(Deep) Induction Rules For GADTs

By: **Patricia Johann** and **Enrico Ghiorzi**

Abstract

Deep data types are those that are constructed from other data types, including, possibly, themselves. In this case, they are said to be truly nested. Deep induction is an extension of structural induction that traverses all of the structure in a deep data type, propagating predicates on its primitive data throughout the entire structure. Deep induction can be used to prove properties of nested types, including truly nested types, that cannot be proved via structural induction. In this paper we show how to extend deep induction to GADTs that are not truly nested GADTs. This opens the way to incorporating automatic generation of (deep) induction rules for them into proof assistants. We also show that the techniques developed in this paper do not suffice for extending deep induction to truly nested GADTs, so more sophisticated techniques are needed to derive deep induction rules for them.

Johann P, Ghiorzi E. (Deep) induction rules for GADTs. Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. January 2022:324-337. doi:10.1145/3497775.3503680. Publisher version of record available at: http://dl.acm.org/doi/10.1145/3497775.3503680

(Deep) Induction Rules for GADTs

Patricia Johann
johannp@appstate.edu
Appalachian State University
USA

Enrico Ghiorzi[*]
ghiorzie@appstate.edu
Appalachian State University
USA

Abstract

Deep data types are those that are constructed from other data types, including, possibly, themselves. In this case, they are said to be truly nested. Deep induction is an extension of structural induction that traverses *all* of the structure in a deep data type, propagating predicates on its primitive data throughout the entire structure. Deep induction can be used to prove properties of nested types, including truly nested types, that cannot be proved via structural induction. In this paper we show how to extend deep induction to GADTs that are not truly nested GADTs. This opens the way to incorporating automatic generation of (deep) induction rules for them into proof assistants. We also show that the techniques developed in this paper do not suffice for extending deep induction to truly nested GADTs, so more sophisticated techniques are needed to derive deep induction rules for them.

CCS Concepts: • **Theory of computation** → **Semantics and reasoning**; **Categorical semantics**;

Keywords: GADTs, induction, proof assistants

ACM Reference Format:
Patricia Johann and Enrico Ghiorzi. 2022. (Deep) Induction Rules for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '22), January 17–18, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3497775.3503680

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 [24], for example, functions and predicates over advanced types are defined inductively, and almost all non-trivial proofs of their properties are either proved by induction outright or rely on lemmas that are. Every time a new inductive type is declared in Coq an induction rule is automatically generated for it.

The inductive data types handled by Coq include (possibly mutually inductive) polynomial algebraic data types (ADTs), and the induction rules Coq generates for them are the expected ones for standard structural induction. However, as discussed in [13], it has long been understood that these rules are too weak to be genuinely useful for deep ADTs.[1] The following data type of rose trees, here coded in Agda and defined in terms of the standard type List of lists (see Section 2), is a deep ADT:

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

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

```
∀ (A : Set) (P : Rose A → Set) →
    P empty →
    (∀ (a : A) (ts : List (Rose A)) → P (node a ts)) →
    ∀ (x : Rose A) → P x
```

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 types becoming increasingly ubiquitous in formalizations, such an enhanced notion of induction is essential if proof assistants are to be able to automatically generate genuinely useful induction rules for data types that go beyond traditional ADTs. These include not just deep ADTs, but also (truly[2]) nested types [3], generalized algebraic data types

[*] New address: Istituto Italiano di Tecnologia, Italy, enrico.ghiorzi@iit.it

[1] An ADT/nested type/GADT is *deep* if it is (possibly mutually inductively) defined in terms of other ADTs/nested types/GADTs (including, possibly, itself).

[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.

https://doi.org/10.1145/3497775.3503680

(GADTs) [4, 14, 22, 27], more richly indexed families [5], and deep variants of all of these. A summary of the various classes of data types considered in this paper is given in Table 1.

Table 1. Data types in this paper

Data types	Discussed in Sections	Examples
ADTs	1	List, Rose
Nested types	2	PTree
Truly nested types	2	Bush
GADTs	3	Eq, Seq
Truly nested GADTs	3 and 6	G in (12)

Deep induction [13] 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 [13] to deliver induction rules appropriate to nested types, including ADTs. For the (deep) ADT of rose trees, for example, it gives the following genuinely useful induction rule:

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

Here, $\mathsf{List}^{\wedge}$ (resp., $\mathsf{Rose}^{\wedge}$) lifts its predicate argument P (resp., Q) on data of type Rose A (resp., A) to a predicate on data of type List (Rose A) (resp., Rose A) asserting that P (resp., Q) holds for every element of its list (resp., rose tree) argument.[3] Deep induction was also shown in [13] 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 (truly) nested types is reviewed in Section 2.

This paper shows how to extend deep induction to proper GADTs, i.e., GADTs that are not nested types (and thus are not ADTs). Typical applications of such GADTs include generic programming, modeling programming languages via higher-order abstract syntax, maintaining invariants in data structures, and expressing constraints in embedded domain-specific languages. They have also been used to implement tagless interpreters [14, 18, 19] by trading the definition of a universal value domain for a direct specification of the property of being a value. Other applications are described in, e.g., [17, 20]. A constructor for 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. These can even include instances involving G itself. But if G is a proper GADT, then at least one of its constructors will also have a structured instance of G — albeit one not involving G itself — as its codomain. For example, the constructor pair for the GADT

$$
\begin{array}{l}
\mathsf{data\,Seq} \ : \mathsf{Set} \to \mathsf{Set\,where} \\
\quad \mathsf{const} : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{Seq\,A} \\
\quad \mathsf{pair} \quad : \forall\{\mathsf{A\,B} : \mathsf{Set}\} \to \mathsf{Seq\,A} \to \mathsf{Seq\,B} \to \mathsf{Seq}\,(\mathsf{A} \times \mathsf{B})
\end{array}
\tag{2}
$$

of sequences[4] only constructs sequences of data whose types are pair-structured, rather than sequences of arbitrary type, as does const. If one or more of the data constructors for a GADT G return structured instances of G, then the GADT will have two distinct, but equally natural, semantics: a functorial semantics interpreting it as a left Kan extension [15], and a parametric semantics interpreting it as the interpretation of its Church encoding [1, 26]. As detailed in [10], a key difference in the two semantics is that the former views GADTs as their *functorial completions* [12], 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. Fortunately, these two views of GADTs coincide for those GADTs that are ADTs or (other, including truly) nested types. However, both they and their attendant properties differ greatly for proper GADTs. In fact, the functorial and parametric semantics for proper GADTs are sufficiently disparate that, by contrast with the semantics customarily given for ADTs and nested types [2, 7, 11], it is not at all clear how to define a functorial parametric semantics for GADTs [10].

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 GADTs viewed as their functorial completions do not obviously support induction rules. This makes sense intuitively: induction is a syntactic proof technique, so it may not be possible to

[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. The predicate lifting for a container type like List A or Rose A simply traverses containers of that type and applies its predicate argument pointwise to the constituent data of type A. The ability to define predicate liftings for more general data types will be critical to deriving their deep induction rules in Section 5.

[4] The type of Seq is actually $\mathsf{Set} \to \mathsf{Set}_1$, but to aid readability we elide the explicit tracking of Agda universe levels in this paper.

use it to prove properties of those elements of a GADT's functorial completion that are not expressible in syntax. All is not lost, however. As we show below, the syntax-only view of GADTs determined by their Church encodings *does* support induction rules — including deep induction rules — for GADTs. Indeed, this paper gives the first-ever deep induction rules for proper GADTs. But it actually delivers far more: it gives a *general framework* for deriving deep induction rules for GADTs that can be instantiated to particular GADTs of interest. This framework can serve as a basis for extending modern proof assistants' automatic generation of structural induction rules for ADTs to automatic generation of deep induction rules for GADTs. In addition, as for ADTs and nested types, the structural induction rule for any GADT can be recovered from its deep induction rule by taking the custom predicates in its deep induction rule to be constantly True-valued (i.e., constantly ⊤-valued) predicates.

Significantly, deep induction rules for GADTs cannot be derived by somehow extending the approach of [13] to syntax-only GADTs. Indeed, the approach taken there makes crucial use of the functoriality of data types' interpretations from [12], and functoriality is precisely what interpreting GADTs as the interpretations of their Church encodings fails to deliver; see [9] for a discussion of why Seq, e.g., is not functorial. Our approach is to instead first give a predicate lifting styled after those of [13], together with a (deep) induction rule, for the simplest — and arguably most important — GADT, namely the equality GADT (4). We then derive the deep induction rule for a more complex GADT G by *i*) using the equality GADT to represent G as its so-called *Henry Ford encoding* [4, 8, 16, 21, 22], and *ii*) using the predicate liftings for the equality GADT and the 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 GADTs. To illustrate, we introduce a series of increasingly complex GADTs as running examples in Section 3 and derive a deep induction rule for each of them in Section 4. In particular, we derive the deep induction rules for the equality data type in Section 4.1 and the Seq data type in (2) in Section 4.2. We present our general framework for deriving (deep) induction rules for 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 clear how to extend induction — deep or otherwise — to *truly nested GADTs*, i.e., to proper GADTs whose recursive occurrences appear below themselves.[5] This does not appear to be much of a restriction, however, since truly nested GADTs do not, to our knowledge, appear in practice or in the literature. Section 7 comprises a case study in using deep induction. All of the deep induction rules appearing in this paper have been derived by instantiating our general framework. Our Agda implementation of them is available at https://cs.appstate.edu/~johannp/CPP22Code.html.

Additional Related Work Various techniques for deriving induction rules for data types that go beyond ADTs have been studied. For example, Fu and Selinger [6] show, via examples, how to derive induction rules for arbitrary nested types. Unfortunately, however, their technique is rather *ad hoc*, so is unclear how to generalize it to nested types other than the specific ones studied there. Moreover, [6] 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 [23] derives induction rules for data type definitions in Coq using unary parametricity. His technique seems to be essentially equivalent to that of [12] for nested types, although he does not permit true nesting. More recently, Ullrich [25] has implemented a plugin in MetaCoq to generate induction rules for nested types. This plugin is also based on unary parametricity, and true nesting still is not permitted. As far as we know no attempts have been made to extend either implementation to truly nested types or to proper GADTs or their deep variants. Other systems, including Isabelle and Lean, also derive induction rules for data types that go beyond ADTs. But we know of no work other than that reported here that specifically addresses induction rules for the (deep) GADTs considered in this paper.

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 : Set is a function P : A → Set mapping each a : A to the set of proofs that P a holds. For example, the

[5] Note carefully the distinction between a GADT that is not a nested type — i.e., a proper GADT — and a proper GADT that is not a truly nested GADT. In fact, truly nested types are not proper GADTs and truly nested GADTs are not (truly) nested types. There is ample scope for confusion in light of the original, and now well-established, use of the term "nested type" to refer to *any* type that allows non-variable instances in the domains of its constructors, whether or not that type involves actual nesting [3].

structural induction rule for the list type

$$\begin{array}{l}\mathsf{data\ List : Set \to Set\ where}\\ \quad \mathsf{nil} \quad\ \ : \forall\{\mathsf{A : Set}\} \to \mathsf{List\ A}\\ \quad \mathsf{cons} \ : \forall\{\mathsf{A : Set}\} \to \mathsf{A} \to \mathsf{List\ A} \to \mathsf{List\ A}\end{array}$$

is

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

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

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

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

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

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 $\mathsf{PTree\,(A \times A)}$ to construct data of type PTree A. Thus, it is clear that the instances of PTree at various indices cannot be defined independently, and that the entire inductive family of types must therefore be defined at once. This intertwinedness of the instances of nested types is reflected in their structural induction rules, which, as explained in [13], 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

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

The deep induction rule for perfect trees similarly uses polymorphic predicates but otherwise follows the familiar pattern. It is given by the first expression in Figure 1. There, $\mathsf{Pair}^\wedge : \forall(\mathsf{A\ B : Set}) \to (\mathsf{A \to Set}) \to (\mathsf{B \to Set}) \to \mathsf{A \times B \to Set}$ lifts predicates $\mathsf{Q_A}$ on data of type A and $\mathsf{Q_B}$ on data of type B to a predicate on pairs of type $\mathsf{A \times B}$ in such a way that $\mathsf{Pair}^\wedge\,\mathsf{A\ B\ Q_A\ Q_B\ (a, b) = Q_A\ a \times Q_B\ b}$. Similarly, $\mathsf{PTree}^\wedge : \forall(\mathsf{A : Set}) \to (\mathsf{A \to Set}) \to \mathsf{PTree\ A \to Set}$ lifts a predicate Q on data of type A to a predicate on data of type PTree A asserting that Q holds for every element of type A contained in its perfect tree argument. A general definition of liftings for a robust class of GADTs including all those appearing in the literature is given in Section 5.

Using deep induction we can 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[6][3]:

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

Even defining a structural induction rule for bushes requires that we be able to lift the rule's polymorphic predicate argument to Bush itself. This observation was, in fact, the original motivation for the development of deep induction in [13]. The deep induction rule for bushes is given by the second expression in Figure 1, where

$$\mathsf{Bush}^\wedge : \forall(\mathsf{A : Set}) \to (\mathsf{A \to Set}) \to \mathsf{Bush\ A \to Set}$$

is the following lifting of a predicate Q on data of type A to a predicate on data of type Bush A asserting that Q holds for every element of type A contained in its argument bush:

$$\begin{array}{lcl}\mathsf{Bush}^\wedge\,\mathsf{A\ Q\ bnil} & = & \top\\ \mathsf{Bush}^\wedge\,\mathsf{A\ Q\,(bcons\ a\ bb)} & = & \\ \qquad \mathsf{Q\ a} \times \mathsf{Bush}^\wedge\,\mathsf{(Bush\ A)}\,(\mathsf{Bush}^\wedge\,\mathsf{A\ Q})\ \mathsf{bb} & & \end{array} \tag{3}$$

We note that, as for ADTs, the structural induction rule for any (truly) nested type can be obtained as the special case of its deep induction rule in which the custom predicates are taken to be constantly $\top$-valued predicates. This instantiation ensures that the resulting induction rule only inspects the top-level structure of its argument, rather than the contents of that structure, which is exactly what structural induction should do.

Under some circumstances deep induction can be mimicked by hand-threading applications of structural induction through the layers of data comprising a deep data type. But this is not the case if, e.g., one or more of the custom predicates in the data type's deep induction rule is not the characteristic function of an inductive data type.

3 (Deep) GADTs

While a data constructor for a nested type can take *as arguments* data whose types involve instances of that type at indices other than the one being defined, its return type must still be at the (variable) type instance being defined. For example, each of pleaf and pnode returns an element of type PTree A regardless of the instances of PTree appearing

[6] To define truly nested types in Agda we must use the NO_POSITIVITY_CHECK flag, and to define functions over them we must use the TERMINATING flag. (Similar flags are required in Coq.) Although as programmers we know from the metatheory in [12] that Bush is well-defined and the functions we define over them terminate, the flags are necessary because Agda fails to infer these facts. Analogous comments apply at several places below.

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

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

Figure 1. Deep induction rules for perfect trees and bushes

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. For every GADT in this paper, we require that the instance of the return type for each of its data constructors is a polynomial in that constructor's type arguments.

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 using pattern matching to force the desired type refinement. For example, the *equality* GADT

$$\begin{aligned}
&\mathsf{data}\ \mathsf{Equal} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}\ \mathsf{where} \\
&\quad \mathsf{refl} :\ \forall \{\mathsf{A} : \mathsf{Set}\} \to \mathsf{Equal}\,\mathsf{A}\,\mathsf{A}
\end{aligned} \tag{4}$$

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

The importance of the equality GADT lies in the fact that we can understand other GADTs in terms of it. For example, the GADT Seq from (2) comprises constant sequences of data of any type A and sequences obtained by pairing the data in two already existing sequences. This GADT can be rewritten as its Henry Ford encoding [4, 8, 16, 21, 22], which makes critical use of the equality GADT, as follows:

$$\begin{aligned}
&\mathsf{data}\ \mathsf{Seq} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where} \\
&\quad \mathsf{const} :\ \forall \{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{Seq}\,\mathsf{A} \\
&\quad \mathsf{pair}\ \ :\ \forall \{\mathsf{A} : \mathsf{Set}\} \to \forall (\mathsf{B}\,\mathsf{C} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{C}) \to \\
&\qquad\qquad \mathsf{Seq}\,\mathsf{B} \to \mathsf{Seq}\,\mathsf{C} \to \mathsf{Seq}\,\mathsf{A}
\end{aligned} \tag{5}$$

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.

Although Seq does not at first glance appear to be a deep GADT, when written as its Henry Ford encoding, it, like all GADTs, can be regarded as "deep over Equal". By contrast, the GADT LTerm in Figure 2, which is inspired by [28], is *inherently deep*. It encodes terms of a simply typed lambda calculus. More robust variations on LTerm are, of course, possible, but this variation is rich enough to illustrate all essential aspects of deep GADTs — and later, in Section 4.3, their deep induction rules — while still being small enough to ensure clarity of exposition.

Types are either booleans, arrow types, or list types. They are represented by the Henry Ford GADT LType in Figure 2. Terms are either variables, abstractions, applications, or lists of terms. They are similarly represented by the Henry Ford GADT LTerm. The type parameter for LTerm tracks the types of simply typed lambda calculus terms. For example, LTerm A is the type of simply typed lambda terms of type A. Variables are tagged with their types by the data constructors var and abs, whose LType arguments ensure that their type tags are legal types. This ensures that all lambda terms produced by var, abs, app, and list are well-typed. We will revisit these GADTs in Sections 4 and 7.

4 (Deep) Induction for GADTs

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

$$\begin{aligned}
\mathsf{Equal}^{\wedge} : \forall (\mathsf{A}\,\mathsf{B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to (\mathsf{B} \to \mathsf{Set}) \to \\
\mathsf{Equal}\,\mathsf{A}\,\mathsf{B} \to \mathsf{Set}
\end{aligned}$$

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 does exactly what we expect: it takes two predicates on the same type as input and is inhabited iff they are extensionally

```
data LType : Set → Set where
  bool : ∀{A : Set} → ∀(B : Set) → Equal A Bool → LType A
  arr   : ∀{A : Set} → ∀(B C : Set) → Equal A (B → C) → LType B → LType C → LType A
  list   : ∀{A : Set} → ∀(B : Set) → Equal A (List  B) → LType B → LType A

data LTerm : Set → Set where
  var   : ∀{A : Set} → String → LType A → LTerm A
  abs  : ∀{A : Set} → ∀(B C : Set) → Equal A (B → C) → String → LType B → LTerm C → LTerm A
  app  : ∀{A : Set} → ∀(B : Set) → LTerm(B → A) → LTerm B → LTerm A
  list   : ∀{A : Set} → ∀(B : Set) → Equal A (List B) → List (LTerm B) → LTerm A
```

Figure 2. The LType and LTerm data types

equal. 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 dIndRefl is thus associated with the refl constructor for Equal:

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

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

$$\begin{aligned}\forall(\mathsf{P} : \forall(\mathsf{A\,B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to (\mathsf{B} \to \mathsf{Set}) \to\\ \mathsf{Equal\,A\,B} \to \mathsf{Set}) \to \mathsf{dIndRefl\,P} \to\\ \forall(\mathsf{A\,B} : \mathsf{Set})(\mathsf{Q_A} : \mathsf{A} \to \mathsf{Set})(\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set})(\mathsf{e} : \mathsf{Equal\,A\,B})\\ \to \mathsf{Equal}^\wedge\,\mathsf{A\,B\,Q_A\,Q_B\,e} \to \mathsf{P\,A\,B\,Q_A\,Q_B\,e}\end{aligned} \tag{6}$$

To prove that this rule is sound we must provide a witness dIndEqual inhabiting the type in (6). By pattern matching, we need only consider the case where A = B and e = refl, so we can define dIndEqual by

$$\mathsf{dIndEqual\ P\ crefl\ A\ A\ Q_A\ Q'_A\ refl\ liftE} = \mathsf{crefl\ A\ Q_A\ Q'_A\ liftE}$$

To recover Equal's structural induction rule

$$\begin{aligned}\forall(\mathsf{Q} : \forall(\mathsf{A\,B} : \mathsf{Set}) \to \mathsf{Equal\,A\,B} \to \mathsf{Set}) \to\\ \big(\forall(\mathsf{C} : \mathsf{Set}) \to \mathsf{Q\,C\,C\,refl}\big) \to\\ \forall(\mathsf{A\,B} : \mathsf{Set})(\mathsf{e} : \mathsf{Equal\,A\,B}) \to \mathsf{Q\,A\,B\,e}\end{aligned} \tag{7}$$

we define a term indEqual of the type in (7) by indEqual Q srefl A B refl = dIndEqual P srefl′ A B $\mathsf{K}_\top^\mathsf{A}$ $\mathsf{K}_\top^\mathsf{B}$ refl sliftE. Here,

$$\begin{aligned}\mathsf{P} : \forall(\mathsf{A\,B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to (\mathsf{B} \to \mathsf{Set}) \to\\ \mathsf{Equal\,A\,B}\!\to \mathsf{Set}\end{aligned}$$

is defined by P A B $\mathsf{Q_A}$ $\mathsf{Q_B}$ e = Q A B e, $\mathsf{K}_\top^\mathsf{A}$ and $\mathsf{K}_\top^\mathsf{B}$ are the constantly ⊤-valued predicates on A and B, respectively, sliftE : $\mathsf{Equal}^\wedge$ A B $\mathsf{K}_\top^\mathsf{A}$ $\mathsf{K}_\top^\mathsf{B}$ refl is defined by

$$\mathsf{sliftE\,a} = \mathsf{refl} : \mathsf{Equal}\ \top\ \top$$

for every a : A, and

$$\begin{aligned}\mathsf{srefl'} : \forall(\mathsf{C} : \mathsf{Set})(\mathsf{Q_c\,Q'_c} : \mathsf{C} \to \mathsf{Set}) \to\\ \mathsf{Equal}^\wedge\,\mathsf{C\,C\,Q_c\,Q'_c\,refl} \to \mathsf{Q\,C\,C\,refl}\end{aligned}$$

is defined by srefl′ C $\mathsf{Q_c}$ $\mathsf{Q'_c}$ liftE′ = srefl C. The structural induction rule for any GADT G that is not truly nested can similarly be recovered from its deep induction rule by instantiating every custom predicate by the appropriate constantly ⊤-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 (5). We first define its predicate lifting

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

as in Figure 3. There, a : A, $\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}$, $\mathsf{Q_C} : \mathsf{C} \to \mathsf{Set}$, e : Equal A (B × C), $\mathsf{s_B}$: Seq B, $\mathsf{s_C}$: Seq C, and ∃[x] F x is syntactic sugar for the type of dependent pairs (x, b), where x : A, b : F x, and F : A → Set. The lifting $\mathsf{Seq}^\wedge$ is derived as in Section 5. Next, let dIndConst and dIndPair be the induction hypotheses associated with the constructors const and pair, respectively. These are given in Figure 3 as well. Then the deep induction rule for Seq is given in the last two lines of Figure 3.

To prove that this rule is sound we provide a witness dIndSeq inhabiting the type in the last two lines of Figure 3 by

$$\begin{aligned}&\mathsf{dIndSeq\ P\ cconst\ cpair\ A\ Q_A\ (const\,a)\ liftA}\\ =\ &\mathsf{cconst\ A\ Q_A\ a\ liftA}\end{aligned}$$

and

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

$$
\begin{array}{lcl}
\mathsf{Seq}^{\wedge}\,\mathsf{A}\,\mathsf{Q_A}\,(\mathsf{const\,a}) & = & \mathsf{Q_A\,a} \\
\mathsf{Seq}^{\wedge}\,\mathsf{A}\,\mathsf{Q_A}\,(\mathsf{pair\,B\,C\,e}\,\mathsf{s_B}\,\mathsf{s_C}) & = & \exists[\mathsf{Q_B}]\exists[\mathsf{Q_C}]\,\mathsf{Equal}^{\wedge}\,\mathsf{A}\,(\mathsf{B}\times\mathsf{C})\,\mathsf{Q_A}\,(\mathsf{Pair}^{\wedge}\,\mathsf{B\,C\,Q_B\,Q_C})\,\mathsf{e}\times\mathsf{Seq}^{\wedge}\,\mathsf{B\,Q_B\,s_B}\times\mathsf{Seq}^{\wedge}\,\mathsf{C\,Q_C\,s_C} \\
\\
\mathsf{dIndConst} & = & \lambda(\mathsf{P}:\forall(\mathsf{A}:\mathsf{Set})\to(\mathsf{A}\to\mathsf{Set})\to\mathsf{Seq\,A}\to\mathsf{Set})\to \\
& & \quad\forall(\mathsf{A}:\mathsf{Set})(\mathsf{Q_A}:\mathsf{A}\to\mathsf{Set})(\mathsf{a}:\mathsf{A})\to\mathsf{Q_A\,a}\to\mathsf{P\,A\,Q_A}\,(\mathsf{const\,a}) \\
\\
\mathsf{dIndPair} & = & \lambda(\mathsf{P}:\forall(\mathsf{A}:\mathsf{Set})\to(\mathsf{A}\to\mathsf{Set})\to\mathsf{Seq\,A}\to\mathsf{Set})\to \\
& & \quad\forall(\mathsf{A\,B\,C}:\mathsf{Set})(\mathsf{Q_A}:\mathsf{A}\to\mathsf{Set})(\mathsf{Q_B}:\mathsf{B}\to\mathsf{Set})(\mathsf{Q_C}:\mathsf{C}\to\mathsf{Set}) \\
& & \qquad\qquad(\mathsf{s_B}:\mathsf{Seq\,B})(\mathsf{s_C}:\mathsf{Seq\,C})(\mathsf{e}:\mathsf{Equal\,A}\,(\mathsf{B}\times\mathsf{C}))\to \\
& & \quad\mathsf{Equal}^{\wedge}\mathsf{A}\,(\mathsf{B}\times\mathsf{C})\,\mathsf{Q_A}\,(\mathsf{Pair}^{\wedge}\,\mathsf{B\,C\,Q_B\,Q_C})\,\mathsf{e}\to\mathsf{P\,B\,Q_B\,s_B}\to \\
& & \qquad\qquad\mathsf{P\,C\,Q_C\,s_C}\to\mathsf{P\,A\,Q_A}\,(\mathsf{pair\,B\,C\,e\,s_B\,s_C})
\end{array}
$$

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

Figure 3. Deep induction rule for Seq

In the first clause, $\mathsf{a}:\mathsf{A}$, $\mathsf{Q_A}:\mathsf{A}\to\mathsf{Set}$, and $\mathsf{liftA}:\mathsf{Seq}^{\wedge}\,\mathsf{A\,Q_A}\,(\mathsf{const\,a})=\mathsf{Q_A\,a}$. In the second clause we also have

$$
\begin{array}{lcl}
\mathsf{Q_B} & : & \mathsf{B}\to\mathsf{Set} \\
\mathsf{Q_C} & : & \mathsf{C}\to\mathsf{Set} \\
\mathsf{e} & : & \mathsf{Equal\,A}\,(\mathsf{B}\times\mathsf{C}) \\
\mathsf{s_B} & : & \mathsf{Seq\,B} \\
\mathsf{s_C} & : & \mathsf{Seq\,C} \\
\mathsf{liftE} & : & \mathsf{Equal}^{\wedge}\,\mathsf{A}\,(\mathsf{B}\times\mathsf{C})\,\mathsf{Q_A}\,(\mathsf{Pair}^{\wedge}\,\mathsf{B\,C\,Q_B\,Q_C})\,\mathsf{e} \\
\mathsf{liftB} & : & \mathsf{Seq}^{\wedge}\,\mathsf{B\,Q_B\,s_B} \\
\mathsf{liftC} & : & \mathsf{Seq}^{\wedge}\,\mathsf{C\,Q_C\,s_C}
\end{array}
$$

Together these give that

$$(\mathsf{Q_B},\mathsf{Q_C},\mathsf{liftE},\mathsf{liftB},\mathsf{liftC})\ :\ \mathsf{Seq}^{\wedge}\,\mathsf{A\,Q}\,(\mathsf{pair\,B\,C\,e\,s_B\,s_C})$$

We therefore have

$$
\begin{array}{lclcl}
\mathsf{p_B} & = & \mathsf{dIndSeq\,P\,cconst\,cpair\,B\,Q_B\,s_B\,liftB} & : & \mathsf{P\,B\,Q_B\,s_B} \\
\mathsf{p_C} & = & \mathsf{dIndSeq\,P\,cconst\,cpair\,C\,Q_C\,s_C\,liftC} & : & \mathsf{P\,C\,Q_C\,s_C}
\end{array}
$$

4.3 (Deep) Induction for LTerm

To derive the deep induction rule for the GADT LTerm we use its Henry Ford encoding from Figure 2. We first define the predicate lifting

$$
\begin{array}{l}
\mathsf{Arr}^{\wedge}:\forall(\mathsf{A\,B}:\mathsf{Set})\to(\mathsf{A}\to\mathsf{Set})\to(\mathsf{B}\to\mathsf{Set})\to \\
\qquad\qquad\qquad\qquad(\mathsf{A}\to\mathsf{B})\to\mathsf{Set}
\end{array}
$$

for arrow types following the general framework in Section 5, since arrow types appear in LType and LTerm. It is given by

$$\mathsf{Arr}^{\wedge}\,\mathsf{A\,B\,Q_A\,Q_B\,f}=\forall(\mathsf{a}:\mathsf{A})\to\mathsf{Q_A\,a}\to\mathsf{Q_B}\,(\mathsf{f\,a})$$

The predicate liftings

$$\mathsf{LType}^{\wedge}:\forall(\mathsf{A}:\mathsf{Set})\to(\mathsf{A}\to\mathsf{Set})\to\mathsf{LType\,A}\to\mathsf{Set}$$

for LType and

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

for LTerm are defined in Figure 4 following the general framework in Section 5. There,

$$
\begin{array}{lcl}
\mathsf{s} & : & \mathsf{String} \\
\mathsf{Q_A} & : & \mathsf{A}\to\mathsf{Set} \\
\mathsf{Q_B} & : & \mathsf{B}\to\mathsf{Set} \\
\mathsf{Q_C} & : & \mathsf{C}\to\mathsf{Set} \\
\mathsf{T_A} & : & \mathsf{LType\,A} \\
\mathsf{T_B} & : & \mathsf{LType\,B} \\
\mathsf{T_C} & : & \mathsf{LType\,C} \\
\mathsf{t_B} & : & \mathsf{LTerm\,B} \\
\mathsf{t_C} & : & \mathsf{LTerm\,C} \\
\mathsf{t_{BA}} & : & \mathsf{LTerm}\,(\mathsf{B}\to\mathsf{A}) \\
\mathsf{ts} & : & \mathsf{List}\,(\mathsf{LTerm\,B})
\end{array}
$$

and $\mathsf{K}_{\top}^{\mathsf{Bool}}$ is the constantly $\top$-valued predicate on Bool and $\mathsf{List}^{\wedge}$ is the predicate lifting for lists from (1). Also,

$$
\begin{array}{lcll}
\mathsf{e} & : & \mathsf{Equal\,A\,Bool} & \text{in the first clause,} \\
\mathsf{e} & : & \mathsf{Equal\,A}\,(\mathsf{B}\to\mathsf{C}) & \text{in the second and fifth clauses,} \\
\mathsf{e} & : & \mathsf{Equal\,A}\,(\mathsf{List\,B}) & \text{in the third clause,} \\
\mathsf{e} & : & \mathsf{Equal\ \ A\ \ }(\mathsf{List\ \ B}) & \text{in the seventh clause.}
\end{array}
$$

With these liftings in hand we can define the induction hypotheses dIndVar, dIndAbs, dIndApp, and dIndList associated with LTerms's data constructors. These are given in Figure 5. The deep induction rule for LTerm is thus

$$
\begin{array}{l}
\forall(\mathsf{P}:\forall(\mathsf{A}:\mathsf{Set})\to(\mathsf{A}\to\mathsf{Set})\to\mathsf{LTerm\,A}\to\mathsf{Set})\to \\
\quad\mathsf{dIndVar\,P}\to\mathsf{dIndAbs\,P}\to\mathsf{dIndApp\,P}\to\mathsf{dIndList\,P}\to \\
\qquad\forall(\mathsf{A}:\mathsf{Set})(\mathsf{Q_A}:\mathsf{A}\to\mathsf{Set})(\mathsf{t_A}:\mathsf{LTerm\,A})\to \\
\qquad\quad\mathsf{LTerm}^{\wedge}\,\mathsf{A\,Q_A\,t_A}\to\mathsf{P\,A\,Q_A\,t_A}
\end{array}
\tag{8}
$$

To prove this rule sound we define a witness dIndLTerm inhabiting the type in (8) as in Figure 6. There,

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

Figure 4. Predicate liftings for LType and LTerm

$$\begin{array}{lcl}
\mathrm{dIndVar} & = & \lambda(\mathrm{P} : \forall(\mathrm{A} : \mathrm{Set}) \to (\mathrm{A} \to \mathrm{Set}) \to \mathrm{LTerm}\ \mathrm{A} \to \mathrm{Set}) \to \\
& & \quad \forall(\mathrm{A} : \mathrm{Set})(\mathrm{Q_A} : \mathrm{A} \to \mathrm{Set})(\mathrm{s} : \mathrm{String})(\mathrm{T_A} : \mathrm{LType}\ \mathrm{A}) \to \mathrm{LType}^{\wedge}\ \mathrm{A}\ \mathrm{Q_A}\ \mathrm{T_A} \to \mathrm{P}\ \mathrm{A}\ \mathrm{Q_A}\ (\mathrm{var}\ \mathrm{s}\ \mathrm{T_A}) \\
\mathrm{dIndAbs} & = & \lambda(\mathrm{P} : \forall(\mathrm{A} : \mathrm{Set}) \to (\mathrm{A} \to \mathrm{Set}) \to \mathrm{LTerm}\ \mathrm{A} \to \mathrm{Set}) \to \\
& & \quad \forall(\mathrm{A}\ \mathrm{B}\ \mathrm{C} : \mathrm{Set})(\mathrm{Q_A} : \mathrm{A} \to \mathrm{Set})(\mathrm{Q_B} : \mathrm{B} \to \mathrm{Set})(\mathrm{Q_C} : \mathrm{C} \to \mathrm{Set})(\mathrm{e} : \mathrm{Equal}\ \mathrm{A}\ (\mathrm{B} \to \mathrm{C}))(\mathrm{s} : \mathrm{String}) \to \\
& & \quad (\mathrm{T_B} : \mathrm{LType}\ \mathrm{B}) \to (\mathrm{t_C} : \mathrm{LTerm}\ \mathrm{C}) \to \mathrm{Equal}^{\wedge}\ \mathrm{A}\ (\mathrm{B} \to \mathrm{C})\ \mathrm{Q_A}\ (\mathrm{Arr}^{\wedge}\ \mathrm{B}\ \mathrm{C}\ \mathrm{Q_B}\ \mathrm{Q_C})\ \mathrm{e} \to \\
& & \quad \mathrm{LType}^{\wedge}\ \mathrm{B}\ \mathrm{Q_B}\ \mathrm{T_B} \to \mathrm{P}\ \mathrm{C}\ \mathrm{Q_C}\ \mathrm{t_C} \to \mathrm{P}\ \mathrm{A}\ \mathrm{Q_A}\ (\mathrm{abs}\ \mathrm{B}\ \mathrm{C}\ \mathrm{e}\ \mathrm{s}\ \mathrm{T_B}\ \mathrm{t_C}) \\
\\
\mathrm{dIndApp} & = & \lambda(\mathrm{P} : \forall(\mathrm{A} : \mathrm{Set}) \to (\mathrm{A} \to \mathrm{Set}) \to \mathrm{LTerm}\ \mathrm{A} \to \mathrm{Set}) \to \\
& & \quad \forall(\mathrm{A}\ \mathrm{B} : \mathrm{Set})(\mathrm{Q_A} : \mathrm{A} \to \mathrm{Set})(\mathrm{Q_B} : \mathrm{B} \to \mathrm{Set})(\mathrm{t_{BA}} : \mathrm{LTerm}\ (\mathrm{B} \to \mathrm{A}))(\mathrm{t_B} : \mathrm{LTerm}\ \mathrm{B}) \to \\
& & \quad \mathrm{P}\ (\mathrm{B} \to \mathrm{A})\ (\mathrm{Arr}^{\wedge}\ \mathrm{B}\ \mathrm{A}\ \mathrm{Q_B}\ \mathrm{Q_A})\ \mathrm{t_{BA}} \to \mathrm{P}\ \mathrm{B}\ \mathrm{Q_B}\ \mathrm{t_B} \to \mathrm{P}\ \mathrm{A}\ \mathrm{Q_A}\ (\mathrm{app}\ \mathrm{B}\ \mathrm{t_{BA}}\ \mathrm{t_B}) \\
\mathrm{dIndList} & = & \lambda(\mathrm{P} : \forall(\mathrm{A} : \mathrm{Set}) \to (\mathrm{A} \to \mathrm{Set}) \to \mathrm{LTerm}\ \mathrm{A} \to \mathrm{Set}) \to \\
& & \quad \forall(\mathrm{A}\ \mathrm{B} : \mathrm{Set})(\mathrm{Q_A} : \mathrm{A} \to \mathrm{Set})(\mathrm{Q_B} : \mathrm{B} \to \mathrm{Set})(\mathrm{e} : \mathrm{Equal}\ \mathrm{A}\ (\mathrm{List}\ \mathrm{B}))(\mathrm{ts} : \mathrm{List}\ (\mathrm{LTerm}\ \mathrm{B})) \to \\
& & \quad \mathrm{Equal}^{\wedge}\ \mathrm{A}\ (\mathrm{List}\ \mathrm{B})\ \mathrm{Q_A}\ (\mathrm{List}^{\wedge}\ \mathrm{B}\ \mathrm{Q_B})\ \mathrm{e} \to \mathrm{List}^{\wedge}\ (\mathrm{LTerm}\ \mathrm{B})(\mathrm{P}\ \mathrm{B}\ \mathrm{Q_B})\ \mathrm{ts} \to \mathrm{P}\ \mathrm{A}\ \mathrm{Q_A}\ (\mathrm{list}\ \mathrm{B}\ \mathrm{e}\ \mathrm{ts})
\end{array}$$

Figure 5. Induction hypotheses for LTerm

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

where

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

Figure 6. dIndLTerm

$$\begin{array}{lcl}
\mathrm{s} & : & \mathrm{String} \\
\mathrm{Q_A} & : & \mathrm{A} \to \mathrm{Set} \\
\mathrm{Q_B} & : & \mathrm{B} \to \mathrm{Set} \\
\mathrm{Q_C} & : & \mathrm{C} \to \mathrm{Set} \\
\mathrm{T_A} & : & \mathrm{LType}\ \mathrm{A} \\
\mathrm{T_B} & : & \mathrm{LType}\ \mathrm{B} \\
\mathrm{t_B} & : & \mathrm{LTerm}\ \mathrm{B} \\
\mathrm{t_C} & : & \mathrm{LTerm}\ \mathrm{C} \\
\mathrm{t_{BA}} & : & \mathrm{LTerm}\ (\mathrm{B} \to \mathrm{A}) \\
\mathrm{ts} & : & \mathrm{List}\ (\mathrm{LTerm}\ \mathrm{B}) \\
\mathrm{liftA} & : & \mathrm{LTerm}^{\wedge}\ \mathrm{A}\ \mathrm{Q_A}\ (\mathrm{var}\ \mathrm{s}\ \mathrm{T_A}) = \mathrm{LType}^{\wedge}\ \mathrm{A}\ \mathrm{Q_A}\ \mathrm{T_A} \\
\mathrm{liftE} & : & \mathrm{Equal}^{\wedge}\ \mathrm{A}\ (\mathrm{B} \to \mathrm{C})\ \mathrm{Q_A}\ (\mathrm{Arr}^{\wedge}\ \mathrm{B}\ \mathrm{C}\ \mathrm{Q_B}\ \mathrm{Q_C})\ \mathrm{e} \\
\mathrm{lift}_{\mathrm{T_B}} & : & \mathrm{LType}^{\wedge}\ \mathrm{B}\ \mathrm{Q_B}\ \mathrm{T_B}
\end{array}$$

$$\begin{array}{lcl}
\mathrm{lift}_{\mathrm{t_C}} & : & \mathrm{LTerm}^{\wedge}\ \mathrm{C}\ \mathrm{Q_C}\ \mathrm{t_C} \\
\mathrm{lift}_{\mathrm{t_{BA}}} & : & \mathrm{LTerm}^{\wedge}\ (\mathrm{B} \to \mathrm{A})\ (\mathrm{Arr}^{\wedge}\ \mathrm{B}\ \mathrm{A}\ \mathrm{Q_B}\ \mathrm{Q_A})\ \mathrm{t_{BA}} \\
\mathrm{lift}_{\mathrm{t_B}} & : & \mathrm{LTerm}^{\wedge}\ \mathrm{B}\ \mathrm{Q_B}\ \mathrm{t_B} \\
\mathrm{liftE}' & : & \mathrm{Equal}^{\wedge}\ \mathrm{A}\ (\mathrm{List}\ \mathrm{B})\ \mathrm{Q_A}\ (\mathrm{List}^{\wedge}\ \mathrm{B}\ \mathrm{Q_B})\ \mathrm{e} \\
\mathrm{lift}_{\mathrm{List}} & : & \mathrm{List}^{\wedge}\ (\mathrm{LTerm}\ \mathrm{B})\ (\mathrm{LTerm}^{\wedge}\ \mathrm{B}\ \mathrm{Q_B})\ \mathrm{ts}
\end{array}$$

Moreover, in the definition of $\mathrm{p_{ts}}$,

$$\mathrm{PredMap} : \forall\ (\mathrm{A} : \mathrm{Set}) \to (\mathrm{A} \to \mathrm{Set}) \to (\mathrm{A} \to \mathrm{Set}) \to \mathrm{Set}$$

is the type constructor producing the type of morphisms between predicates defined by

$$\mathrm{PredMap}\ \mathrm{A}\ \mathrm{Q}\ \mathrm{Q}' = \forall\ (\mathrm{a} : \mathrm{A}) \to \mathrm{Q}\ \mathrm{a} \to \mathrm{Q}'\ \mathrm{a}$$

and

$$\begin{aligned}\mathsf{liftListMap} : \forall\,(A : \mathsf{Set}) \to (Q\,Q' : A \to \mathsf{Set}) \to{} &\\ \mathsf{PredMap}\,A\,Q\,Q' \to{} &\\ \mathsf{PredMap}\,(\mathsf{List}\,A)\,(\mathsf{List}^\wedge\,A\,Q)\,(\mathsf{List}^\wedge\,A\,Q') &\end{aligned}$$

which takes a morphism f of predicates and produces a morphism of lifted predicates, is defined by

$$\mathsf{liftListMap}\,A\,Q\,Q'\,m\,\mathsf{nil}\,\mathsf{tt} = \mathsf{tt}$$

(since $x : \mathsf{List}^\wedge\,A\,Q\,\mathsf{nil}$ must necessarily be the sole inhabitant tt of $\top$), and by

$$\begin{aligned}&\mathsf{liftListMap}\,A\,Q\,Q'\,m\,(\mathsf{cons}\,a\,l')\,(y, x')\\ =\;&(m\,a\,y,\ \mathsf{liftListMap}\,A\,Q\,Q'\,m\,l'\,x')\end{aligned}$$

(since $x : \mathsf{List}^\wedge\,A\,Q\;\;(\mathsf{cons}\,a\,l')$ must be of the form $x = (y, x')$ where $y : Q\,a$ and $x' :\ \mathsf{List}^\wedge\,A\,Q\,l'$).

5 The General Framework

We can generalize the approach in Section 4 to a general framework for deriving deep induction rules for GADTs that are not truly nested GADTs. We will treat GADTs of the form

$$\begin{aligned}&\mathsf{data}\ G : \mathsf{Set}^\alpha \to \mathsf{Set}\ \mathsf{where}\\ &\quad c\ :\ \forall\{\overline{B : \mathsf{Set}}\} \to F\,G\,\overline{B} \to G(\overline{K\,\overline{B}})\end{aligned} \tag{9}$$

For brevity and clarity we indicate only one constructor c in (9), even though a GADT can have any finite number of them, each with a type of the same form as c's. In (9), F and each K in $\overline{K}$ 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 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 $\overline{K}$ (resp., $\overline{B}$) is thus α (resp., β). In addition, the type constructor F must be constructed inductively according to the following grammar:

$$\begin{aligned}F\,G\,\overline{B}\quad := \quad & F_1\,G\,\overline{B} \times F_2\,G\,\overline{B} \mid F_1\,G\,\overline{B} + F_2\,G\,\overline{B}\\ & \mid F_1\,\overline{B} \to F_2\,G\,\overline{B} \mid G\,(\overline{F_1\,\overline{B}}) \mid H\,\overline{B} \mid H\,(\overline{F_1\,G\,\overline{B}})\end{aligned}$$

This grammar is subject to the following restrictions. In the third clause the type constructor F_1 does not contain G. In the fourth clause, none of the α-many type constructors in $\overline{F_1}$ contains G. This prevents nesting, which would make it impossible to give an induction rule for G; see Section 6 below. In the fifth and sixth clauses, $H : \mathsf{Set}^\gamma \to \mathsf{Set}$ is the syntactic reflection of some functor, and thus has an associated map function. It is worth noting that the fifth clause subsumes the cases in which $F\,G\,\overline{B}$ is a closed type or one of the B_i, and that H can be the data type constructor for any (truly) nested type. From the map function for H we can also construct a map function

$$\begin{aligned}H^\wedge\mathsf{Map}\quad :\quad & \forall\overline{(A : \mathsf{Set})}\,\overline{(Q\,Q' : A \to \mathsf{Set})} \to\\ & \overline{\mathsf{PredMap}\,A\,Q\,Q'} \to\\ & \mathsf{PredMap}\,(H\,\overline{A})\,(H^\wedge\,\overline{A}\,\overline{Q})\,(H^\wedge\,\overline{A}\,\overline{Q'})\end{aligned} \tag{10}$$

for $H^\wedge$. A concrete way to define $H^\wedge\mathsf{Map}$ 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 side of the above grammar, including those in $\overline{K}$, is that they must all admit predicate liftings. This is not an overly restrictive condition, though: all GADTs constructed from the above grammar admit predicate liftings. (The fact that the domain of an arrow type is independent of G is crucial for this.) In particular, the lifting for each type constructor H is constructed using its map function. A concrete way to define more general predicate liftings is, again, by induction on the structure of the types in a suitable calculus; this will ensure that the liftings satisfy the crucial property needed to derive deep induction rules, namely that of distributing over the type constructors. We do not give a general definition of predicate liftings here, since that would require us to first design a full type calculus, which is beyond the scope of the present paper. We can however, define liftings for the type constructor F defined by the grammar on page 9 by

- $F\,G\,\overline{B} = F_1\,G\,\overline{B} \times F_2\,G\,\overline{B}$ then

  $$\begin{aligned}&F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\\ =\;&\mathsf{Pair}^\wedge\,(F_1\,G\,\overline{B})(F_2\,G\,\overline{B})\,(F_1^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})(F_2^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})\end{aligned}$$

- $F\,G\,\overline{B} = F_1\,G\,\overline{B} + F_2\,G\,\overline{B}$ then

  $$\begin{aligned}&F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\\ =\;&\mathsf{Pair}^\wedge\,(F_1\,G\,\overline{B})(F_2\,G\,\overline{B})(F_1^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})(F_2^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})\end{aligned}$$

- If $F\,G\,\overline{B} = F_1\,\overline{B} \to F_2\,G\,\overline{B}$ then

  $$\begin{aligned}&F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\,x\\ =\;&\forall(z : F_1\,\overline{B}) \to F_1^\wedge\,\overline{B}\,\overline{Q_B}\,z \to F_2^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\,(x\,z)\end{aligned}$$

- If $F\,G\,\overline{B} = G\,(F_1\,\overline{B})$ and F_1 does not contain G, then

  $$F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B} = P\,(F_1\,\overline{B})\,(F_1^\wedge\,\overline{B}\,\overline{Q_B})$$

  for all $P : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to G\,A \to \mathsf{Set}$.
- If $F\,G\,\overline{B} = H\,\overline{B}$ and H does not contain G, then

  $$F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B} = H^\wedge\,\overline{Q_B}$$

  for all $P : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to G\,A \to \mathsf{Set}$.
- If $F\,G\,\overline{B} = H\,(\overline{F_k\,G\,\overline{B}})$ and H does not contain G, then

  $$F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B} = H^\wedge\,(\overline{F_k\,G\,\overline{B}})\,(\overline{F_k^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}})$$

  for all $P : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to G\,A \to \mathsf{Set}$.

We assume in the development below that G is a unary type constructor, i.e., that $\alpha = 1$ in (9). Extending the argument to GADTs of arbitrary arity presents no difficulty other than heavier notation. In this case the type of G's single data constructor c can be rewritten as

$$c : \forall(\overline{B : \mathsf{Set}}) \to \mathsf{Equal}\,A\,(K\,\overline{B}) \to F\,G\,\overline{B} \to G\,A$$

The predicate lifting $G^\wedge : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to G\,A \to \mathsf{Set}$ for G is therefore

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

where $Q_A : A \to \mathsf{Set}$, $\overline{Q_B : B \to \mathsf{Set}}$, $e : \mathsf{Equal}\,A\,(K\,\overline{B})$, and $x : F\,G\,\overline{B}$. If we have predicate liftings

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

for F and

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

for K, then the induction hypothesis dIndC associated with c is

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

and the induction rule for G is

$$\begin{aligned}&\forall(P : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to G\,A \to \mathsf{Set}) \to \\ &\quad \mathsf{dIndC}\,P \to \forall(A : \mathsf{Set})(Q_A : A \to \mathsf{Set})(y : G\,A) \to \\ &\qquad G^\wedge\,A\,Q_A\,y \to P\,A\,Q_A\,y\end{aligned} \tag{11}$$

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

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

Here,

$$\begin{array}{lcl} cc & : & \mathsf{dIndC}\,P \\ e & : & \mathsf{Equal}\,A\,(K\,\overline{B}) \\ x & : & F\,G\,\overline{B} \\ Q_A & : & A \to \mathsf{Set} \\ \mathsf{liftE} & : & \mathsf{Equal}^\wedge\,A\,(K\,\overline{B})\,Q_A\,(K^\wedge\,\overline{B}\,\overline{Q_B})\,e \\ \mathsf{liftF} & : & F^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B}\,x \end{array}$$

and $\overline{Q_B : B \to \mathsf{Set}}$, so

$$(\overline{Q_B}, \mathsf{liftE}, \mathsf{liftF}) : G^\wedge\,A\,Q_A(c\,\overline{B}\,e\,x)$$

as expected. Finally, the morphism of predicates

$$p : \mathsf{PredMap}\,(F\,G\,\overline{B})(F^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B})(F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})$$

is defined by structural induction on F as follows:

- If $F\,G\,\overline{B} = F_1\,G\,\overline{B} \times F_2\,G\,\overline{B}$ then

  $$\begin{aligned}&F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B} \\ =\;& \mathsf{Pair}^\wedge\,(F_1\,G\,\overline{B})\,(F_2\,G\,\overline{B})(F_1^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})\,(F_2^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})\end{aligned}$$

  The induction hypothesis ensures morphisms of predicates

  $$p_1 : \mathsf{PredMap}\,(F_1\,G\,\overline{B})\,(F_1^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B Q})(F_1^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})$$

  and

  $$p_2 : \mathsf{PredMap}\,(F_2\,G\,\overline{B})\,(F_2^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B})(F_2^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})$$

  For $x_1 : F_1\,G\,\overline{B}$, $\mathsf{liftF}_1 : F_1^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B}\,x_1$, $x_2 : F_2\,G\,\overline{B}$ and $\mathsf{liftF}_2 : F_2^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B}\,x_2$ we then define

  $$p\,(x_1, x_2)\,(\mathsf{liftF}_1, \mathsf{liftF}_2) = (p_1\,x_1\,\mathsf{liftF}_1,\; p_2\,x_2\,\mathsf{liftF}_2)$$

- The case $F\,G\,\overline{B} = F_1\,G\,\overline{B} + F_2\,G\,\overline{B}$ is analogous.
- If $F\,G\,\overline{B} = F_1\,\overline{B} \to F_2\,G\,\overline{B}$ then

  $$\begin{aligned}&F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\,x \\ =\;& \forall(z : F_1\,\overline{B}) \to F_1^\wedge\,\overline{B}\,\overline{Q_B}\,z \to F_2^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\,(x\,z)\end{aligned}$$

  where $x : F\,G\,\overline{B}$. The induction hypothesis ensures a morphism of predicates

  $$p_2 : \mathsf{PredMap}\,(F_2\,G\,\overline{B})\,(F_2^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B})\,(F_2^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})$$

  We therefore define $p\,x\,\mathsf{liftF} : F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\,x$, where liftF $: F^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B}\,x$, to be

  $$p\,x\,\mathsf{liftF}\;z\;\mathsf{liftF}_1 = p_2\,(x\,z)\,(\mathsf{liftF}\,z\,\mathsf{liftF}_1)$$

  for $z : F_1\,\overline{B}$ and $\mathsf{liftF}_1 : F_1^\wedge\,\overline{B}\,\overline{Q_B}\,z$. Note that F_1 not containing G is a necessary restriction since the proof relies on $F^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B}\,x$ and $F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\,x$ having the same domain $F_1^\wedge\,\overline{B}\,\overline{Q_B}\,z$.
- If $F\,G\,\overline{B} = G\,(F_1\,\overline{B})$ and F_1 does not contain G, then

  $$F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B} = P\,(F_1\,\overline{B})\,(F_1^\wedge\,\overline{B}\,\overline{Q_B})$$

  for all $P : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to G\,A \to \mathsf{Set}$. We then define $p = \mathsf{dIndG}\,P\,cc\,(F_1\,\overline{B})\,(F_1^\wedge\,\overline{B}\,\overline{Q_B})$.
- If $F\,G\,\overline{B} = H\,\overline{B}$ and H does not contain G, then

  $$F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B} = H^\wedge\,\overline{Q_B}$$

  for all $P : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to G\,A \to \mathsf{Set}$. We therefore define

  $$p : \mathsf{PredMap}\,(H\,\overline{B})\,(H^\wedge\,\overline{B}\,\overline{Q_B})\,(H^\wedge\,\overline{B}\,\overline{Q_B})$$

  to be the identity morphism on predicates.
- If $F\,G\,\overline{B} = H\,(\overline{F_k\,G\,\overline{B}})$ and H does not contain G, then

  $$F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B} = H^\wedge\,(\overline{F_k\,G\,\overline{B}})\,(\overline{F_k^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}})$$

  for all $P : \forall(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to G\,A \to \mathsf{Set}$. Since H is not a GADT, $H^\wedge$ has a map function $H^\wedge\mathsf{Map}$ as in (10). The induction hypothesis ensures morphisms of predicates

  $$\overline{p_k : \mathsf{PredMap}\,(F_k\,G\,\overline{B})(F_k^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B})(F_k^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})}$$

  We therefore define

  $$p = H^\wedge\mathsf{Map}\,(\overline{F_k\,G\,\overline{B}})(\overline{F_k^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B}})\,(\overline{F_k^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}})\,\overline{p_k}$$

Observing that the above development essentially uses the equality GADT and its predicate lifting in the discrete category of types to extend the lifting in [13] — now specialized to the same category — to GADTs, we have established the following theorem:

Theorem 5.1. *A GADT* G *of the form in* (9) *admits the deep induction rule in* (11).

6 Truly Nested GADTs Need Not Admit Deep Induction Rules

In Sections 4 and 5 we derived deep induction rules for GADTs that are not truly nested GADTs. Since both (truly) nested types and GADTs without true nesting admit deep induction rules, we might expect truly nested GADTs to admit them as well. Unfortunately, however, the techniques developed in the previous sections do not extend to truly nested GADTs. Indeed, while the induction rule for a data type generally relies on (unary) parametricity of the model interpreting it, deep induction for a truly nested type or a truly nested GADT crucially relies on this interpretation being functorial. Whereas ADTs and nested types both admit functorial parametric semantics, proper GADTs admit parametric semantics but do not admit functorial semantics. In this section we show how the techniques developed in this paper for deriving deep induction rules go wrong for truly nested GADTs by analyzing the following very simple example:

$$\begin{array}{l} \mathsf{data}\ G : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where} \\ \qquad c :\ \forall \{A : \mathsf{Set}\} \to G\,(G\,A) \to G\,(A \times A) \end{array} \tag{12}$$

We acknowledge that G is semantically equivalent to the empty data type, and thus has a trivial (deep) induction principle. We could, of course, consider a more realistic counterexample, but this would only add notational overhead for no gain in conceptual clarity. Indeed, we need only exhibit a single GADT whose deep induction rule cannot be obtained using the techniques of this paper, and for which more robust techniques will therefore be needed if deep induction rules are to be derived for them.

To see this, we first rewrite the constructor c's type as

$$c : \forall\,(B : \mathsf{Set}) \to \mathsf{Equal}\,A\,(B \times B) \to G\,(G\,B) \to G\,A$$

The predicate lifting

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

for G is therefore

$$\begin{array}{l} G^{\wedge}\,A\,Q_A\,(c\,B\,e\,x) = \\ \quad \exists\,[Q_B]\ \mathsf{Equal}^{\wedge}\,A\,(B \times B)\,Q_A\,(\mathsf{Pair}^{\wedge}\,B\,B\,Q_B\,Q_B)\,e \\ \qquad\qquad\qquad\qquad\qquad \times\ G^{\wedge}\,(G\,B)\,(G^{\wedge}\,B\,Q_B)\,x \end{array}$$

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

$$\begin{array}{l} \lambda\,(P : \forall\,(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to G\,A \to \mathsf{Set}) \to \\ \quad \forall\,(A\ B : \mathsf{Set})\,(Q_A : A \to \mathsf{Set})\,(Q_B : B \to \mathsf{Set}) \\ \qquad\qquad\qquad\qquad (e : \mathsf{Equal}\,A\,(B \times B))\,(x : G\,(G\,B)) \to \\ \quad \mathsf{Equal}^{\wedge}\,A\,(B \times B)\,Q_A\,(\mathsf{Pair}^{\wedge}\,B\,B\,Q_B\,Q_B)\,e \to \\ \qquad\qquad\qquad\qquad P\,(G\,B)\,(P\,B\,Q_B)\,x \to P\,A\,Q_A\,(c\,B\,e\,x) \end{array}$$

so the deep induction rule for G is

$$\begin{array}{l} \forall\,(P : \forall\,(A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to G\,A \to \mathsf{Set}) \to \\ \quad \mathsf{dIndC}\,P \to \forall\,(A : \mathsf{Set})\,(Q : A \to \mathsf{Set})\,(y : G\,A) \to \\ \qquad G^{\wedge}\,A\,Q\,y \to P\,A\,Q\,y \end{array}$$

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

$$\begin{array}{rl} & \mathsf{dIndG}\,P\,cc\,A\,Q\,(c\,B\,e\,x)\,(Q',\mathsf{liftE},\mathsf{liftG}) \\ = & cc\,A\,B\,Q\,Q'\,e\,x\,\mathsf{liftE}\,p \end{array}$$

where

$$\begin{array}{lcl} cc & : & \mathsf{dIndC}\,P \\ Q & : & A \to \mathsf{Set} \\ Q' & : & B \to \mathsf{Set} \\ e & : & \mathsf{Equal}\,A\,(B \times B) \\ x & : & G\,(G\,B) \\ \mathsf{liftG} & : & G^{\wedge}\,(G\,B)\,(G^{\wedge} B\,Q')\,x \\ \mathsf{liftE} & : & \mathsf{Equal}^{\wedge}\,A\,(B \times B)\,Q\,(\mathsf{Pair}^{\wedge}\,B\,B\,Q'\,Q')\,e \end{array}$$

but we still need to define $p : P\,(G\,B)\,(P\,B\,Q')\,x$. For this we can use the induction rule and let

$$p = \mathsf{dIndG}\,P\,cc\,(G\,B)\,(P\,B\,Q')\,x\,q$$

but we still need to define

$$q : G^{\wedge}\,(G\,B)\,(P\,B\,Q')\,x$$

If we had the map function

$$\begin{array}{l} G^{\wedge}\mathsf{Map} : \forall\,(A : \mathsf{Set})\,(Q\ Q' : A \to \mathsf{Set}) \to \\ \qquad\qquad\qquad \mathsf{PredMap}\,A\,Q\,Q' \to \\ \qquad\qquad\qquad\quad \mathsf{PredMap}\,(G\,A)\,(G^{\wedge}\,A\,Q)\,(G^{\wedge}\,A\,Q') \end{array}$$

for $G^{\wedge}$ then we could define q as

$$G^{\wedge}\mathsf{Map}\,(G\,B)\,(G^{\wedge}\,B\,Q')\,(P\,B\,Q')\,(\mathsf{dIndG}\,P\,cc\,B\,Q')\,x\,\mathsf{liftG}$$

Unfortunately, however, we cannot define $G^{\wedge}\mathsf{Map}$. Indeed, its definition would have to be

$$\begin{array}{rl} & G^{\wedge}\mathsf{Map}\,A\,Q\,Q'\,m\,(c\,B\,e\,x)\,(Q_B,\mathsf{liftE},\mathsf{liftG}) \\ = & (Q'_B,\mathsf{liftE}',\mathsf{liftG}') \end{array}$$

for some

$$\begin{array}{lcl} Q'_B & : & B \to \mathsf{Set} \\ \mathsf{liftE}' & : & \mathsf{Equal}^{\wedge}\,A\,(B \times B)\,Q'\,(\mathsf{Pair}^{\wedge}\,B\,B\,Q'_B\,Q'_B)\,e \\ \mathsf{liftG}' & : & G^{\wedge}\,(G\,B)\,(G^{\wedge}\,B\,Q'_B)\,x \end{array}$$

where

$$\begin{array}{lcl} Q & : & A \to \mathsf{Set} \\ Q' & : & A \to \mathsf{Set} \\ Q_B & : & B \to \mathsf{Set} \\ m & : & \mathsf{PredMap}\,A\,Q\,Q' \\ e & : & \mathsf{Equal}\,A\,(B \times B) \\ x & : & G\,(G\,B) \\ \mathsf{liftE} & : & \mathsf{Equal}^{\wedge}\,A\,(B \times B)\,Q\,(\mathsf{Pair}^{\wedge}\,B\,B\,Q_B\,Q_B)\,e \\ \mathsf{liftG} & : & G^{\wedge}\,(G\,B)\,(G^{\wedge}\,B\,Q_B)\,x \end{array}$$

That is, we would need to produce a proof liftE' of the (extensional) equality of the predicates Q' and $\mathsf{Pair}^{\wedge}\,B\,B\,Q'_B\,Q'_B$ from just a proof liftE of the (extensional) equality of the predicates Q and $\mathsf{Pair}^\wedge\ \mathsf{B}\ \mathsf{B}\ \mathsf{Q_B}\ \mathsf{Q_B}$ 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 $\mathsf{Pair}^\wedge\ \mathsf{B}\ \mathsf{B}\ \mathsf{Q_B}\ \mathsf{Q_B}$ 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 [10], 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'. And not being able to define

$$\begin{array}{l} \mathsf{Equal^\wedge Map} : \forall (\mathsf{A}\ \mathsf{B} : \mathsf{Set}) \\ \qquad\qquad (\mathsf{Q_A}\ \mathsf{Q}'_\mathsf{A} : \mathsf{A} \to \mathsf{Set}) \\ \qquad\qquad (\mathsf{Q_B}\ \mathsf{Q}'_\mathsf{B} : \mathsf{B} \to \mathsf{Set}) \to \\ \quad \mathsf{PredMap}\ \mathsf{A}\ \mathsf{Q_A}\ \mathsf{Q}'_\mathsf{A} \to \\ \quad \mathsf{PredMap}\ \mathsf{B}\ \mathsf{Q_B}\ \mathsf{Q}'_\mathsf{B} \to \\ \quad \mathsf{PredMap}\ (\mathsf{Equal}\ \mathsf{A}\ \mathsf{B})\ (\mathsf{Equal^\wedge A}\ \mathsf{B}\ \mathsf{Q_A}\ \mathsf{Q_B}) \\ \quad (\mathsf{Equal^\wedge A}\ \mathsf{B}\ \mathsf{Q}'_\mathsf{A}\ \mathsf{Q}'_\mathsf{B}) \end{array}$$

of course makes it unclear how to define $\mathsf{G^\wedge Map}$ for more general G.

7 Case Study: Extracting Types of Lambda Terms

In this section, we use deep induction for the LTerm GADT from Figure 2 to infer the type from a lambda term. The following predicate either returns the type of its input lambda term if that type can be inferred or indicates that the type inference fails:

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

Of course, since GetType is (trivially) defined by structural induction, we could perform type inference using hand-threaded applications of structural induction as observed at the end of Section 2. Nevertheless, the example as given nicely illustrates deep induction.

By construction every lambda term in LTerm is well-typed, but that (necessarily unique) type cannot always be inferred. The predicate GetType uses the standard Maybe data type to represent failure of type inference. It is defined by:

$$\begin{array}{l} \mathsf{data\ Maybe} : \mathsf{Set} \to \mathsf{Set\ where} \\ \quad \mathsf{nothing} : \forall \{\mathsf{A} : \mathsf{Set}\} \to \mathsf{Maybe}\ \mathsf{A} \\ \quad \mathsf{just} \quad\ : \forall \{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{Maybe}\ \mathsf{A} \end{array} \tag{13}$$

We want to show that $\mathsf{GetType}\ \mathsf{A}\ \mathsf{t}$ is satisfied by every element t in $\mathsf{LTerm}\ \mathsf{A}$, 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 can be proved with deep induction, which is used to apply the induction hypothesis to the individual terms in the list of terms that the data constructor list takes as an argument. Indeed, using the deep induction rule dIndLTerm from Section 4.3 we can define getTypeProof by

$$\begin{array}{rl} & \mathsf{getTypeProof}\ \mathsf{A}\ \mathsf{t} \\ = & \mathsf{dIndLTerm}\ \mathsf{P}\ \mathsf{cvar}\ \mathsf{cabs}\ \mathsf{capp}\ \mathsf{clist}\ \mathsf{A}\ \mathsf{K_\top}\ \mathsf{t}\ (\mathsf{LTerm^\wedge KT}\ \mathsf{A}\ \mathsf{t}) \end{array}$$

where $\mathsf{t} : \mathsf{LTerm}\ \mathsf{A}$, P is the polymorphic predicate

$$\lambda\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{t} : \mathsf{LTerm}\ \mathsf{A}) \to \mathsf{Maybe}\,(\mathsf{LType}\ \mathsf{A})$$

and $\mathsf{K_\top}$ is the constantly $\top$-valued predicate on A, and

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

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 given in Figure 7. In the first clause, cvar returns $\mathsf{just}\ \mathsf{T_A}$. In the second clause, cabs returns nothing if its final argument is nothing and

$$\begin{array}{rl} & \mathsf{cabs}\ \mathsf{A}\ \mathsf{B}\ \mathsf{C}\ \mathsf{Q_A}\ \mathsf{Q_B}\ \mathsf{Q_C}\ \mathsf{e}\ \mathsf{s}\ \mathsf{T_B}\ \mathsf{t_C}\ \mathsf{liftE}\ \mathsf{lift_{T_B}}\ (\mathsf{just}\ \mathsf{T_C}) \\ = & \mathsf{just}\,(\mathsf{arr}\ \mathsf{B}\ \mathsf{C}\ \mathsf{e}\ \mathsf{T_B}\ \mathsf{T_C}) \end{array}$$

otherwise. In the third clause,

$$\begin{array}{rl} & \mathsf{capp}\ \mathsf{A}\ \mathsf{B}\ \mathsf{Q_A}\ \mathsf{Q_B}\ \mathsf{t_{BA}}\ \mathsf{t_A}\ (\mathsf{just}\ (\mathsf{arr}\ \mathsf{B}\ \mathsf{A}\ \mathsf{refl}\ \mathsf{T_B}\ \mathsf{T_A}))\ \mathsf{mb} \\ = & \mathsf{just}\ \mathsf{T_A} \end{array}$$

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 $\mathsf{ts} = \mathsf{nil}$ we define

$$\mathsf{clist}\ \mathsf{A}\ \mathsf{B}\ \mathsf{Q}\ \mathsf{Q}'\ \mathsf{e}\ \mathsf{nil}\ \mathsf{liftE}\ \mathsf{lift_{ts}} = \mathsf{nothing}$$

where $\mathsf{liftE} : \mathsf{Equal}^\wedge\ \mathsf{A}\,(\mathsf{List}\ \mathsf{B})\ \mathsf{Q}\ (\mathsf{List}^\wedge\ \mathsf{B}\ \mathsf{Q}')\ \mathsf{e}$, and $\mathsf{lift_{ts}} : \mathsf{List}^\wedge\ (\mathsf{LTerm}\ \mathsf{B})\,(\mathsf{GetType}\ \mathsf{B})\,\mathsf{ts}$. When $\mathsf{ts} = \mathsf{cons}\ \mathsf{t}\ \mathsf{ts}'$ the type of $\mathsf{lift_{ts}}$ becomes

$$\begin{array}{rl} & \mathsf{List}^\wedge\ (\mathsf{LTerm}\ \mathsf{B})\,(\mathsf{GetType}\ \mathsf{B})\,(\mathsf{cons}\ \mathsf{t}\ \mathsf{ts}') \\ = & \mathsf{GetType}\ \mathsf{B}\ \mathsf{t} \times \mathsf{List}^\wedge\ (\mathsf{LTerm}\ \mathsf{B})\,(\mathsf{GetType}\ \mathsf{B})\ \mathsf{ts}' \\ = & \mathsf{Maybe}\,(\mathsf{LType}\ \mathsf{B}) \times \mathsf{List}^\wedge\ (\mathsf{LTerm}\ \mathsf{B})\,(\mathsf{GetType}\ \mathsf{B})\ \mathsf{ts}' \end{array}$$

We pattern match on the first component of the pair to define

$$\begin{array}{l} \mathsf{clist}\ \mathsf{A}\ \mathsf{B}\ \mathsf{Q}\ \mathsf{Q}'\ \mathsf{e}\ (\mathsf{cons}\ \mathsf{t}\ \mathsf{ts}')\ \mathsf{liftE}\ (\mathsf{nothing}, \mathsf{lift_{ts'}}) = \mathsf{nothing} \\ \mathsf{clist}\ \mathsf{A}\ \mathsf{B}\ \mathsf{Q}\ \mathsf{Q}'\mathsf{e}\,(\mathsf{cons}\,\mathsf{t}\,\mathsf{ts}')\,\mathsf{liftE}\,(\mathsf{just}\ \mathsf{T}', \mathsf{lift_{ts'}}) = \mathsf{just}(\mathsf{list}\ \mathsf{B}\ \mathsf{e}\ \mathsf{T}') \end{array}$$

Here $\mathsf{e} : \mathsf{Equal}\ \mathsf{A}\,(\mathsf{List}\ \mathsf{B})$, $\mathsf{T}' : \mathsf{LType}\ \mathsf{B}$, and

$$\mathsf{lift_{ts'}} : \mathsf{List}^\wedge\ (\mathsf{LTerm}\ \mathsf{B})\,(\mathsf{GetType}\ \mathsf{B})\,\mathsf{ts}'$$

To finish defining getTypeProof we still need a proof

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

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 $\mathsf{T} : \mathsf{LType}\ \mathsf{A}$ we define

$$\mathsf{LTerm^\wedge KT}\ \mathsf{A}\,(\mathsf{var}\ \mathsf{s}\ \mathsf{T}) = \mathsf{LType^\wedge KT}\ \mathsf{A}\ \mathsf{T}$$

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

Figure 7. Applied induction hypotheses for LTerm

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

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

where $\mathsf{K_T : A \to Set}$, $\mathsf{Q_B : B \to Set}$, and $\mathsf{Q_C : C \to Set}$. The only reasonable choice is to let both $\mathsf{Q_B}$ and $\mathsf{Q_C}$ be $\mathsf{K_T}$, which means we need proofs of $\mathsf{Equal}^{\wedge}\,\mathsf{A\,(B \to C)\,K_T\,(Arr^{\wedge}\,B\,C\,K_T\,K_T)\,e}$, $\mathsf{LType}^{\wedge}\,\mathsf{B\,K_T\,T}$ and $\mathsf{LTerm}^{\wedge}\,\mathsf{C\,K_T\,t'}$. 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 $\mathsf{Arr}^{\wedge}\,\mathsf{B\,C\,K_T\,K_T}$ of $\mathsf{K_T}$ to arrow types is not identical to $\mathsf{K_T}$ on arrow types but rather (extensionally) isomorphic. We discuss this issue in more detail at the end of the section, but for now we simply assume a proof

$$\begin{array}{rl} & \mathsf{Equal^{\wedge}ArrKT} \\ : & \mathsf{Equal}^{\wedge}\,\mathsf{A\,(B \to C)\,K_T\,(Arr^{\wedge}\,B\,C\,K_T\,K_T)\,e} \end{array}$$

and define

$$\begin{array}{rl} & \mathsf{LTerm^{\wedge}KT\,A\,(abs\,B\,C\,e\,s\,T\,t')} \\ = & (\mathsf{K_T}, \mathsf{K_T}, \mathsf{Equal^{\wedge}ArrKT}, \\ & \qquad \mathsf{LType^{\wedge}KT\,B\,T}, \mathsf{LTerm^{\wedge}KT\,C\,t'}) \end{array}$$

- If $\mathsf{t_1 : LTerm\,(B \to A)}$ and $\mathsf{t_2 : LTerm\,B}$ then, by the same reasoning as in the previous case, we need to define

$$\begin{array}{rl} & \mathsf{LTerm^{\wedge}KT\,A\,(app\,B\,t_1\,t_2)} \\ : & \mathsf{LTerm}^{\wedge}\,\mathsf{(B \to A)\,(Arr^{\wedge}\,B\,A\,K_T\,K_T)\,t_1} \times \\ & \qquad\qquad\qquad\qquad \mathsf{LTerm}^{\wedge}\,\mathsf{B\,K_T\,t_2} \end{array}$$

We define the second component of the pair to be $\mathsf{LTerm^{\wedge}KT\,B\,t_2}$. We define the first component from a proof of $\mathsf{LTerm}^{\wedge}\,\mathsf{(B \to A)\,K_T\,t_1}$ and the function

$$\begin{array}{rl} & \mathsf{LTerm^{\wedge}EqualMap} \\ : & \forall\,(\mathsf{A : Set})\,(\mathsf{Q\,Q' : A \to Set}) \to \\ & \ \mathsf{Equal}^{\wedge}\,\mathsf{A\,A\,Q\,Q'\,refl} \to \\ & \quad \mathsf{PredMap\,(LTerm\,A)\,(LTerm^{\wedge}\,A\,Q)\,(LTerm^{\wedge}\,A\,Q')} \end{array}$$

that takes two (extensionally) equal predicates with the same carrier and produces a morphism of predicates between their liftings. We define $\mathsf{LTerm^{\wedge}EqualMap}$ straightforwardly by pattern matching on the first two arguments to PredMap in its return type, using transitivity and symmetry of the type constructor Equal, together with the two analogously defined functions $\mathsf{LType^{\wedge}EqualMap}$ and $\mathsf{Arr^{\wedge}EqualMap}$ in the case the first argument to PredMap is constructed using var and app, respectively. If $\mathsf{L_{K_T} : LTerm^{\wedge}\,(B \to A)\,K_T\,t_1}$ is the proof $\mathsf{L_{K_T}} = \mathsf{LTerm^{\wedge}KT\,(B \to A)\,t_1}$ and $\mathsf{LTerm^{\wedge}Arr}$: $\mathsf{LTerm}^{\wedge}\,\mathsf{(B \to A)\,(Arr^{\wedge}\,B\,A\,K_T\,K_T)\,t_1}$ is the proof

$$\begin{array}{rl} \mathsf{LTerm^{\wedge}Arr} = & \mathsf{LTerm^{\wedge}EqualMap\,K_T\,(Arr^{\wedge}\,B\,A\,K_T\,K_T)} \\ & \qquad\qquad \mathsf{Equal^{\wedge}ArrKT\,t_1\,L_{K_T}} \end{array}$$

then we define

$$\begin{array}{rl} & \mathsf{LTerm^{\wedge}KT\,A\,(app\,B\,t_1\,t_2)} \\ = & (\mathsf{K_T}, \mathsf{LTerm^{\wedge}Arr}, \mathsf{LTerm^{\wedge}KT\,B\,t_2}) \end{array}$$

- If $\mathsf{e : Equal\,A\,(List\,B)}$ and $\mathsf{ts : List\,(LTerm\,B)}$ then, as above, we need to define

$$\begin{array}{rl} & \mathsf{LTerm^{\wedge}KT\,A\,(list\,B\,e\,ts)} \\ : & \mathsf{Equal}^{\wedge}\,\mathsf{A\,(List\,B)\,K_T\,(List^{\wedge}\,B\,K_T)\,e} \times \\ & \quad \mathsf{List}^{\wedge}\,\mathsf{(LTerm\,B)\,(LTerm^{\wedge}\,B\,K_T)\,ts} \end{array}$$

As in that case we assume a proof

$$\mathsf{Equal^{\wedge}ListKT : Equal^{\wedge}\,A\,(List\,B)\,K_T\,(List^{\wedge}\,B\,K_T)\,e}$$

for the first component. We can define the second component using liftListMap from Section 4.3 to map a morphism $\mathsf{PredMap\,(LTerm\,B)\,(K_T)\,(LTerm^{\wedge}\,B\,K_T)}$ of predicates to a morphism $\mathsf{PredMap\,(List\,(LTerm\,B))}$ $\mathsf{(List^{\wedge}\,(LTerm\,B)\,K_T)(List^{\wedge}\,(LTerm\,B)(LTerm^{\wedge}\,B\,K_T))}$ of lifted predicates. Taking

$$\mathsf{m_{K_T} : PredMap\,(LTerm\,B)\,(K_T)\,(LTerm^{\wedge}\,B\,K_T)}$$

to be the proof

$$\mathsf{m_{K_T}\,t'\,tt} = \mathsf{LTerm^{\wedge}KT\,B\,t'}$$

where $\mathsf{t' : LTerm\,B}$ and tt is the single element of $\mathsf{K_T\,t'}$, and taking

$$\mathsf{L_{List^{\wedge}LTerm^{\wedge}KT} : List^{\wedge}\,(LTerm\,B)\,(LTerm^{\wedge}\,B\,K_T)\,ts}$$

to be the proof

$$\begin{array}{rl} & \mathsf{L_{List^{\wedge}LTerm^{\wedge}KT}} \\ = & \mathsf{liftListMap\,(LTerm\,B)\,K_T\,(LTerm^{\wedge}\,B\,K_T)\,m_{K_T}\,ts} \\ & \quad \mathsf{(List^{\wedge}KT\,(LTerm\,B)\,ts)} \end{array}$$

we define

$$\begin{aligned}&\mathsf{LTerm}^\wedge\mathsf{KT}\;\mathsf{A}\;(\mathsf{list}\;\mathsf{B}\;\mathsf{e}\;\mathsf{ts})\\ =\;&(\mathsf{K}_\top,\;\mathsf{Equal}^\wedge\mathsf{ListKT},\;\mathsf{L}_{\mathsf{List}^\wedge\mathsf{LTerm}^\wedge\mathsf{KT}})\end{aligned}$$

The above techniques can be used to define a function $\mathsf{G}^\wedge\mathsf{KT} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{x} : \mathsf{G}\,\mathsf{A}) \to \mathsf{G}^\wedge\;\mathsf{A}\;\mathsf{K}_\top\;\mathsf{x}$ for any GADT G as defined in Section 5. To provide a proof of $\mathsf{G}^\wedge\;\mathsf{A}\;\mathsf{K}_\top\;\mathsf{x}$ for every term $\mathsf{x} : \mathsf{G}\,\mathsf{A}$, we need to know that, if G has a constructor $\mathsf{c} : \mathsf{H}\,(\overline{\mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}}}) \to \mathsf{G}\,(\overline{\mathsf{K}\,\overline{\mathsf{B}}})$, then H cannot construct a GADT so the generalization $\mathsf{H}^\wedge\mathsf{Map}$ of 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 $\mathsf{Pair}^\wedge\;\mathsf{A}\;\mathsf{B}\;\mathsf{K}_\top\;\mathsf{K}_\top$ 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

$$\mathsf{Pair}^\wedge\;\mathsf{A}\;\mathsf{B}\;\mathsf{K}_\top\;\mathsf{K}_\top(\mathsf{a}, \mathsf{b}) = \mathsf{K}_\top\;\mathsf{a} \times \mathsf{K}_\top\;\mathsf{b} = \top \times \top$$

whereas $\mathsf{K}_\top\;(\mathsf{a}, \mathsf{b}) = \top$. While these types are not equal, they are clearly isomorphic. Similar isomorphisms between $\mathsf{F}^\wedge\;\mathsf{A}\;\mathsf{K}_\top$ 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 $\mathsf{F}^\wedge\;\mathsf{A}\;\mathsf{K}_\top = \mathsf{K}_\top$ 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 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 GADTs that are not truly nested GADTs. It also shows that truly nested GADTs do not obviously admit (deep) induction rules. Our development is implemented in Agda, as is our case study from Section 7. Our development opens the way to incorporating automatic generation of (deep) induction rules for them into proof assistants.

Acknowledgments

This work was supported by National Science Foundation award 1906388.

References

[1] R. Atkey. 2012. Relational parametricity for higher kinds. In *Computer Science Logic*. 46–61.

[2] E. S. Bainbridge, P. Freyd, A. Scedrov, and P. J. Scott. 1990. Functorial polymorphism. *Theoretical Computer Science* 70(1) (1990), 35–64. https://doi.org/10.1016/0304-3975(90)90151-7

[3] R. Bird and L. Meertens. 1998. Nested datatypes. In *Mathematics of Program Construction*. 52–67. https://doi.org/10.1007/BFb0054285

[4] J. Cheney and R. Hinze. 2003. First-class phantom types. (2003). CUCIS TR2003-1901, Cornell University.

[5] P. Dybjer. 1994. Inductive families. *Formal Aspects of Computing* 6(4) (1994), 440–465. https://doi.org/10.1007/BF01211308

[6] P. Fu and P. Selinger. 2018. Dependently typed folds for nested data types. (2018). https://arxiv.org/abs/1806.05230

[7] N. Ghani, P. Johann, F. Nordvall Forsberg, F. Orsanigo, and T. Revell. 2015. Bifibrational functorial semantics for parametric polymorphism. In *Mathematical Foundations of Program Semantics*. 165–181. https://doi.org/10.1016/j.entcs.2015.12.011

[8] R. Hinze. 2003. Fun with phantom types. In *The Fun of Programming*. 245–262.

[9] P. Johann and N. Ghani. 2008. Foundations for Structured Programming with GADTs. In *Proceedings, Principles of Programming Languages*. 297–308. https://doi.org/10.1145/1328438.1328475

[10] P. Johann, E. Ghiorzi, and D. Jeffries. 2021. GADTs, functoriality, parametricity: Pick two. In *Logical and Semantic Frameworks with Applications*.

[11] P. Johann, E. Ghiorzi, and D. Jeffries. 2021. Parametricity for primitive nested types. In *Foundations of Software Science and Computation Structures*. 324–343. https://doi.org/10.1007/978-3-030-71995-1_17

[12] P. Johann and A. Polonsky. 2019. Higher-kinded data types: Syntax and semantics. In *Logic in Computer Science*. 1–13. https://doi.org/10.1109/LICS.2019.8785657

[13] P. Johann and A. Polonsky. 2020. Deep induction: Induction rules for (truly) nested types. In *Foundations of Software Science and Computation Structures*. 339–358. https://doi.org/10.1007/978-3-030-45231-5_18

[14] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. 2006. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming*. 50–61. https://doi.org/10.1145/1160074.1159811

[15] S. Mac Lane. 1971. *Categories for the Working Mathematician*. Springer.

[16] C. McBride. 1999. Dependently Typed Programs and their Proofs. (1999). PhD thesis, University of Edinburgh.

[17] Y. Minsky. 2015. Why GADTs Matter for Performance. (2015). https://blogs.janestreet.com/why-gadts-matter-for-performance

[18] E. Pasalic and N. Linger. 2004. Meta-programming with typed object-language representations. In *Generic Programming and Component Engineering*. 136–167. https://doi.org/10.1007/978-3-540-30175-2_8

[19] F. Pottier and Y. Régis-Gianas. 2006. Stratified type inference for generalized algebraic data types. In *Principles of Programming Languages*. 232–244. https://doi.org/10.1145/1111320.1111058

[20] D. Roundy. 2006. Implementing the darcs Patch Formalism ...and Verifying It. (2006). https://physics.oregonstate.edu/~roundyd/talks/fosdem

[21] T. Schrijvers, S. L. Peyton Jones, M. Sulzmann, and D. Vytiniotis. 2009. Complete and decidable type inference for GADTs. In *International Conference on Functional Programming*. 341–352. https://doi.org/10.1145/1631687.1596599

[22] T. Sheard and E. Pasalic. 2004. Meta-programming with built-in type equality. In *Workshop on Logical Frameworks and Meta-languages*. 106–124.

[23] E. Tassi. 2019. Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. In *Interactive Theorem Proving*. 1–18. https://doi.org/10.4230/LIPIcs.ITP.2019.29

[24] The Coq Development Team. 2020. The Coq Proof Assistant, version 8.11.0. (2020). https://doi.org/10.5281/zenodo.3744225

[25] M. Ullrich. 2020. Generating Induction Principles for Nested Induction Types in MetaCoq. (2020). PhD thesis, Saarland University.

[26] D. Vytiniotis and S. Weirich. 2010. Parametricity, type equality, and higher-order polymorphism. *Journal of Functional Programming* 20(2) (2010), 175–210. https://doi.org/10.1017/S0956796810000079

[27] H. Xi, C. Chen, and G. Chen. 2003. Guarded recursive datatype constructors. In *Principles of Programming Languages*. 224–235. https://doi.org/10.1145/604131.604150

[28] N. Zilberstein. 2015. CIS194 homepage. (2015). https://www.seas.upenn.edu/~cis194/spring15/lectures/11-stlc.html