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].

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 in their examples. 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 attempt has yet been made to extend either implementation truly nested types or to proper GADTs. In fact, we know of no work other than that reported here that specifically addresses induction rules for (deep) GADTs.

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 type $\mathsf{A} : \mathsf{Set}$ is a function $\mathsf{P} : \mathsf{A} \to \mathsf{Set}$ mapping each $\mathsf{a} : \mathsf{A}$ to the set of proofs that $\mathsf{P}\,\mathsf{a}$ holds. For example, the structural induction rule for the list type

$$\begin{aligned} &\mathsf{data}\ \mathsf{List} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where} \\ &\quad \mathsf{nil}\ \ : \mathsf{List}\,\mathsf{A} \\ &\quad \mathsf{cons} : \mathsf{A} \to \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{A} \end{aligned}$$

is

$$\forall(\mathsf{A} : \mathsf{Set})(\mathsf{P} : \mathsf{List}\,\mathsf{A} \to \mathsf{Set}) \to \mathsf{P}\,\mathsf{nil} \to \big(\forall(\mathsf{a} : \mathsf{A})(\mathsf{as} : \mathsf{List}\,\mathsf{A}) \to \mathsf{P}\,\mathsf{as} \to \mathsf{P}\,(\mathsf{cons}\,\mathsf{a}\,\mathsf{as})\big) \to \forall(\mathsf{as} : \mathsf{List}\,\mathsf{A}) \to \mathsf{P}\,\mathsf{as}$$

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 $\mathsf{List}^\wedge$ as described in the introduction the deep induction rule for lists is

$$\begin{aligned} &\forall(\mathsf{A} : \mathsf{Set})(\mathsf{P} : \mathsf{List}\,\mathsf{A} \to \mathsf{Set})(\mathsf{Q} : \mathsf{A} \to \mathsf{Set}) \to \mathsf{P}\,\mathsf{nil} \to \big(\forall(\mathsf{a} : \mathsf{A})(\mathsf{as} : \mathsf{List}\,\mathsf{A}) \to \mathsf{Q}\,\mathsf{a} \to \mathsf{P}\,\mathsf{as} \to \mathsf{P}\,(\mathsf{cons}\,\mathsf{a}\,\mathsf{as})\big) \\ &\quad \to \forall(\mathsf{as} : \mathsf{List}\,\mathsf{A}) \to \mathsf{List}^\wedge\,\mathsf{A}\,\mathsf{Q}\,\mathsf{as} \to \mathsf{P}\,\mathsf{as} \end{aligned}$$

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

$$\begin{aligned}&\mathsf{data}\ \mathsf{PTree} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{pleaf}\ \ : \mathsf{A} \to \mathsf{PTree}\,\mathsf{A}\\ &\quad \mathsf{pnode} : \mathsf{PTree}\,(\mathsf{A} \times \mathsf{A}) \to \mathsf{PTree}\,\mathsf{A}\end{aligned}$$

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 $\mathsf{PTree}\,(\mathsf{A} \times \mathsf{A})$ to construct data of type $\mathsf{PTree}\,\mathsf{A}$. 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

$$\begin{aligned}&\forall(\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to \mathsf{PTree}\,\mathsf{A} \to \mathsf{Set}) \to \big(\forall(\mathsf{A} : \mathsf{Set})(\mathsf{a} : \mathsf{A}) \to \mathsf{P}\,\mathsf{A}\,(\mathsf{pleaf}\,\mathsf{a})\big)\\ &\quad \to \big(\forall(\mathsf{A} : \mathsf{Set})(\mathsf{pp} : \mathsf{PTree}\,(\mathsf{A} \times \mathsf{A})) \to \mathsf{P}\,(\mathsf{A} \times \mathsf{A})\,\mathsf{pp} \to \mathsf{P}\,\mathsf{a}\,(\mathsf{pnode}\,\mathsf{pp})\big) \to \forall(\mathsf{A} : \mathsf{Set})(\mathsf{p} : \mathsf{PTree}\,\mathsf{A}) \to \mathsf{P}\,\mathsf{A}\,\mathsf{p}\end{aligned}$$

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

$$\begin{aligned}&\forall(\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{PTree}\,\mathsf{A} \to \mathsf{Set}) \to \big(\forall(\mathsf{A} : \mathsf{Set})(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})(\mathsf{a} : \mathsf{A}) \to \mathsf{Q}\,\mathsf{a} \to \mathsf{P}\,\mathsf{A}\,\mathsf{Q}\,(\mathsf{Pleaf}\,\mathsf{a})\big)\\ &\quad \to \big(\forall(\mathsf{A} : \mathsf{Set})(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})(\mathsf{pp} : \mathsf{PTree}\,(\mathsf{A} \times \mathsf{A})) \to \mathsf{P}\,(\mathsf{A} \times \mathsf{A})\,(\mathsf{Pair}^{\wedge}\,\mathsf{A}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{Q})\,\mathsf{pp} \to \mathsf{P}\,\mathsf{A}\,\mathsf{Q}\,(\mathsf{Pnode}\,\mathsf{pp})\big)\\ &\quad \to \forall(\mathsf{A} : \mathsf{Set})(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})(\mathsf{p} : \mathsf{PTree}\,\mathsf{A}) \to \mathsf{PTree}^{\wedge}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{p} \to \mathsf{P}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{p}\end{aligned}$$

Here, $\mathsf{Pair}^{\wedge} : \forall(\mathsf{A}\,\mathsf{B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to (\mathsf{B} \to \mathsf{Set}) \to \mathsf{A} \times \mathsf{B} \to \mathsf{Set}$ lifts predicates Q_{A} on data of type A and Q_{B} on data of type B to a predicate on pairs of type $\mathsf{A} \times \mathsf{B}$ in such a way that $\mathsf{Pair}^{\wedge}\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}_{\mathsf{A}}\,\mathsf{Q}_{\mathsf{B}}\,(\mathsf{a}, \mathsf{b}) = \mathsf{Q}_{\mathsf{A}}\,\mathsf{a} \times \mathsf{Q}_{\mathsf{B}}\,\mathsf{b}$. Similarly, $\mathsf{PTree}^{\wedge} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{PTree}\,\mathsf{A} \to \mathsf{Set}$ lifts a predicate Q on data of type A to a predicate on data of type $\mathsf{PTree}\,\mathsf{A}$ 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 [3]:

$$\begin{aligned}&\mathsf{data}\ \mathsf{Bush} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{bnil}\ \ \ : \mathsf{Bush}\,\mathsf{A}\\ &\quad \mathsf{bcons} : \mathsf{A} \to \mathsf{Bush}\,(\mathsf{Bush}\,\mathsf{A}) \to \mathsf{Bush}\,\mathsf{A}\end{aligned}$$

Flags to overcome positivity restrictions? 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

$$\begin{aligned}&\forall(\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{Bush}\,\mathsf{A} \to \mathsf{Set}) \to \big(\forall(\mathsf{A} : \mathsf{Set}) \to \mathsf{P}\,\mathsf{A}\,\mathsf{bnil}\big)\\ &\quad \to \big(\forall(\mathsf{A} : \mathsf{Set})(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})(\mathsf{a} : \mathsf{A})(\mathsf{bb} : \mathsf{Bush}\,(\mathsf{Bush}\,\mathsf{A})) \to \mathsf{Q}\,\mathsf{a} \to \mathsf{P}\,(\mathsf{Bush}\,\mathsf{A})\,(\mathsf{Bush}^{\wedge}\,\mathsf{A}\,\mathsf{Q})\,\mathsf{bb} \to \mathsf{P}\,\mathsf{A}\,\mathsf{Q}\,(\mathsf{bcons}\,\mathsf{a}\,\mathsf{bb})\big)\\ &\quad \to \forall(\mathsf{A} : \mathsf{Set})(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})(\mathsf{b} : \mathsf{Bush}\,\mathsf{A}) \to \mathsf{Bush}^{\wedge}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{b} \to \mathsf{P}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{b}\end{aligned}$$

Here, $\mathsf{Bush}^{\wedge} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{Bush}\,\mathsf{A} \to \mathsf{Set}$ is the following lifting of a predicate Q on data of type A to a predicate on data of type $\mathsf{Bush}\,\mathsf{A}$ asserting that Q holds for every element of type A contained in its argument bush:

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

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 exactly coincides with what structural induction should do.

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, every data constructor for PTree A returns an element of type PTree A, 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

$$\begin{aligned} &\mathsf{data}\ \mathsf{Equal} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}\ \mathsf{where} \\ &\qquad \mathsf{refl} : \mathsf{Equal}\,\mathsf{A}\,\mathsf{A} \end{aligned} \tag{3}$$

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, which makes critical use of the equality GADT, as follows:

$$\begin{aligned} &\mathsf{data}\ \mathsf{Seq} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where} \\ &\quad \mathsf{const} : \mathsf{A} \to \mathsf{Seq}\,\mathsf{A} \\ &\quad \mathsf{pair}\ \ : \forall(\mathsf{B}\,\mathsf{C} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{C}) \to \mathsf{Seq}\,\mathsf{B} \to \mathsf{Seq}\,\mathsf{C} \to \mathsf{Seq}\,\mathsf{A} \end{aligned} \tag{4}$$

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 since 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, we keep it to a minimum.

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

$$\begin{aligned} &\mathsf{data}\ \mathsf{LType} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where} \\ &\quad \mathsf{bool} : \forall(\mathsf{B} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,\mathsf{Bool} \to \mathsf{LType}\,\mathsf{A} \\ &\quad \mathsf{arr}\ \ : \forall(\mathsf{B}\,\mathsf{C} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C}) \to \mathsf{LType}\,\mathsf{B} \to \mathsf{LType}\,\mathsf{C} \to \mathsf{LType}\,\mathsf{A} \\ &\quad \mathsf{list}\ \ : \forall(\mathsf{B} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,(\mathsf{List}\ \mathsf{B}) \to \mathsf{LType}\,\mathsf{B} \to \mathsf{LType}\,\mathsf{A} \end{aligned} \tag{5}$$

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

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

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 then 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 $\mathsf{G}\,\mathsf{A}$. Such a predicate lifting $\mathsf{Equal}^\wedge : \forall(\mathsf{A}\,\mathsf{B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to (\mathsf{B} \to \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,\mathsf{B} \to \mathsf{Set}$ for Equal is defined by $\mathsf{Equal}^\wedge\,\mathsf{A}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{Q}'\,\mathsf{refl} = \forall(\mathsf{a} : \mathsf{A}) \to \mathsf{Equal}\,(\mathsf{Q}\,\mathsf{a})(\mathsf{Q}'\,\mathsf{a})$. 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 $\mathsf{dIndRefl}$ is thus associated with the refl constructor for Equal:

$$\begin{aligned}&\lambda(\mathsf{P} : \forall(\mathsf{A}\,\mathsf{B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to (\mathsf{B} \to \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,\mathsf{B} \to \mathsf{Set})\\ &\quad \to \forall(\mathsf{C} : \mathsf{Set})(\mathsf{Q}\,\mathsf{Q}' : \mathsf{C} \to \mathsf{Set}) \to \mathsf{Equal}^\wedge\,\mathsf{C}\,\mathsf{C}\,\mathsf{Q}\,\mathsf{Q}'\,\mathsf{refl} \to \mathsf{P}\,\mathsf{C}\,\mathsf{C}\,\mathsf{Q}\,\mathsf{Q}'\,\mathsf{refl}\end{aligned}$$

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 Equal is thus

$$\begin{aligned}&\forall(\mathsf{P} : \forall(\mathsf{A}\,\mathsf{B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to (\mathsf{B} \to \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,\mathsf{B} \to \mathsf{Set}) \to \mathsf{dIndRefl}\,\mathsf{P} \to\\ &\quad \forall(\mathsf{A}\,\mathsf{B} : \mathsf{Set})(\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set})(\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set})(\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,\mathsf{B}) \to \mathsf{Equal}^\wedge\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}_\mathsf{A}\,\mathsf{Q}_\mathsf{B}\,\mathsf{e} \to \mathsf{P}\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}_\mathsf{A}\,\mathsf{Q}_\mathsf{B}\,\mathsf{e}\end{aligned} \tag{7}$$

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

$$\forall(\mathsf{Q} : \forall(\mathsf{A}\,\mathsf{B} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,\mathsf{B} \to \mathsf{Set}) \to \big(\forall(\mathsf{C} : \mathsf{Set}) \to \mathsf{P}\,\mathsf{C}\,\mathsf{C}\,\mathsf{refl}\big) \to \forall(\mathsf{A}\,\mathsf{B} : \mathsf{Set})(\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,\mathsf{B}) \to \mathsf{P}\,\mathsf{A}\,\mathsf{B}\,\mathsf{e} \tag{8}$$

we define a term $\mathsf{indEqual}$ of the type in (8) by $\mathsf{indEqual}\,\mathsf{Q}\,\mathsf{srefl}\,\mathsf{A}\,\mathsf{B}\,\mathsf{refl} = \mathsf{dIndEqual}\,\mathsf{P}\,\mathsf{srefl}'\,\mathsf{A}\,\mathsf{B}\,\mathsf{K}^\mathsf{A}_\top\,\mathsf{K}^\mathsf{B}_\top\,\mathsf{refl}\,\mathsf{sliftE}$. Here, $\mathsf{P} : \forall(\mathsf{A}\,\mathsf{B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to (\mathsf{B} \to \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,\mathsf{B} \to \mathsf{Set}$ is defined by $\mathsf{P}\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}_\mathsf{A}\,\mathsf{Q}_\mathsf{B}\,\mathsf{e} = \mathsf{Q}\,\mathsf{A}\,\mathsf{B}\,\mathsf{e}$, $\mathsf{K}^\mathsf{A}_\top$ and $\mathsf{K}^\mathsf{B}_\top$ are the constantly $\top$-valued predicates on A and B, respectively, $\mathsf{sliftE} : \mathsf{Equal}^\wedge\,\mathsf{A}\,\mathsf{B}\,\mathsf{K}^\mathsf{A}_\top\,\mathsf{K}^\mathsf{B}_\top\,\mathsf{refl}$ is defined by $\mathsf{sliftE}\,\mathsf{a} = \mathsf{refl} : \mathsf{Equal}\,\top\,\top$ for every $\mathsf{a} : \mathsf{A}$, and $\mathsf{srefl}' : \forall(\mathsf{C} : \mathsf{Set})(\mathsf{Q}_\mathsf{c}\,\mathsf{Q}'_\mathsf{c} : \mathsf{C} \to \mathsf{Set}) \to \mathsf{Equal}^\wedge\,\mathsf{C}\,\mathsf{C}\,\mathsf{Q}_\mathsf{c}\,\mathsf{Q}'_\mathsf{c}\,\mathsf{refl} \to \mathsf{Q}\,\mathsf{C}\,\mathsf{C}\,\mathsf{refl}$ is defined by $\mathsf{srefl}'\,\mathsf{C}\,\mathsf{Q}_\mathsf{c}\,\mathsf{Q}'_\mathsf{c}\,\mathsf{liftE}' = \mathsf{srefl}\,\mathsf{C}$. 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 Seq we use its Henry Ford encoding from (4). We first define its predicate lifting $\mathsf{Seq}^\wedge : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{Seq}\,\mathsf{A} \to \mathsf{Set}$ by

$$\begin{aligned}\mathsf{Seq}^\wedge\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,(\mathsf{const}\,\mathsf{a}) &= \mathsf{Q}_\mathsf{A}\,\mathsf{a}\\ \mathsf{Seq}^\wedge\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,(\mathsf{sPair}\,\mathsf{B}\,\mathsf{C}\,\mathsf{e}\,\mathsf{s}_\mathsf{B}\,\mathsf{s}_\mathsf{C}) &= \exists[\mathsf{Q}_\mathsf{B}]\exists[\mathsf{Q}_\mathsf{C}]\,\mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \times \mathsf{C})\,\mathsf{Q}_\mathsf{A}\,(\mathsf{Q}_\mathsf{B} \times \mathsf{Q}_\mathsf{C})\,\mathsf{e} \times \mathsf{Seq}^\wedge\,\mathsf{B}\,\mathsf{Q}_\mathsf{B}\,\mathsf{s}_\mathsf{B} \times \mathsf{Seq}^\wedge\,\mathsf{C}\,\mathsf{Q}_\mathsf{C}\,\mathsf{s}_\mathsf{C}\end{aligned}$$

Here, $\mathsf{a} : \mathsf{A}$, $\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set}$, $\mathsf{Q}_\mathsf{C} : \mathsf{C} \to \mathsf{Set}$, $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{C})$, $\mathsf{s}_\mathsf{B} : \mathsf{Seq}\,\mathsf{B}$, $\mathsf{s}_\mathsf{C} : \mathsf{Seq}\,\mathsf{C}$, and $\exists[\mathsf{x}]\,\mathsf{F}\,\mathsf{x}$ is syntactic sugar for the type of dependent pairs (x, b) where $\mathsf{x} : \mathsf{A}$ and $\mathsf{b} : \mathsf{F}\,\mathsf{x}$ and $\mathsf{F} : \mathsf{A} \to \mathsf{Set}$.

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

$$\lambda(\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{Seq}\,\mathsf{A} \to \mathsf{Set}) \to \forall(\mathsf{A} : \mathsf{Set})(\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set})(\mathsf{a} : \mathsf{A}) \to \mathsf{Q}_\mathsf{A}\,\mathsf{a} \to \mathsf{P}\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,(\mathsf{const}\,\mathsf{a})$$

$$
\begin{array}{lcl}
\mathsf{LType}^\wedge\, A\, Q_A\, (\mathsf{bool}\, B\, e) & = & \exists[Q_B]\, \mathsf{Equal}^\wedge\, A\, B\, Q_A\, K_\top^{\mathsf{Bool}}\, e \\
\mathsf{LType}^\wedge\, A\, Q_A\, (\mathsf{arr}\, B\, C\, e\, T_B\, T_C) & = & \exists[Q_B]\, \exists[Q_c]\, \mathsf{Equal}^\wedge\, A\, (B \to C)\, Q_A\, (\mathsf{Arr}^\wedge\, B\, C\, Q_B\, Q_C)\, e \times \mathsf{LType}^\wedge\, B\, Q_B\, T_B \times \mathsf{LType}^\wedge\, C\, Q_C\, T_C \\
\mathsf{LType}^\wedge\, A\, Q_A\, (\mathsf{list}\, B\, e\, T_B) & = & \exists[Q_B]\, \mathsf{Equal}^\wedge\, A\, (\mathsf{List}\, B)\, Q_A\, (\mathsf{List}^\wedge\, B\, Q_B)\, e \times \mathsf{LType}^\wedge\, B\, Q_B\, T_B \\
\mathsf{LTerm}^\wedge\, A\, Q_A\, (\mathsf{var}\, s\, T_A) & = & \mathsf{LType}^\wedge\, A\, Q_A\, T_A \\
\mathsf{LTerm}^\wedge\, A\, Q_A\, (\mathsf{abs}\, B\, C\, e\, s\, T_B\, t_C) & = & \exists[Q_B]\, \exists[Q_C]\, \mathsf{Equal}^\wedge\, A\, (B \to C)\, Q_A\, (\mathsf{Arr}^\wedge\, B\, C\, Q_B\, Q_C)\, e \times \mathsf{LType}^\wedge\, B\, Q_B\, T_B \times \mathsf{LTerm}^\wedge\, C\, Q_C\, t_C \\
\mathsf{LTerm}^\wedge\, A\, Q_A\, (\mathsf{app}\, B\, t_{BA}\, t_B) & = & \exists[Q_B]\, \mathsf{LTerm}^\wedge\, (B \to A)\, (\mathsf{Arr}^\wedge\, B\, A\, Q_B\, Q_A)\, t_{BA} \times \mathsf{LTerm}^\wedge\, B\, Q_B\, t_B \\
\mathsf{LTerm}^\wedge\, A\, Q_A\, (\mathsf{list}\, B\, e\, ts) & = & \exists[Q_B]\, \mathsf{Equal}^\wedge\, A\, (\mathsf{List}\, B)\, Q_A\, (\mathsf{List}^\wedge\, B\, Q_B)\, e \times \mathsf{List}^\wedge\, (\mathsf{LTerm}\, B)\, (\mathsf{LTerm}^\wedge\, B\, Q_B)\, ts
\end{array}
$$

Fig. 1. Predicate liftings for LType and LTerm

associated with the constructor const, and let dIndPair be the induction hypothesis

$$
\begin{array}{l}
\lambda(P : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to \mathsf{Seq}\, A \to \mathsf{Set}) \to \\
\quad \forall(A\, B\, C : \mathsf{Set})(Q_A : A \to \mathsf{Set})(Q_B : B \to \mathsf{Set})(Q_C : C \to \mathsf{Set})(s_B : \mathsf{Seq}\, B)(s_C : \mathsf{Seq}\, C)(e : \mathsf{Equal}\, A\, (B \times C)) \to \\
\quad \mathsf{Equal}^\wedge A\, (B \times C)\, Q_A\, (\mathsf{Pair}^\wedge\, B\, C\, Q_B\, Q_C)\, e \to P\, B\, Q_B\, s_B \to P\, C\, Q_C\, s_C \to P A Q_A (\mathsf{pair}\, B\, C\, e\, s_B\, s_C)
\end{array}
$$

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

$$
\begin{array}{l}
\forall(P : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to \mathsf{Seq}\, A \to \mathsf{Set}) \to \mathsf{dIndConst}\, P \to \mathsf{dIndPair}\, P \to \\
\quad \forall(A : \mathsf{Set})(Q_A : A \to \mathsf{Set})(s_A : \mathsf{Seq}\, A) \to \mathsf{Seq}^\wedge\, A\, Q_A\, s_A \to P\, A\, Q_A\, s_A
\end{array} \tag{9}
$$

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

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

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

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 $\mathsf{Arr}^\wedge : \forall(A\, B : \mathsf{Set}) \to (A \to \mathsf{Set}) \to (B \to \mathsf{Set}) \to (A \to B) \to \mathsf{Set}$ for arrow types, since arrow types appear in LType and LTerm. It is given by $\mathsf{Arr}^\wedge\, A\, B\, Q_A\, Q_B\, f = \forall(a : A) \to Q_A\, a \to Q_B\, (f\, a)$. The predicate liftings $\mathsf{LType}^\wedge : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to \mathsf{LType}\, A \to \mathsf{Set}$ for LType and $\mathsf{LTerm}^\wedge : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to \mathsf{LTerm}\, A \to \mathsf{Set}$ for LTerm are defined in Figure 1. There, $s : \mathsf{String}$, $Q_A : A \to \mathsf{Set}$, $Q_B : B \to \mathsf{Set}$, $Q_C : C \to \mathsf{Set}$, $K_\top^{\mathsf{Bool}}$ is the constantly $\top$-valued predicate on Bool, $T_A : \mathsf{LType}\, A$, $T_B : \mathsf{LType}\, B$, $T_C : \mathsf{LType}\, C$, $t_B : \mathsf{LTerm}\, B$, $t_C : \mathsf{LTerm}\, C$, and $t_{BA} : \mathsf{LTerm}\, (B \to A)$. Moreover, $e : \mathsf{Equal}\, A\, \mathsf{Bool}$ in the first clause, $e : \mathsf{Equal}\, A\, (B \to C)$ in the second, $e : \mathsf{Equal}\, A\, (\mathsf{List}\, B)$ in the third, $e : \mathsf{Equal}\, A\, (B \to C)$ in the fifth, and $e : \mathsf{Equal}\, A\, (\mathsf{List}\, B)$, $ts : \mathsf{List}\, (\mathsf{LTermB})$, and $\mathsf{List}^\wedge$ 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,

$$
\begin{array}{l}
\lambda(P : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to \mathsf{LTerm}\, A \to \mathsf{Set}) \to \\
\quad \forall(A : \mathsf{Set})(Q_A : A \to \mathsf{Set})(s : \mathsf{String})(T_A : \mathsf{LType}\, A) \to \mathsf{LType}^\wedge\, A\, Q_A\, T_A \to P\, A\, Q_A\, (\mathsf{var}\, s\, T_A)
\end{array}
$$

$$
\begin{array}{l}
\lambda(P : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to \mathsf{LTerm}\, A \to \mathsf{Set}) \\
\quad \to \forall(A\, B\, C : \mathsf{Set})(Q_A : A \to \mathsf{Set})(Q_B : B \to \mathsf{Set})(Q_C : C \to \mathsf{Set})(e : \mathsf{Equal}\, A\, (B \to C))(s : \mathsf{String}) \\
\quad \to (T_B : \mathsf{LType}\, B) \to (t_C : \mathsf{LTerm}\, C) \to \mathsf{Equal}^\wedge\, A\, (B \to C)\, Q_A\, (\mathsf{Arr}^\wedge\, B\, C\, Q_B\, Q_C)\, e \\
\quad \to \mathsf{LType}^\wedge\, B\, Q_B\, T_B \to P\, C\, Q_C\, t_C \to P\, A\, Q_A\, (\mathsf{abs}\, B\, C\, e\, s\, T_B\, t_C)
\end{array}
$$

$$
\begin{aligned}
&\mathsf{dIndLTerm}\ \mathsf{P}\ \mathsf{cvar}\ \mathsf{cabs}\ \mathsf{capp}\ \mathsf{clist}\ \mathsf{A}\ \mathsf{Q_A}\ (\mathsf{var}\ \mathsf{s}\ \mathsf{T_A})\ \mathsf{liftA} && = \mathsf{cvar}\ \mathsf{A}\ \mathsf{Q_A}\ \mathsf{s}\ \mathsf{T_A}\ \mathsf{liftA} \\
&\mathsf{dIndLTerm}\ \mathsf{P}\ \mathsf{cvar}\ \mathsf{cabs}\ \mathsf{capp}\ \mathsf{clist}\ \mathsf{A}\ \mathsf{Q_A}\ (\mathsf{abs}\ \mathsf{B}\ \mathsf{C}\ \mathsf{e}\ \mathsf{s}\ \mathsf{T_B}\ \mathsf{t_C})\ (\mathsf{Q_B}, \mathsf{Q_C}, \mathsf{liftE}, \mathsf{lift_{T_B}}, \mathsf{lift_{t_C}}) && = \mathsf{cabs}\ \mathsf{A}\ \mathsf{B}\ \mathsf{C}\ \mathsf{Q_A}\ \mathsf{Q_B}\ \mathsf{Q_C}\ \mathsf{e}\ \mathsf{s}\ \mathsf{T_B}\ \mathsf{t_C}\ \mathsf{liftE}\ \mathsf{lift_{T_B}}\ \mathsf{p_C} \\
&\mathsf{dIndLTerm}\ \mathsf{P}\ \mathsf{cvar}\ \mathsf{cabs}\ \mathsf{capp}\ \mathsf{clist}\ \mathsf{A}\ \mathsf{Q_A}\ (\mathsf{app}\ \mathsf{B}\ \mathsf{t_{BA}}\ \mathsf{t_B})\ (\mathsf{Q_B}, \mathsf{list_{t_{BA}}}, \mathsf{list_{t_B}}) && = \mathsf{capp}\ \mathsf{A}\ \mathsf{B}\ \mathsf{Q_A}\ \mathsf{Q_B}\ \mathsf{t_{BA}}\ \mathsf{t_B}\ \mathsf{p_{BA}}\ \mathsf{p_B} \\
&\mathsf{dIndLTerm}\ \mathsf{P}\ \mathsf{cvar}\ \mathsf{cabs}\ \mathsf{capp}\ \mathsf{clistc}\ \mathsf{A}\ \mathsf{Q_A}\ (\mathsf{list}\ \mathsf{B}\ \mathsf{e}\ \mathsf{ts})\ (\mathsf{Q_B}, \mathsf{liftE'}, \mathsf{lift_{List}}) && = \mathsf{clistc}\ \mathsf{A}\ \mathsf{B}\ \mathsf{Q_A}\ \mathsf{Q_B}\ \mathsf{e}\ \mathsf{ts}\ \mathsf{liftE'}\ \mathsf{p_{List}}
\end{aligned}
$$

Fig. 2. dIndLTerm

$$
\begin{aligned}
&\lambda(\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{LTerm}\ \mathsf{A} \to \mathsf{Set}) \\
&\quad \to \forall(\mathsf{A}\ \mathsf{B} : \mathsf{Set})(\mathsf{Q_A} : \mathsf{A} \to \mathsf{Set})(\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set})(\mathsf{t_{BA}} : \mathsf{LTerm}\ (\mathsf{B} \to \mathsf{A}))(\mathsf{t_B} : \mathsf{LTerm}\ \mathsf{B}) \\
&\quad \to \mathsf{P}\ (\mathsf{B} \to \mathsf{A})\ (\mathsf{Arr}^\wedge\ \mathsf{B}\ \mathsf{A}\ \mathsf{Q_B}\ \mathsf{Q_A})\ \mathsf{t_{BA}} \to \mathsf{P}\ \mathsf{B}\ \mathsf{Q_B}\ \mathsf{t_B} \to \mathsf{P}\ \mathsf{A}\ \mathsf{Q_A}\ (\mathsf{app}\ \mathsf{B}\ \mathsf{t_{BA}}\ \mathsf{t_B})
\end{aligned}
$$

$$
\begin{aligned}
&\lambda(\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{LTerm}\ \mathsf{A} \to \mathsf{Set}) \\
&\quad \to \forall(\mathsf{A}\ \mathsf{B} : \mathsf{Set})(\mathsf{Q_A} : \mathsf{A} \to \mathsf{Set})(\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set})(\mathsf{e} : \mathsf{Equal}\ \mathsf{A}\ (\mathsf{List}\ \mathsf{B}))(\mathsf{ts} : \mathsf{List}\ (\mathsf{LTerm}\ \mathsf{B})) \\
&\quad \to \mathsf{Equal}^\wedge\ \mathsf{A}\ (\mathsf{List}\ \mathsf{B})\ \mathsf{Q_A}\ (\mathsf{List}^\wedge\ \mathsf{B}\ \mathsf{Q_B})\ \mathsf{e} \to \mathsf{List}^\wedge\ (\mathsf{LTerm}\ \mathsf{B})(\mathsf{P}\ \mathsf{B}\ \mathsf{Q_B})\ \mathsf{ts} \to \mathsf{P}\ \mathsf{A}\ \mathsf{Q_A}\ (\mathsf{list}\ \mathsf{B}\ \mathsf{e}\ \mathsf{ts})
\end{aligned}
$$

The deep induction rule for LTerm is thus

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

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

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

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

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

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

For brevity and clarity we indicate only one constructor c in (11), even though a GADT can, in general, have any finite number of them, each with a type the same form as c's. In (11), F and K are type constructors with signatures $(\mathsf{Set}^\alpha \to \mathsf{Set}) \to \mathsf{Set}^\beta \to \mathsf{Set}$ and $\mathsf{Set}^\beta \to \mathsf{Set}$, respectively. If T is a type constructor with signature $\mathsf{Set}^\gamma \to \mathsf{Set}$ then we say that T has arity γ. 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 $\overline{\mathsf{K}}$ must therefore be α. In addition, the type constructor F must be constructed inductively according to the following grammar:

$$
\mathsf{F}\ \mathsf{G}\ \overline{\mathsf{B}} \coloneqq \mathsf{F_1}\ \mathsf{G}\ \overline{\mathsf{B}} \times \mathsf{F_2}\ \mathsf{G}\ \overline{\mathsf{B}} \mid \mathsf{F_1}\ \mathsf{G}\ \overline{\mathsf{B}} + \mathsf{F_2}\ \mathsf{G}\ \overline{\mathsf{B}} \mid \mathsf{F_1}\ \overline{\mathsf{B}} \to \mathsf{F_2}\ \mathsf{G}\ \overline{\mathsf{B}} \mid \mathsf{G}\ (\overline{\mathsf{F_1}\ \overline{\mathsf{B}}}) \mid \mathsf{H}\ \overline{\mathsf{B}} \mid \mathsf{H}\ (\overline{\mathsf{F_1}\ \mathsf{G}\ \overline{\mathsf{B}}})
$$

This grammar is subject to the following restrictions. In the third clause the type constructor $\mathsf{F_1}$ does not contain G. In the fourth clause, none of the α-many type constructors in $\overline{\mathsf{F_1}}$ 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 $\mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}}$ is a closed type or one of the $\mathsf{B_i}$. In the sixth clause, $\mathsf{H} : \mathsf{Set}^\gamma \to \mathsf{Set}$ does not contain G (although $\overline{\mathsf{F_1}\,\mathsf{G}\,\overline{\mathsf{B}}}$ 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

$$\mathsf{H}^\wedge\mathsf{Map} : \forall(\overline{\mathsf{A} : \mathsf{Set}})(\overline{\mathsf{Q}\,\mathsf{Q}' : \mathsf{A} \to \mathsf{Set}}) \to \overline{\mathsf{PredMap}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{Q}'} \to \mathsf{PredMap}\,(\mathsf{H}\,\overline{\mathsf{A}})\,(\mathsf{H}^\wedge\,\overline{\mathsf{A}}\,\overline{\mathsf{Q}})\,(\mathsf{H}^\wedge\,\overline{\mathsf{A}}\,\overline{\mathsf{Q}'}) \tag{12}$$

for $\mathsf{H}^\wedge$. A concrete way to define $\mathsf{H}^\wedge\mathsf{Map}$ 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 sides of (11), including those in $\overline{\mathsf{K}}$, 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 above 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. Awkward paragraph? Rephrase?

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 $\mathsf{c} : \forall(\overline{\mathsf{B} : \mathsf{Set}}) \to \mathsf{Equal}\,\mathsf{A}\,(\mathsf{K}\,\overline{\mathsf{B}}) \to \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}} \to \mathsf{G}\,\mathsf{A}$. The predicate lifting $\mathsf{G}^\wedge : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{G}\,\mathsf{A} \to \mathsf{Set}$ for G is therefore

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

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

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

and the induction rule for G is

$$\forall(\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{G}\,\mathsf{A} \to \mathsf{Set}) \to \mathsf{dIndC}\,\mathsf{P} \to \forall(\mathsf{A} : \mathsf{Set})(\mathsf{Q_A} : \mathsf{A} \to \mathsf{Set})(\mathsf{y} : \mathsf{G}\,\mathsf{A}) \to \mathsf{G}^\wedge\,\mathsf{A}\,\mathsf{Q_A}\,\mathsf{y} \to \mathsf{P}\,\mathsf{A}\,\mathsf{Q_A}\,\mathsf{y}$$

To prove that this rule is sound we define a witness dIndG inhabiting this type by

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

Here, $\mathsf{cc} : \mathsf{dIndC}\,\mathsf{P}$, $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{K}\,\overline{\mathsf{B}})$, $\mathsf{x} : \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}}$, $\mathsf{Q_A} : \mathsf{A} \to \mathsf{Set}$, $\overline{\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}}$, $\mathsf{liftE} : \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{K}\,\overline{\mathsf{B}})\,\mathsf{Q_A}\,(\mathsf{K}^\wedge\,\overline{\mathsf{B}}\,\overline{\mathsf{Q_B}})\,\mathsf{e}$, and $\mathsf{liftF} : \mathsf{F}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q_B}}\,\mathsf{x}$. As a result, $(\overline{\mathsf{Q_B}}, \mathsf{liftE}, \mathsf{liftF}) : \mathsf{G}^\wedge\,\mathsf{A}\,\mathsf{Q_A}(\mathsf{c}\,\overline{\mathsf{B}}\,\mathsf{e}\,\mathsf{x})$ as expected. 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:

- If $\mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}} = \mathsf{F_1}\,\mathsf{G}\,\overline{\mathsf{B}} \times \mathsf{F_2}\,\mathsf{G}\,\overline{\mathsf{B}}$ then $\mathsf{F}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q_B}} = \mathsf{Pair}^\wedge\,(\mathsf{F_1}\,\mathsf{G}\,\overline{\mathsf{B}})\,(\mathsf{F_2}\,\mathsf{G}\,\overline{\mathsf{B}})\,(\mathsf{F_1^\wedge}\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q_B}})\,(\mathsf{F_2^\wedge}\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q_B}})$. The induction hypothesis ensures morphisms of predicates $\mathsf{p_1} : \mathsf{PredMap}\,(\mathsf{F_1}\,\mathsf{G}\,\overline{\mathsf{B}})\,(\mathsf{F_1^\wedge}\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q_B}\mathsf{Q}})(\mathsf{F_1^\wedge}\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q_B}})$ and $\mathsf{p_2} : \mathsf{PredMap}\,(\mathsf{F_2}\,\mathsf{G}\,\overline{\mathsf{B}})\,(\mathsf{F_2^\wedge}\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q_B}})(\mathsf{F_2^\wedge}\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q_B}})$. For $\mathsf{x_1} : \mathsf{F_1}\,\mathsf{G}\,\overline{\mathsf{B}}$, $\mathsf{liftF_1} : \mathsf{F_1^\wedge}\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q_B}}\,\mathsf{x_1}$, $\mathsf{x_2} : \mathsf{F_2}\,\mathsf{G}\,\overline{\mathsf{B}}$ and $\mathsf{liftF_2} : \mathsf{F_2^\wedge}\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q_B}}\,\mathsf{x_2}$ we therefore define $\mathsf{p}\,(\mathsf{x_1}, \mathsf{x_2})\,(\mathsf{liftF_1}, \mathsf{liftF_2}) = (\mathsf{p_1}\,\mathsf{x_1}\,\mathsf{liftF_1},\ \mathsf{p_2}\,\mathsf{x_2}\,\mathsf{liftF_2})$.
- The case $\mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}} = \mathsf{F_1}\,\mathsf{G}\,\overline{\mathsf{B}} + \mathsf{F_2}\,\mathsf{G}\,\overline{\mathsf{B}}$ is analogous.
- If $\mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}} = \mathsf{F_1}\,\overline{\mathsf{B}} \to \mathsf{F_2}\,\mathsf{G}\,\overline{\mathsf{B}}$ then $\mathsf{F}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q_B}}\,\mathsf{x} = \forall(\mathsf{z} : \mathsf{F_1}\,\overline{\mathsf{B}}) \to \mathsf{F_1^\wedge}\,\overline{\mathsf{B}}\,\overline{\mathsf{Q_B}}\,\mathsf{z} \to \mathsf{F_2^\wedge}\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q_B}}\,(\mathsf{x}\,\mathsf{z})$, where $\mathsf{x} : \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}}$. The induction hypothesis ensures a morphism of predicates $\mathsf{p_2} : \mathsf{PredMap}\,(\mathsf{F_2}\,\mathsf{G}\,\overline{\mathsf{B}})\,(\mathsf{F_2^\wedge}\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q_B}})\,(\mathsf{F_2^\wedge}\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q_B}})$. We define $\mathsf{p}\,\mathsf{x}\,\mathsf{liftF} : \mathsf{F}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q_B}}\,\mathsf{x}$, where $\mathsf{liftF} : \mathsf{F}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q_B}}\,\mathsf{x}$, to be $\mathsf{p}\,\mathsf{x}\,\mathsf{liftF}\,\mathsf{z}\,\mathsf{liftF_1} = \mathsf{p_2}\,(\mathsf{x}\,\mathsf{z})\,(\mathsf{liftF}\,\mathsf{z}\,\mathsf{liftF_1})$ for $\mathsf{z} : \mathsf{F_1}\,\overline{\mathsf{B}}$ and $\mathsf{liftF_1} : \mathsf{F_1^\wedge}\,\overline{\mathsf{B}}\,\overline{\mathsf{Q_B}}\,\mathsf{z}$. Note that $\mathsf{F_1}$ not containing G is a necessary restriction, since 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 $\mathsf{F_1^\wedge}\,\overline{\mathsf{B}}\,\overline{\mathsf{Q_B}}\,\mathsf{z}$.

- If $\mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}} = \mathsf{G}\,(\mathsf{F}_1\,\overline{\mathsf{B}})$ and F_1 does not contain G, then $\mathsf{F}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q}_\mathsf{B}} = \mathsf{P}\,(\mathsf{F}_1\,\overline{\mathsf{B}})\,(\mathsf{F}_1^\wedge\,\overline{\mathsf{B}}\,\overline{\mathsf{Q}_\mathsf{B}})$ for all $\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{G}\,\mathsf{A} \to \mathsf{Set}$. We therefore define $\mathsf{p} = \mathsf{dIndG}\,\mathsf{P}\,\mathsf{cc}\,(\mathsf{F}_1\,\overline{\mathsf{B}})\,(\mathsf{F}_1^\wedge\,\overline{\mathsf{B}}\,\overline{\mathsf{Q}_\mathsf{B}})$.
- If $\mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}} = \mathsf{H}\,\overline{\mathsf{B}}$ and H does not contain G, then $\mathsf{F}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q}_\mathsf{B}} = \mathsf{H}^\wedge\,\overline{\mathsf{Q}_\mathsf{B}}$ for all $\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{G}\,\mathsf{A} \to \mathsf{Set}$. We therefore define $\mathsf{p} : \mathsf{PredMap}\,(\mathsf{H}\,\overline{\mathsf{B}})\,(\mathsf{H}^\wedge\,\overline{\mathsf{B}}\,\overline{\mathsf{Q}_\mathsf{B}})\,(\mathsf{H}^\wedge\,\overline{\mathsf{B}}\,\overline{\mathsf{Q}_\mathsf{B}})$ to be the identity morphism on predicates.
- If $\mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}} = \mathsf{H}\,(\overline{\mathsf{F}_\mathsf{k}\,\mathsf{G}\,\overline{\mathsf{B}}})$ and H does not contain G, then $\mathsf{F}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q}_\mathsf{B}} = \mathsf{H}^\wedge\,(\overline{\mathsf{F}_\mathsf{k}\,\mathsf{G}\,\overline{\mathsf{B}}})\,(\overline{\mathsf{F}_\mathsf{k}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q}_\mathsf{B}}})$ for all $\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{G}\,\mathsf{A} \to \mathsf{Set}$. Since H is not a GADT, $\mathsf{H}^\wedge$ has a map function $\mathsf{H}^\wedge\mathsf{Map}$ as in (12). The induction hypothesis ensures morphisms of predicates $\overline{\mathsf{p}_\mathsf{k} : \mathsf{PredMap}\,(\mathsf{F}_\mathsf{k}\,\mathsf{G}\,\overline{\mathsf{B}})(\mathsf{F}_\mathsf{k}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q}_\mathsf{B}})(\mathsf{F}_\mathsf{k}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q}_\mathsf{B}})}$. We therefore define $\mathsf{p} = \mathsf{H}^\wedge\mathsf{Map}\,(\overline{\mathsf{F}_\mathsf{k}\,\mathsf{G}\,\overline{\mathsf{B}}})\,(\overline{\mathsf{F}_\mathsf{k}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q}_\mathsf{B}}})\,(\overline{\mathsf{F}_\mathsf{k}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q}_\mathsf{B}}})\,\overline{\mathsf{p}_\mathsf{k}}$.

6 Truly Nested GADTs Do 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 nesting goes wrong by analyzing the following very simple nested proper GADT:

$$\begin{aligned}&\mathsf{data}\ \mathsf{G} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{c} : \mathsf{G}\,(\mathsf{G}\,\mathsf{A}) \to \mathsf{G}\,(\mathsf{A} \times \mathsf{A})\end{aligned}$$

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

$$\mathsf{G}^\wedge\,\mathsf{A}\,\mathsf{Q}\,(\mathsf{c}\,\mathsf{B}\,\mathsf{e}\,\mathsf{x}) = \exists\,[\mathsf{Q}_\mathsf{A}]\,\mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B})\,\mathsf{Q}_\mathsf{A}\,(\mathsf{Pair}^\wedge\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q}_\mathsf{B}\,\mathsf{Q}_\mathsf{B})\,\mathsf{e} \times \mathsf{G}^\wedge\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{G}^\wedge\,\mathsf{B}\,\mathsf{Q}_\mathsf{B})\,\mathsf{x}$$

where $\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set}$, $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B})$, and $\mathsf{x} : \mathsf{G}\,(\mathsf{G}\,\mathsf{B})$. The induction hypothesis dIndC for c is

$$\begin{aligned}&\lambda\,(\mathsf{P} : \forall\,(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{G}\,\mathsf{A} \to \mathsf{Set})\\ &\quad \to \forall\,(\mathsf{A}\,\mathsf{B} : \mathsf{Set})\,(\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set})\,(\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B}))\,(\mathsf{x} : \mathsf{G}\,(\mathsf{G}\,\mathsf{B}))\\ &\quad \to \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B})\,\mathsf{Q}_\mathsf{A}\,(\mathsf{Pair}^\wedge\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q}_\mathsf{B}\,\mathsf{Q}_\mathsf{B})\,\mathsf{e} \to \mathsf{P}\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{P}\,\mathsf{B}\,\mathsf{Q}_\mathsf{B})\,\mathsf{x} \to \mathsf{P}\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,(\mathsf{c}\,\mathsf{B}\,\mathsf{e}\,\mathsf{x})\end{aligned}$$

so the deep induction rule for G is

$$\forall\,(\mathsf{P} : \forall\,(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{G}\,\mathsf{A} \to \mathsf{Set}) \to \mathsf{dIndC}\,\mathsf{P} \to \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{y} : \mathsf{G}\,\mathsf{a}) \to \mathsf{G}^\wedge\,\mathsf{A}\,\mathsf{Q}\,\mathsf{y} \to \mathsf{P}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{y}$$

But if we now try to show that this rule is sound by constructing a witness dIndG inhabiting this type we run into problems. We can define $\mathsf{dIndG}\,\mathsf{P}\,\mathsf{cc}\,\mathsf{A}\,\mathsf{Q}\,(\mathsf{c}\,\mathsf{B}\,\mathsf{e}\,\mathsf{x})\,(\mathsf{Q}', \mathsf{liftE}, \mathsf{liftG}) = \mathsf{cc}\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}\,\mathsf{Q}'\,\mathsf{e}\,\mathsf{x}\,\mathsf{liftE}\,\mathsf{p}$, where $\mathsf{cc} : \mathsf{dIndC}\,\mathsf{P}$, $\mathsf{Q} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{Q}' : \mathsf{B} \to \mathsf{Set}$, $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B})$, $\mathsf{x} : \mathsf{G}\,(\mathsf{G}\,\mathsf{B})$, $\mathsf{liftG} : \mathsf{G}^\wedge\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{G}^\wedge\,\mathsf{B}\,\mathsf{Q}')\,\mathsf{x}$, and $\mathsf{liftE} : \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B})\,\mathsf{Q}\,(\mathsf{Pair}^\wedge\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q}'\,\mathsf{Q}')\,\mathsf{e}$, but we still need to define $\mathsf{p} : \mathsf{P}\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{P}\,\mathsf{B}\,\mathsf{Q}')\,\mathsf{x}$. For this we can use the induction rule and let $\mathsf{p} = \mathsf{dIndG}\,\mathsf{P}\,\mathsf{cc}\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{P}\,\mathsf{B}\,\mathsf{Q}')\,\mathsf{x}\,\mathsf{q}$, but we still need to define $\mathsf{q} : \mathsf{G}^\wedge\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{P}\,\mathsf{B}\,\mathsf{Q}')\,\mathsf{x}$. If we had the map function $\mathsf{G}^\wedge\mathsf{Map} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{Q}\,\mathsf{Q}' : \mathsf{A} \to \mathsf{Set}) \to \mathsf{PredMap}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{Q}' \to \mathsf{PredMap}\,(\mathsf{G}\,\mathsf{A})\,(\mathsf{G}^\wedge\,\mathsf{A}\,\mathsf{Q})\,(\mathsf{G}^\wedge\,\mathsf{A}\,\mathsf{Q}')$ for $\mathsf{G}^\wedge$, then we could define $\mathsf{q} = \mathsf{G}^\wedge\mathsf{Map}\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{G}^\wedge\,\mathsf{B}\,\mathsf{Q}')\,(\mathsf{P}\,\mathsf{B}\,\mathsf{Q}')\,(\mathsf{dIndG}\,\mathsf{P}\,\mathsf{cc}\,\mathsf{B}\,\mathsf{Q}')\,\mathsf{x}\,\mathsf{liftG}$. Unfortunately, however, we cannot define $\mathsf{G}^\wedge\mathsf{Map}$. Indeed, its definition would have to be $\mathsf{G}^\wedge\mathsf{Map}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{Q}'\,\mathsf{m}\,(\mathsf{c}\,\mathsf{B}\,\mathsf{e}\,\mathsf{x})\,(\mathsf{Q}_\mathsf{B}, \mathsf{liftE}, \mathsf{liftG}) = (\mathsf{Q}'_\mathsf{B}, \mathsf{liftE}', \mathsf{liftG}')$ for some $\mathsf{Q}'_\mathsf{B} : \mathsf{B} \to \mathsf{Set}$, $\mathsf{liftE}' : \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B})\,\mathsf{Q}'\,(\mathsf{Pair}^\wedge\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q}'_\mathsf{B}\,\mathsf{Q}'_\mathsf{B})\,\mathsf{e}$, and $\mathsf{liftG}' : \mathsf{G}^\wedge\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{G}^\wedge\,\mathsf{B}\,\mathsf{Q}'_\mathsf{B})\,\mathsf{x}$, where $\mathsf{Q} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{Q}' : \mathsf{A} \to \mathsf{Set}$, $\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set}$, $\mathsf{m} : \mathsf{PredMap}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{Q}'$, $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B})$, $\mathsf{x} : \mathsf{G}\,(\mathsf{G}\,\mathsf{B})$, $\mathsf{liftE} : \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B})\,\mathsf{Q}\,(\mathsf{Pair}^\wedge\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q}_\mathsf{B}\,\mathsf{Q}_\mathsf{B})\,\mathsf{e}$, and $\mathsf{liftG} : \mathsf{G}^\wedge\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{G}^\wedge\,\mathsf{B}\,\mathsf{Q}_\mathsf{B})\,\mathsf{x}$. In other words, we would need to produce a proof liftE' of the (extensional) equality of the predicates Q' and $\mathsf{Pair}^\wedge\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q}'_\mathsf{B}\,\mathsf{Q}'_\mathsf{B}$ from just a proof liftE of the (extensional) equality of the predicates Q and $\mathsf{Pair}^\wedge\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q}_\mathsf{B}\,\mathsf{Q}_\mathsf{B}$ and a morphism of predicates m from Q to Q'_2. But this will not be possible in general: the facts that Q is equal to $\mathsf{Pair}^\wedge\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q}_\mathsf{B}\,\mathsf{Q}_\mathsf{B}$ and that there is a morphism of predicates m from Q to Q' do not guarantee that there exists a predicate Q'_B such that Q' is equal to $\mathsf{Pair}^\wedge\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q}'_\mathsf{B}\,\mathsf{Q}'_\mathsf{B}$.

At a deeper level, the fundamental issue is that the Equal type does not have functorial semantics [13], so that having morphisms $\mathsf{A} \to \mathsf{A}'$ and $\mathsf{B} \to \mathsf{B}'$ (for any type $\mathsf{A}, \mathsf{A}', \mathsf{B}$ and B') and a proof that A is equal to A′ does not provide a proof that B is equal to B′. This difficulty propagates to any truly nested GADT because it is not possible to define a map function Equal^Map for Eq^ when one of the type arguments to Equal^ is structured (i.e., not a variable). And this is, of course, a key component in any approach to deep induction for truly nested GADTs.

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 takes a lambda term as input and either returns its type if it is well-typed or indicates that it is not:

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

The predicate GetType uses the standard Maybe data type to represent potential ill-typedness. It is defined by:

$$\begin{aligned}&\mathsf{data}\ \mathsf{Maybe} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where}\\&\quad \mathsf{nothing} : \mathsf{Maybe}\,\mathsf{A}\\&\quad \mathsf{just}\quad\ : \mathsf{A} \to \mathsf{Maybe}\,\mathsf{A}\end{aligned} \tag{13}$$

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

$$\mathsf{getTypeProof} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{t} : \mathsf{LTerm}\,\mathsf{A}) \to \mathsf{GetType}\,\mathsf{A}\,\mathsf{t}$$

This property cannot be proved without deep induction, which is needed to apply the induction hypothesis to the individual terms in the list of terms that the data constructor list takes as an argument. But using the deep induction rule dIndLTerm from Section 4.3 we can define getTypeProof by

$$\mathsf{getTypeProof}\,\mathsf{A}\,\mathsf{t} = \mathsf{dIndLTerm}\,\mathsf{P}\,\mathsf{cvar}\,\mathsf{cabs}\,\mathsf{capp}\,\mathsf{clistc}\,\mathsf{A}\,\mathsf{K}_\top\,\mathsf{t}\,(\mathsf{LTerm}^\wedge\mathsf{KT}\,\mathsf{A}\,\mathsf{t})$$

where $\mathsf{t} : \mathsf{LTerm}\,\mathsf{A}$, P is the polymorphic predicate $\lambda\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{t} : \mathsf{LTerm}\,\mathsf{A}) \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{A})$, $\mathsf{K}_\top$ is the constantly $\top$-valued predicate on A, and $\mathsf{LTerm}^\wedge\mathsf{KT} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{t} : \mathsf{LTerm}\mathsf{A}) \to \mathsf{LTerm}^\wedge\,\mathsf{A}\,\mathsf{K}_\top\,\mathsf{t}$ is a term, to be defined below, witnessing that $\mathsf{K}_\top$ 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}
\mathsf{cvar}\ &: \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{s} : \mathsf{String})\,(\mathsf{T}_\mathsf{A} : \mathsf{LType}\,\mathsf{A}) \to \mathsf{LType}^\wedge\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,\mathsf{T}_\mathsf{A} \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{A})\\
\mathsf{cabs}\ &: \forall\,(\mathsf{A}\,\mathsf{B}\,\mathsf{C} : \mathsf{Set})\,(\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set})\,(\mathsf{Q}_\mathsf{C} : \mathsf{C} \to \mathsf{Set})\\
&\quad (\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C}))\,(\mathsf{s} : \mathsf{String})\,(\mathsf{T}_\mathsf{B} : \mathsf{LType}\,\mathsf{B})\,(\mathsf{t}_\mathsf{C} : \mathsf{LTerm}\,\mathsf{C})\\
&\quad\ \to \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C})\,\mathsf{Q}_\mathsf{A}\,(\mathsf{Arr}^\wedge\,\mathsf{B}\,\mathsf{C}\,\mathsf{Q}_\mathsf{B}\,\mathsf{Q}_\mathsf{C})\,\mathsf{e} \to \mathsf{LType}^\wedge\,\mathsf{B}\,\mathsf{Q}_\mathsf{B}\,\mathsf{T}_\mathsf{B} \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{C}) \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{A})\\
\mathsf{capp}\ &: \forall\,(\mathsf{A}\,\mathsf{B} : \mathsf{Set})\,(\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set})\,(\mathsf{t}_{\mathsf{BA}} : \mathsf{LTerm}\,(\mathsf{B} \to \mathsf{A}))\,(\mathsf{t}_\mathsf{B} : \mathsf{LTerm}\,\mathsf{B})\\
&\quad \to \mathsf{Maybe}\,(\mathsf{LType}\,(\mathsf{B} \to \mathsf{A})) \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{B}) \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{A})\\
\mathsf{clist}\ &: \forall\,(\mathsf{A}\,\mathsf{B} : \mathsf{Set})\,(\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set})\,(\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{List}\,\mathsf{B}))\,(\mathsf{ts} : \mathsf{List}\,(\mathsf{LTerm}\,\mathsf{B}))\\
&\quad \to \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{List}\,\mathsf{B})\,\mathsf{Q}_\mathsf{A}\,(\mathsf{List}^\wedge\,\mathsf{B}\,\mathsf{Q}_\mathsf{B})\,\mathsf{e} \to \mathsf{List}^\wedge\,(\mathsf{LTerm}\,\mathsf{B})\,(\mathsf{GetType}\,\mathsf{B})\,\mathsf{ts} \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{A})
\end{aligned}$$

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

$$\begin{aligned}&\mathsf{clist}\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}\,\mathsf{Q}'\,\mathsf{e}\,(\mathsf{cons}\,\mathsf{t}\,\mathsf{ts}')\,\mathsf{liftE}\,(\mathsf{nothing}, \mathsf{lift}_{\mathsf{ts}'}) = \mathsf{nothing}\\&\mathsf{clist}\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}\,\mathsf{Q}'\,\mathsf{e}\,(\mathsf{cons}\,\mathsf{t}\,\mathsf{ts}')\,\mathsf{liftE}\,(\mathsf{just}\,\mathsf{T}', \mathsf{lift}_{\mathsf{ts}'}) = \mathsf{just}\,(\mathsf{TList}\,\mathsf{B}\,\mathsf{e}\,\mathsf{T}')\end{aligned}$$

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

To finish defining $\mathsf{getTypeProof}$ we still need a proof

$$\mathsf{LTerm^\wedge KT} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{t} : \mathsf{LTerm\,A}) \to \mathsf{LTerm^\wedge\,A\,K_\top\,t}$$

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

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

$$\begin{aligned}&\mathsf{LTerm^\wedge\,A\,K_\top\,(abs\,B\,C\,e\,s\,T\,t')}\\ &\quad = \exists[\mathsf{Q_B}]\,[\mathsf{Q_C}]\,\mathsf{Equal^\wedge\,A\,(B \to C)\,K_\top\,(Arr^\wedge\,B\,C\,Q_B\,Q_C)\,e} \times \mathsf{LType^\wedge\,B\,Q_B\,T} \times \mathsf{LTerm^\wedge\,C\,Q_C\,t'}\end{aligned}$$

  where $\mathsf{K_\top} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}$, and $\mathsf{Q_C} : \mathsf{C} \to \mathsf{Set}$. The only reasonable choice is to let both $\mathsf{Q_B}$ and $\mathsf{Q_C}$ be $\mathsf{K_\top}$, which means we need proofs of $\mathsf{Equal^\wedge\,A\,(B \to C)\,K_\top\,(Arr^\wedge\,B\,C\,K_\top\,K_\top)\,e}$, $\mathsf{LType^\wedge\,B\,K_\top\,T}$ and $\mathsf{LTerm^\wedge\,C\,K_\top\,t'}$. We take $\mathsf{LType^\wedge KT\,B\,T}$ and $\mathsf{LTerm^\wedge KT\,C\,t'}$) for the latter two proofs. For the former we note that, since we are working with proof-relevant predicates, the lifting $\mathsf{Arr^\wedge\,B\,C\,K_\top\,K_\top}$ of $\mathsf{K_\top}$ to arrow types is not identical to $\mathsf{K_\top}$ 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 $\mathsf{Equal^\wedge ArrKT} : \mathsf{Equal^\wedge\,A\,(B \to C)\,K_\top\,(Arr^\wedge\,B\,C\,K_\top\,K_\top)\,e}$ and define $\mathsf{LTerm^\wedge KT\,A\,(abs\,B\,C\,e\,s\,T\,t')} = (\mathsf{K_\top}, \mathsf{K_\top}, \mathsf{Equal^\wedge ArrKT}, \mathsf{LType^\wedge KT\,B\,T}, \mathsf{LTerm^\wedge KT\,C\,t'})$.
- If $\mathsf{t_1} : \mathsf{LTerm\,(B \to A)}$ and $\mathsf{t_2} : \mathsf{LTerm\,B}$ then, by the same reasoning as in the previous case, we need to define $\mathsf{LTerm^\wedge KT\,A\,(app\,B\,t_1\,t_2)} : \mathsf{LTerm^\wedge\,(B \to A)\,(Arr^\wedge\,B\,A\,K_\top\,K_\top)\,t_1} \times \mathsf{LTerm^\wedge\,B\,K_\top\,t_2}$. We define the second component of the pair to be $\mathsf{LTerm^\wedge KT\,B\,t_2}$. We can define the first component from a proof of $\mathsf{LTerm^\wedge\,(B \to A)\,K_\top\,t_1}$ and the function $\mathsf{LTerm^\wedge EqualMap} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{Q\,Q'} : \mathsf{A} \to \mathsf{Set}) \to \mathsf{Equal^\wedge\,A\,A\,Q\,Q'\,refl} \to \mathsf{PredMap\,(LTerm\,A)\,(LTerm^\wedge\,A\,Q)\,(LTerm^\wedge\,A\,Q')}$ that takes two (extensionally) equal predicates with the same carrier and produces a morphism of predicates between their liftings. The definition of $\mathsf{LTerm^\wedge EqualMap}$ is straightforwardly given by pattern matching on the first two arguments to $\mathsf{PredMap}$ in its return type, using transitivity and symmetry of the type constructor Equal, together with the analogously defined functions $\mathsf{LType^\wedge EqualMap}$ and $\mathsf{Arr^\wedge EqualMap}$ in the cases when the first argument to $\mathsf{PredMap}$ is constructed using var and app, respectively. Taking $\mathsf{L_{K_\top}} : \mathsf{LTerm^\wedge\,(B \to A)\,K_\top\,t_1}$ to be the proof $\mathsf{L_{K_\top}} = \mathsf{LTerm^\wedge KT\,(B \to A)\,t_1}$ and taking $\mathsf{LTerm^\wedge Arr} : \mathsf{LTerm^\wedge\,(B \to A)\,(Arr^\wedge\,B\,A\,K_\top\,K_\top)\,t_1}$ to be the proof $\mathsf{LTerm^\wedge Arr} = \mathsf{LTerm^\wedge EqualMap\,K_\top\,(Arr^\wedge\,B\,A\,K_\top\,K_\top)\;Equal^\wedge ArrKT\,t_1\,L_{K_\top}}$, we define $\mathsf{LTerm^\wedge KT\,A\,(app\,B\,t_1\,t_2)} = (\mathsf{K_\top}, \mathsf{LTerm^\wedge Arr}, \mathsf{LTerm^\wedge KT\,B\,t_2})$.
- If $\mathsf{e} : \mathsf{Equal\,A\,(List\,B)}$ and $\mathsf{ts} : \mathsf{List\,(LTerm\,B)}$ then, as above, we need to define $\mathsf{LTerm^\wedge KT\,A\,(list\,B\,e\,ts)} : \mathsf{Equal^\wedge\,A\,(List\,B)\,K_\top\,(List^\wedge\,B\,K_\top)\,e} \times \mathsf{List^\wedge\,(LTerm\,B)\,(LTerm^\wedge\,B\,K_\top)\,ts}$. As in that case we assume a proof $\mathsf{EqualLListKT} : \mathsf{Equal^\wedge\,A\,(List\,B)\,K_\top\,(List^\wedge\,B\,K_\top)\,e}$ for the first component. We can define the second component using $\mathsf{liftListMap}$ from Section 4.3 to map a morphism $\mathsf{PredMap\,(LTerm\,B)\,(K_\top)\,(LTerm^\wedge\,B\,K_\top)}$ of predicates to a morphism $\mathsf{PredMap\,(List\,(LTerm\,B))\;(List^\wedge\,(LTerm\,B)\,K_\top)\,(List^\wedge\,(LTerm\,B)\,(LTerm^\wedge\,B\,K_\top))}$ of lifted predicates. Taking $\mathsf{m_{K_\top}} : \mathsf{PredMap\,(LTerm\,B)\,(K_\top)\,(LTerm^\wedge\,B\,K_\top)}$ to be the proof $\mathsf{m_{K_\top}\,t'\,tt} = \mathsf{LTerm^\wedge KT\,B\,t'}$, where $\mathsf{t'} : \mathsf{LTerm\,B}$ and tt is the single element of $\mathsf{K_\top\,t'}$, and taking $\mathsf{L_{List^\wedge LTerm^\wedge KT}} : \mathsf{List^\wedge\,(LTerm\,B)\,(LTerm^\wedge\,B\,K_\top)\,ts}$ to be the proof $\mathsf{L_{List^\wedge LTerm^\wedge KT}} = \mathsf{liftListMap\,(LTerm\,B)\,K_\top\,(LTerm^\wedge\,B\,K_\top)\,m_{K_\top}\,ts\,(List^\wedge KT\,(LTerm\,B)\,ts)}$, we define $\mathsf{LTerm^\wedge KT\,A\,(listC\,B\,e\,ts)} = (\mathsf{K_\top}, \mathsf{Equal^\wedge ListKT}, \mathsf{L_{ListL^\wedge Term^\wedge KT}})$.

The above techniques can be used to define a function $\mathsf{G^\wedge KT} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{x} : \mathsf{G\,A}) \to \mathsf{G^\wedge\,A\,K_\top\,x}$ for an arbitrary GADT G as defined in Section 5. To provide a proof of $\mathsf{G^\wedge\,A\,K_\top\,x}$ for every term $\mathsf{x} : \mathsf{G\,A}$, we need to know that if G has a constructor $\mathsf{c} : \mathsf{H}\,(\overline{\mathsf{F\,G\,\overline{B}}}) \to \mathsf{G}\,(\overline{\mathsf{K\,\overline{B}}})$, then H cannot construct a GADT so the generalization $\mathsf{H^\wedge Map}$ of $\mathsf{listLiftMap}$ in the final bullet point above is guaranteed to exist. We also need to know that the lifting of $\mathsf{K_\top}$ to types constructed by any nested type constructor F is extensionally equal to $\mathsf{K_\top}$ on the types it constructs. For example, we might need a proof that $\mathsf{Pair^\wedge\,A\,B\,K_\top\,K_\top}$ is equal to $\mathsf{K_\top}$ on $\mathsf{A} \times \mathsf{B}$. Given a pair $(\mathsf{a}, \mathsf{b}) : \mathsf{A} \times \mathsf{B}$, we have that $\mathsf{Pair^\wedge\,A\,B\,K_\top\,K_\top(a, b)} = \mathsf{K_\top\,a} \times \mathsf{K_\top\,b} = \top \times \top$, whereas $\mathsf{K_\top\,(a, b)} = \top$. While these types are not equal, they are clearly isomorphic. Similar isomorphisms between $\mathsf{F^\wedge\,A\,K_\top}$ and $\mathsf{K_\top}$ hold for all other nested type constructors F as well. These isomorphisms can either be proved on an as-needed basis or, since $\mathsf{F^\wedge\,A\,K_\top} = \mathsf{K_\top}$ 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.* 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. *Functorial GADTs are not Parametric.* 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.