(Deep) Induction for GADTs

Patricia Johann Enrico Ghiorzi Daniel Jeffries

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

Abstract

Deep data types are data types that are defined in terms of other such data types, including, in the case of truly nested types, themselves. Deep induction is an extension of structural induction that traverses *all* of the structure present in a structure of such a type, propagating suitable predicates to *all* of the data contained in that structure. Deep induction has been shown to be the form of induction most suitable for applications involving deep nested types. In this paper we show how to extend deep induction to a robust class of deep GADTs that are not truly nested. We also show that it cannot be extended to truly nested GADTs.

1 Introduction

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

The data types handled by Coq are (possibly mutually inductive) polynomial ADTs, and the induction rules automatically generated for them are the expected ones for standard structural induction. It has long been understood, however, that these rules are too weak to be genuinely useful for so-called *deep ADTs* [15], i.e., ADTs that are (possibly mutually inductively) defined in terms of (other) such ADTs. [1] Consider, for example, the following type of rose trees, here coded in Agda and defined in terms of the standard type of lists:

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

The induction rule Coq automatically generates for rose trees is

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

Unfortunately, this is neither the induction rule we intuitively expect, nor is it expressive enough to prove even basic properties of rose trees that ought to be amenable to inductive proof. What is needed here is an enhanced notion of induction that, when specialized to rose trees, will propagate the predicate P through the outer list structure and to the rose trees sitting inside node's list argument. More generally, this enhanced notion of induction should traverse *all* of the structure present in a data element, propagating suitable predicates to *all* of the data contained in the structure. With data types becoming ever more advanced, and with deeply structured such types becoming ever more ubiquitous in formalizations, it is critically important that proof assistants be able to automatically generate genuinely useful induction rules for data types that go well beyond traditional ADTs. Such data types include (truly) nested types [3] [2], generalized algebraic data types (GADTs) [4,21,24,27], more richly indexed families [5], and deep variants of all of these.

[1] Such data types are called nested inductive types by Chlipala [6], reflecting the fact that "inductive type" means "ADT" in Coq.

[2] A truly nested type is a nested type that is defined over itself.

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

Deep induction [15] is a generalization of structural induction that fits this bill exactly. Whereas structural induction rules induct over only the top-level structure of data, leaving any data internal to the top-level structure untouched, deep induction rules induct over *all* of the structured data present. The key idea is to parameterize induction rules not just over a predicate over the top-level data type being considered, but also over additional custom predicates on the types of primitive data they contain. These custom predicates are then lifted to predicates on any internal structures containing these data, and the resulting predicates on these internal structures are lifted to predicates on any internal structures containing structures at the previous level, and so on, until the internal structures at all levels of the data type definition, including the top level, have been so processed. Satisfaction of a predicate by the data at one level of a structure is then conditioned upon satisfaction of the appropriate predicates by *all* of the data at the preceding level.

Deep induction was shown in [15] to be the form of induction most appropriate to nested types (including ADTs) that are defined over, or mutually recursively with, other such types (including, possibly, themselves). Deep induction delivers the following genuinely useful induction rule for rose trees:

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

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

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

$$\begin{aligned}&\mathsf{data\,Seq\,(a : Set) : Set\,where}\\ &\quad\mathsf{const} : \mathsf{a} \to \mathsf{Seq\,a}\\ &\quad\mathsf{pair} : \mathsf{Seq\,a} \to \mathsf{Seq\,b} \to \mathsf{Seq\,(a \times b)}\end{aligned} \tag{2}$$

of sequences only constructs sequences of pairs, rather than sequences of arbitrary type, as does const. If all of the constructors for a GADT G return structured instances of G, then some of G's instances might not be inhabited. GADTs therefore have two distinct, but equally natural, semantics: a functorial semantics interpreting them as left Kan extensions [16], and a parametric semantics interpreting them as their Church encodings [1,26]. As explained in [13], a key difference in the two semantics is that the former views GADTs as their *functorial completions* [14], and thus as containing more data than just those expressible in syntax. By contrast, the latter views them as what might be called *syntax-only* GADTs. Happily, these two views of GADTs coincide for those that are ADTs or other nested types. However, both they and their attendant properties differ greatly for proper GADTs. In fact, the views deriving from the functorial and parametric semantics for proper GADTs are sufficiently distinct that, by contrast with the situation for ADTs and other nested types [2,9,12], it is not actually possible to define a functorial parametric semantics for them [13].

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

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

the structural induction rule for any GADT can be recovered from its deep induction rule simply by taking the custom predicates in its deep induction rule to be constantly True-valued predicates.

Deep induction rules for GADTs cannot, however, be derived by somehow extending the techniques of [15] to syntax-only GADTs. Indeed, the derivation of induction rules given there makes crucial use of the functoriality of data types' interpretations from [14], and that is precisely what the interpretation of GADTs as their Church encodings fails to deliver. Instead, we first give a predicate lifting styled after those of [15], together with a (deep) induction rule, and for the simplest — and arguably most important — GADT, namely the equality GADT. (See Section 4.1.) We can then derive the deep induction rule for any other GADT G by *i*) using the equality GADT to represent G as its so-called *Henry Ford encoding* [4,10,17,23,24], and *ii*) using the predicate liftings for the equality GADT and any other GADTs appearing in the definition of G to appropriately thread the custom predicates for the primitive types appearing in G through its structure. This two-step process delivers deep induction rules for a broad class of deep GADTs. In Section 3 we introduce a series of increasingly complex GADTs as running examples, and in Section 4 we derive a deep induction rule for each of them. In particular, we derive the deep induction rule for Seq in Section 4.2. We present our general framework for deriving (deep) induction rules for (deep) GADTs in Section 5, and observe that the derivations in Section 4 are all instances of it. In Section 6 we show that, by contrast with truly nested types, which do have a functorial semantics, syntax-only GADTs' lack of functoriality means that it is not possible to extend induction — deep or otherwise — to truly nested GADTs. This does not appear to be much of a restriction, however, since GADTs defined over themselves do not, to our knowledge, appear in applications or the literature.

All of the deep induction rules appearing in this paper have been derived using our general framework. Our Agda code implementing them is available at [11].

Related Work Various techniques for deriving induction rules for data types that go beyond ADTs have been studied. For example, Fu and Selinger [8] show, via examples, how to derive induction rules for arbitrary nested types. Unfortunately, however, their technique is rather *ad hoc*, so is unclear how to generalize it to nested types other than the specific ones in the examples. Moreover, it 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 [25] has done exactly that, deriving induction rules for data type definitions in Coq using unary parametricity. Tassi's technique seems to be essentially equivalent to that of [14] for nested types, although he does not permit true nesting. More recently, Ulrich [28] has implemented a plugin in MetaCoq to generate induction rules for nested types. This plugin is also based on unary parametricity and, again, true nesting is not permitted. As far as we know, no attempt has yet been made to extend either implementation to GADTs. In fact, we know of no work other than that reported here that specifically addresses induction rules for (deep) GADTs.

2 Deep induction for ADTs and nested types

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

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

is

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

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

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

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

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

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

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

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

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

Here, $\mathsf{Pair}^\wedge : \forall (\mathsf{A}\,\mathsf{B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to (\mathsf{B} \to \mathsf{Set}) \to \mathsf{A} \times \mathsf{B} \to \mathsf{Set}$ lifts predicates Q_A on data of type A and Q_B on data of type B to a predicate on pairs of type $\mathsf{A} \times \mathsf{B}$ in such a way that $\mathsf{Pair}^\wedge\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}_\mathsf{A}\,\mathsf{Q}_\mathsf{B}\,(\mathsf{a}, \mathsf{b}) = \mathsf{Q}_\mathsf{A}\,\mathsf{a} \times \mathsf{Q}_\mathsf{B}\,\mathsf{b}$. Similarly, $\mathsf{PTree}^\wedge : \forall (\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{PTree}\,\mathsf{A} \to \mathsf{Set}$ lifts a predicate Q on data of type A to a predicate on data of type $\mathsf{PTree}\,\mathsf{A}$ asserting that Q holds for every element of type A contained in its perfect tree argument.

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

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

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

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

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

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

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

3 (Deep) GADTs

While a data constructor for a nested type can take *as arguments* data whose types involve instances of that type at indices other than the one being defined, its return type must still be at the (variable) type instance being defined. For example, every data constructor for $\mathsf{PTree}\,\mathsf{A}$ must return an element of type $\mathsf{PTree}\,\mathsf{A}$, regardless of the instances of PTree appearing in the types of its arguments. GADTs relax this restriction, allowing their data constructors both to take as arguments *and return as results* data whose types involve instances of them other than the one being defined. And as with the return type of pair in (2), these instances can be structured.

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

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

is parametrized by two type indices, but it is only possible to construct data elements of type $\mathsf{Equal}\,\mathsf{a}\,\mathsf{b}$ if a and b are instantiated at the same type. If the types a and b are syntactically identical then the type $\mathsf{Equal}\,\mathsf{a}\,\mathsf{b}$ contains the single data element refl. It contains no data elements otherwise.

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

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

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

Neither Equal nor Seq is a deep GADT, but the following GADT LTerm, which encodes terms of a simply typed lambda calculus, is. More robust variations on LTerm are, of course, possible. But since this variation is rich enough to illustrate all essential aspects of deep GADTs — and later, in Section 4.3, their deep induction rules — while still being small enough to ensure clarity of exposition, we keep it to a minimum.

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

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

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

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

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

4 (Deep) induction for GADTs

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

4.1 (Deep) induction for Equal

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

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

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

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

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

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

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

4.2 (Deep) induction for Seq

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

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

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

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

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

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

Fig. 1. Predicate liftings for LType and LTerm

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

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

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

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

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

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

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

4.3 (Deep) induction for LTerm

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

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

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

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

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

Fig. 2. dIndLTerm

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

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

The deep induction rule for LTerm is thus

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

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

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

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

5 The general framework

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

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

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

$$\mathsf{F\,G\,\overline{B}} \;:=\; \mathsf{F_1\,G\,\overline{B} \times F_2\,G\,\overline{B} \mid F_1\,G\,\overline{B} + F_2\,G\,\overline{B} \mid F_1\,\overline{B} \to F_2\,G\,\overline{B} \mid G\,(\overline{F_1\,\overline{B}}) \mid H\,\overline{B} \mid H\,(\overline{F_1\,G\,\overline{B}})}$$

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

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

for $\mathsf{H^\wedge}$. A concrete way to define $\mathsf{H^\wedge Map}$ 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 types appearing above, including the Ks in (11), is that they must all admit predicate liftings. This is not an overly restrictive condition, though: all types constructed from sums, products, arrow types and type application admit predicate liftings, and so do GADTs constructed from the above grammar; in fact, we have seen such liftings for products and type application in Section 4. A concrete way to define predicate liftings more generally is, again, by induction on the the structure of the types. We do not give a general definition of predicate liftings here, though, since that would require us to first design a full type calculus, which is beyond the scope of the present paper.

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

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

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

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

for F and for K, respectively, the induction hypothesis for c is

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

and the induction rule for G is

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

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

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

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

- If $\mathsf{F\,G\,\overline{B} = F_1\,G\,\overline{B} \times F_2\,G\,\overline{B}}$ then $\mathsf{F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B} = Pair^\wedge\,(F_1\,G\,\overline{B})\,(F_2\,G\,\overline{B})\,(F_1^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})\,(F_2^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})}$. The induction hypothesis ensures morphisms of predicates $\mathsf{p_1 : PredMap\,(F_1\,G\,\overline{B})\,(F_1^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B Q})(F_1^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})}$ and $\mathsf{p_2 : PredMap\,(F_2\,G\,\overline{B})\,(F_2^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B})(F_2^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})}$. For $\mathsf{x_1 : F_1\,G\,\overline{B}}$, $\mathsf{liftF_1 : F_1^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B}\,x_1}$, $\mathsf{x_2 : F_2\,G\,\overline{B}}$ and $\mathsf{liftF_2 : F_2^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B}\,x_2}$ we therefore define $\mathsf{p\,(x_1, x_2)\,(liftF_1, liftF_2) = (p_1\,x_1\,liftF_1,\; p_2\,x_2\,liftF_2)}$.

- The case $\mathsf{F\,G\,\overline{B} = F_1\,G\,\overline{B} + F_2\,G\,\overline{B}}$ is analogous.
- If $\mathsf{F\,G\,\overline{B} = F_1\,\overline{B} \to F_2\,G\,\overline{B}}$ then $\mathsf{F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\,x = \forall(z : F_1\,\overline{B}) \to F_1^\wedge\,\overline{B}\,\overline{Q_B}\,z \to F_2^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\,(x\,z)}$, where $\mathsf{x : F\,G\,\overline{B}}$. The induction hypothesis ensures a morphism of predicates $\mathsf{p_2 : PredMap\,(F_2\,G\,\overline{B})\,(F_2^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B})\,(F_2^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})}$. We define $\mathsf{p\,x\,liftF : F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\,x}$, where $\mathsf{liftF : F^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B}\,x}$ is defined by $\mathsf{p\,x\,liftF\,z\,liftF_1 = p_2(x\,z)(liftF\,z\,liftF_1)}$ for $\mathsf{z : F_1\,\overline{B}}$ and $\mathsf{liftF_1 : F_1^\wedge\,\overline{B}\,\overline{Q_B}\,z}$. Note that $\mathsf{F_1}$ not containing the recursive variable is a necessary restriction, since the proof relies on $\mathsf{F^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B}\,x}$ and $\mathsf{F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}\,x}$ having the same domain $\mathsf{F_1^\wedge\,\overline{B}\,\overline{Q_B}\,z}$.
- If $\mathsf{F\,G\,\overline{B} = G\,(F_1\,\overline{B})}$ and $\mathsf{F_1}$ does not contain G, then $\mathsf{F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B} = P(F_1\,\overline{B})(F_1^\wedge\,\overline{B}\,\overline{Q_B})}$. We therefore define $\mathsf{p = dIndG\,P\,cc\,(F_1\,\overline{B})\,(F_1^\wedge\,\overline{B}\,\overline{Q_B})}$.
- If $\mathsf{F\,G\,\overline{B} = H\,\overline{B}}$ and H does not contain G, then $\mathsf{p : PredMap\,(H\,\overline{B})\,(H^\wedge\,\overline{B}\,\overline{Q_B})\,(H^\wedge\,\overline{B}\,\overline{Q_B})}$ is defined to be the identity morphism on predicates.
- If $\mathsf{F\,G\,\overline{B} = H\,(\overline{F_k\,G\,\overline{B}})}$, if H does not contain G, if H has a predicate lifting $\mathsf{H^\wedge : \forall(\overline{C : Set}) \to (\overline{C \to Set}) \to H\,\overline{C} \to Set}$, and if $\mathsf{H^\wedge}$ has a map function $\mathsf{H^\wedge Map = \forall(\overline{C : Set})(\overline{Q_C\,Q'_C : C \to Set}) \to PredMap\,\overline{C}\,\overline{Q_C}\,\overline{Q'_C} \to PredMap\,(H\,\overline{C})\,(H^\wedge\,\overline{C}\,\overline{Q_C})\,(H^\wedge\,\overline{C}\,\overline{Q'_C})}$, then since H cannot be a GADT, $\mathsf{F^\wedge\,G\,\overline{B}\,P\,\overline{Q_B} = H^\wedge\,(\overline{F_k\,G\,\overline{B}})\,(\overline{F_k^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}})}$. The induction hypothesis ensures morphisms of predicates $\overline{\mathsf{p_k : PredMap\,(F_k\,G\,\overline{B})(F_k^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B})(F_k^\wedge\,G\,\overline{B}\,P\,\overline{Q_B})}}$. We therefore define $\mathsf{p = H^\wedge Map\,(\overline{F_k\,G\,\overline{B}})\,(\overline{F_k^\wedge\,G\,\overline{B}\,G^\wedge\,\overline{Q_B}})\,(\overline{F_k^\wedge\,G\,\overline{B}\,P\,\overline{Q_B}})\,\overline{p_k}}$.

6 Truly Nested GADTs Do Not Admit Induction Rules

HERE!!

In Sections 4 and 5 we derived induction rules for GADTs that are not truly nesteed. Since both nested types and GADTs without true nesting admit induction rules, we might expect that truly nested GADTs would as well. Surprisingly, however, they do not. That is, our results from the previous section are the strongest possible. In fact, the induction rule for a data type generally relies on (unary) parametricity of its semantic interpretation, and in the case of nested types and GADTs it also relies on the data types having a functorial semantics. But whereas nested types can admit a functorial parametric semantics GADTs cannot admit both functorial and parametric semantics at the same time [13]. In this section we show how induction for truly nested GADTs nesting goes wrong by analyzing the following simple concrete example of such a type.

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

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

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

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

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

so the induction rule for G is

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

But if we now try to show that this induction rule is sound by constructing a witness dIndG inhabiting this type we run into problems. We can try to define $\mathsf{dIndG\,P\,cc\,A\,Q\,(c\,B\,e\,x)\,(Q', liftE, liftG) = cc\,A\,B\,Q\,Q'\,e\,x\,liftE\,p}$, where $\mathsf{cc : dIndC\,P}$, $\mathsf{Q : A \to Set}$, $\mathsf{e : Equal\,A\,(B \times B)}$, $\mathsf{x : G\,(G\,B)}$, $\mathsf{liftG : G^\wedge\,(G\,B)\,(G^\wedge\,B\,Q')\,x}$, and $\mathsf{Q' : B \to Set}$, $\mathsf{liftE : Equal^\wedge\,A\,(B \times B)\,Q\,(Pair^\wedge\,B\,B\,Q'\,Q')\,e}$. However, we still need to define $\mathsf{p : P\,(G\,B)\,(P\,B\,Q')\,x}$. If we try to do so by using the induction rule and letting $\mathsf{p = dIndG\,P\,cc\,(G\,B)\,(P\,B\,Q')\,x\,q}$, then we'd still need to provide $\mathsf{q : G^\wedge\,(G\,B)\,(P\,B\,Q')\,x}$. If we had the map function

$\mathsf{G^{\wedge}Map} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{Q}\,\mathsf{Q''} : \mathsf{A} \to \mathsf{Set}) \to \mathsf{PredMap}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{Q''} \to \mathsf{PredMap}\,(\mathsf{G}\,\mathsf{A})\,(\mathsf{G^{\wedge}}\,\mathsf{A}\,\mathsf{Q})\,(\mathsf{G^{\wedge}}\,\mathsf{A}\,\mathsf{Q''})$ for $\mathsf{G^{\wedge}}$, then we would be able to define $\mathsf{q} = \mathsf{GLMap}\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{G^{\wedge}}\,\mathsf{B}\,\mathsf{Q'})\,(\mathsf{P}\,\mathsf{B}\,\mathsf{Q'})\,(\mathsf{dIndG}\,\mathsf{P}\,\mathsf{cc}\,\mathsf{B}\,\mathsf{Q'})\,\mathsf{x}\,\mathsf{liftG}$. Unfortunately, however, we cannot define such a $\mathsf{G^{\wedge}Map}$. Indeed, its definition would have to be

$$\mathsf{G^{\wedge}Map}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{Q_2}\,\mathsf{m}\,(\mathsf{c}\,\mathsf{B}\,\mathsf{e}\,\mathsf{x})\,(\mathsf{Q_1}, \mathsf{liftE}, \mathsf{liftG}) = (\mathsf{Q_3}, \mathsf{liftE'}, \mathsf{liftG'})$$

for some $\mathsf{Q_3} : \mathsf{B} \to \mathsf{Set}$, $\mathsf{liftE'} : \mathsf{Equal^{\wedge}}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B})\,\mathsf{Q_2}\,(\mathsf{Pair^{\wedge}}\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q_3}\,\mathsf{Q_3})\,\mathsf{e}$, and $\mathsf{liftG'} : \mathsf{G^{\wedge}}\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{G^{\wedge}}\,\mathsf{B}\,\mathsf{Q_3})\,\mathsf{x}$, where $\mathsf{Q} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{Q_2} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{m} : \mathsf{PredMap}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{Q_2}$, $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B})$, $\mathsf{x} : \mathsf{G}\,(\mathsf{G}\,\mathsf{B})$, $\mathsf{Q_1} : \mathsf{B} \to \mathsf{Set}$, $\mathsf{liftE} : \mathsf{Equal^{\wedge}}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{B})\,\mathsf{Q}\,(\mathsf{Pair^{\wedge}}\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q_1}\,\mathsf{Q_1})\,\mathsf{e}$, and $\mathsf{liftG} : \mathsf{G^{\wedge}}\,(\mathsf{G}\,\mathsf{B})\,(\mathsf{G^{\wedge}}\,\mathsf{B}\,\mathsf{Q_1})\,\mathsf{x}$. In other words, we have a proof liftE of the (extensional) equality of the predicates Q and $\mathsf{Pair^{\wedge}}\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q_1}\,\mathsf{Q_1}$ and a morphism of predicates m from Q to $\mathsf{Q_2}$, and we need to use those data to deduce a proof of the (extensional) equality of the predicates $\mathsf{Q_2}$ and $\mathsf{Pair^{\wedge}}\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q_3}\,\mathsf{Q_3}$, for some predicate $\mathsf{Q_3}$ on B. But that is not generally possible: the facts that Q is equal to $\mathsf{Pair^{\wedge}}\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q_1}\,\mathsf{Q_1}$ and that there is a morphism of predicates m from Q to $\mathsf{Q_2}$ do not guarantee that there exists a predicate $\mathsf{Q_3}$ such that $\mathsf{Q_2}$ is equal to $\mathsf{Pair^{\wedge}}\,\mathsf{B}\,\mathsf{B}\,\mathsf{Q_3}\,\mathsf{Q_3}$.

At a deeper level, the fundamental issue is that the Equal type does not have functorial semantics [13], so that having morphisms $\mathsf{A} \to \mathsf{A'}$ and $\mathsf{B} \to \mathsf{B'}$ (for any type $\mathsf{A}, \mathsf{A'}, \mathsf{B}$ and $\mathsf{B'}$) and a proof that A is equal to $\mathsf{A'}$ does not provide a proof that B is equal to $\mathsf{B'}$. This is because GADTs can either have a syntax-only semantics or a functorial semantics. Since we are interested in induction rules, we consider the syntax-only semantics, which is parametric but not functorial. Had we considered the functorial-completion semantics, which is functorial, we would have forfeited parametricity instead. In both cases, thus, we cannot derive an induction rule for GADTs featuring nesting. Unlike nested types, indeed, GADTs do not admit a semantic interpretation that is both parametric and functorial [13].

7 Applications

In this section we use deep induction for the LTerm GADT from (6) to extract the type from a lambda term. Consider the predicate

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

that takes a lambda term and produces the type of its possible types. This predicate uses Maybe to represent potential failure, where Maybe is the type defined as follows:

$$\begin{aligned}\mathsf{data}\;\;\mathsf{Maybe} : \mathsf{Set} \to \mathsf{Set}\;\mathsf{where}&\\\mathsf{nothing} :\;\; \mathsf{Maybe}\,\mathsf{A}&\\\mathsf{just} :\;\; \mathsf{A} \to \mathsf{Maybe}\,\mathsf{A}&\end{aligned}\tag{12}$$

We want to show this predicate is satisfied for every element of $\mathsf{LTerm}\,\mathsf{A}$, i.e., we want to prove:

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

Because of the listC constructor of LTerm, this cannot be achieved without deep induction. In particular, deep induction is required to apply the induction to the individual terms in a list of terms. So, using the deep induction rule for LTerm from Section **??** (add back-reference), we define $\mathsf{getTypeProof}$ as

$$\mathsf{getTypeProof}\,\mathsf{A}\,\mathsf{t} = \mathsf{dIndLTerm}\,\mathsf{P}\,\mathsf{cvar}\,\mathsf{cabs}\,\mathsf{capp}\,\mathsf{clistc}\,\mathsf{A}\,\mathsf{K_{\top}}\,\mathsf{t}\,(\mathsf{LTermLKT}\,\mathsf{A}\,\mathsf{t})$$

where $\mathsf{t} : \mathsf{LTerm}\,\mathsf{A}$, P is the polymorphic predicate $\lambda\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{t} : \mathsf{LTerm}\,\mathsf{A}) \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{A})$ and $\mathsf{LTermLKT} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{t} : \mathsf{LTermA}) \to \mathsf{LTerm^{\wedge}}\,\mathsf{A}\,\mathsf{K_{\top}}\,\mathsf{t}$ is a function that we will define later. In addition to defining $\mathsf{LTermLKT}$, we also have to give a term for each constructor of LTerm, as already discussed in Section **??** (add back-reference):

- a term $\mathsf{cvar} : \mathsf{dIndVar}\,\mathsf{P}$ associated to the constructor var, i.e.,

$$\mathsf{cvar} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{s} : \mathsf{String})\,(\mathsf{T} : \mathsf{LType}\,\mathsf{A}) \to \mathsf{LType^{\wedge}}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{T} \;\to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{A})$$

- a term $\mathsf{cabs} : \mathsf{dIndAbs}\,\mathsf{P}$ associated to the constructor abs, i.e.,

$$\begin{aligned}&\mathsf{cabs} : \forall\,(\mathsf{A}\,\mathsf{B}\,\mathsf{C} : \mathsf{Set})\,(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{Q}' : \mathsf{B} \to \mathsf{Set})\,(\mathsf{Q}'' : \mathsf{C} \to \mathsf{Set})\\ &\quad(\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C}))\,(\mathsf{s} : \mathsf{String})\,(\mathsf{T} : \mathsf{LType}\,\mathsf{B})\,(\mathsf{t} : \mathsf{LTerm}\,\mathsf{C})\\ &\quad\to \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C})\,\mathsf{Q}\,(\mathsf{Arr}^\wedge\,\mathsf{B}\,\mathsf{C}\,\mathsf{Q}'\,\mathsf{Q}'')\,\mathsf{e} \to \mathsf{LType}^\wedge\,\mathsf{B}\,\mathsf{Q}'\,\mathsf{T} \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{C}) \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{A})\end{aligned}$$

- a term $\mathsf{capp} : \mathsf{dIndApp}\,\mathsf{P}$ associated to the constructor app, i.e.,

$$\begin{aligned}&\mathsf{capp} : \forall\,(\mathsf{A}\,\mathsf{B} : \mathsf{Set})\,(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{Q}' : \mathsf{B} \to \mathsf{Set})\,(\mathsf{t} : \mathsf{LTerm}\,(\mathsf{B} \to \mathsf{A}))\,(\mathsf{t}' : \mathsf{LTerm}\,\mathsf{B})\\ &\quad\to \mathsf{Maybe}\,(\mathsf{LType}\,(\mathsf{B} \to \mathsf{A})) \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{B}) \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{A})\end{aligned}$$

- a term $\mathsf{clistc} : \mathsf{dIndListC}\,\mathsf{P}$ associated to the constructor listC, i.e.,

$$\begin{aligned}&\mathsf{clistc} : \forall\,(\mathsf{A}\,\mathsf{B} : \mathsf{Set})\,(\mathsf{Q} : \mathsf{A} \to \mathsf{Set})\,(\mathsf{Q}' : \mathsf{B} \to \mathsf{Set})\,(\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{List}\,\mathsf{B}))\,(\mathsf{ts} : \mathsf{List}\,(\mathsf{LTerm}\,\mathsf{B}))\\ &\quad\to \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{List}\,\mathsf{B})\,\mathsf{Q}\,(\mathsf{List}^\wedge\,\mathsf{B}\,\mathsf{Q}')\,\mathsf{e} \to \mathsf{List}^\wedge\,(\mathsf{LTerm}\,\mathsf{B})\,(\mathsf{GetType}\,\mathsf{B})\,\mathsf{ts} \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{A})\end{aligned}$$

For variables we simply return the type $\mathsf{T} : \mathsf{LType}\,\mathsf{A}$ (wrapped in just), and the cases for abstraction and application are similar. The interesting case is clistc, in which we have to use the results of $\mathsf{List}^\wedge\,(\mathsf{LTerm}\,\mathsf{B})\,(\mathsf{GetType}\,\mathsf{B})\,\mathsf{ts}$ in order to extract the type of one of the terms in the list. To define clistc we pattern-match on the list of terms ts. If ts is the empty list nil, we cannot extract a type, so we return $\mathsf{nothing}$:

$$\mathsf{clistc}\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}\,\mathsf{Q}'\,\mathsf{e}\,\mathsf{nil}\,\mathsf{liftE}\,\mathsf{lift}_{\mathsf{ts}} = \mathsf{nothing}$$

where $\mathsf{liftE} : \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{List}\,\mathsf{B})\,\mathsf{Q}\,(\mathsf{List}^\wedge\,\mathsf{B}\,\mathsf{Q}')\,\mathsf{e}$ and $\mathsf{lift}_{\mathsf{ts}} : \mathsf{List}^\wedge\,(\mathsf{LTerm}\,\mathsf{B})\,(\mathsf{GetType}\,\mathsf{B})\,\mathsf{ts}$. If ts is a non-empty list $\mathsf{cons}\,\mathsf{t}\,\mathsf{ts}'$, we pattern match on $\mathsf{list}_{\mathsf{ts}}$ and use the result to construct the type we need. The type of $\mathsf{list}_{\mathsf{ts}}$ is

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

Thus, we define

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

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

7.1 Defining LTermLKT

Maybe this section can be deleted by assuming $\mathsf{LTerm}^\wedge\,\mathsf{A}\,\mathsf{K}_\top\,\mathsf{t} = \mathsf{K}_\top$. Maybe we can say this derives from parametricity. Currently we say that $\mathsf{K}_\top \times \mathsf{K}_\top = \mathsf{K}_\top$ based on the fact that products are a built-in type and so this seems to be obviously true.

The last piece of infrastructure we need in order to define $\mathsf{getTypeProof}$ is a function

$$\mathsf{LTermLKT} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{t} : \mathsf{LTerm}\,\mathsf{A}) \to \mathsf{LTerm}^\wedge\,\mathsf{A}\,\mathsf{K}_\top\,\mathsf{t}$$

that provides a proof of $\mathsf{LTerm}^\wedge\,\mathsf{A}\,\mathsf{K}_\top\,\mathsf{t}$ for any term $\mathsf{t} : \mathsf{LTerm}\,\mathsf{A}$. Because $\mathsf{LTerm}^\wedge$ is defined in terms of $\mathsf{LType}^\wedge$, $\mathsf{Arr}^\wedge$, and $\mathsf{List}^\wedge$, we need an analogous function for each of these liftings as well (respectively, LTypeLKT, ArrLKT and ListLKT). We only give the definition of LTermLKT, but the definitions for LTypeLKT, ArrLKT and ListLKT are analogous. We define LTermLKT by pattern matching on the lambda term t:

- For the var case, let $\mathsf{t} = \mathsf{var}\,\mathsf{s}\,\mathsf{T}$ for $\mathsf{s} : \mathsf{String}$ and $\mathsf{T} : \mathsf{LType}\,\mathsf{A}$, and define

$$\mathsf{LTermLKT}\,\mathsf{A}\,(\mathsf{var}\,\mathsf{s}\,\mathsf{T}) = \mathsf{LTypeLKT}\,\mathsf{A}\,\mathsf{T}$$

- For the abs case, let $\mathsf{t} = \mathsf{abs}\,\mathsf{B}\,\mathsf{C}\,\mathsf{e}\,\mathsf{s}\,\mathsf{T}\,\mathsf{t}'$ for $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C})$, $\mathsf{s} : \mathsf{String}$, $\mathsf{T} : \mathsf{LType}\,\mathsf{B}$ and $\mathsf{t}' : \mathsf{LTerm}\,\mathsf{C}$, and recall the definition of $\mathsf{LTerm}^\wedge$ for the abs constructor, instantiating the predicate $\mathsf{Q} : \mathsf{A} \to \mathsf{Set}$ to $\mathsf{K}_\top$:

$$\begin{aligned}&\mathsf{LTerm}^\wedge\,\mathsf{A}\,\mathsf{K}_\top\,(\mathsf{abs}\,\mathsf{B}\,\mathsf{C}\,\mathsf{e}\,\mathsf{s}\,\mathsf{T}\,\mathsf{t}')\\ &\quad = \textstyle\sum\,[\mathsf{Q}' : \mathsf{B} \to \mathsf{Set}]\,[\mathsf{Q}'' : \mathsf{C} \to \mathsf{Set}]\,\mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C})\,\mathsf{K}_\top\,(\mathsf{Arr}^\wedge\,\mathsf{B}\,\mathsf{C}\,\mathsf{Q}'\,\mathsf{Q}'')\,\mathsf{e} \times \mathsf{LType}^\wedge\,\mathsf{B}\,\mathsf{Q}'\,\mathsf{T} \times \mathsf{LTerm}^\wedge\,\mathsf{C}\,\mathsf{Q}''\,\mathsf{t}'\end{aligned}$$

  So, to define the abs case of $\mathsf{LTermLKT}$, we need a proof of

$$\mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C})\,\mathsf{K}_\top\,(\mathsf{Arr}^\wedge\,\mathsf{B}\,\mathsf{C}\,\mathsf{Q}'\,\mathsf{Q}'')\,\mathsf{e}$$

  i.e., that $\mathsf{K}_\top$ is (extensionally) equal to the lifting $\mathsf{Arr}^\wedge\,\mathsf{B}\,\mathsf{C}\,\mathsf{Q}'\,\mathsf{Q}''$ for some predicates $\mathsf{Q}' : \mathsf{B} \to \mathsf{Set}$ and $\mathsf{Q}'' : \mathsf{C} \to \mathsf{Set}$. The only reasonable choice for Q' and Q'' is to let both be $\mathsf{K}_\top$, which means we need a proof of

$$\mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C})\,\mathsf{K}_\top\,(\mathsf{Arr}^\wedge\,\mathsf{B}\,\mathsf{C}\,\mathsf{K}_\top\,\mathsf{K}_\top)\,\mathsf{e}$$

  Since we are working with proof-relevant predicates (i.e., functions into Set rather than functions into Bool), the lifting $\mathsf{Arr}^\wedge\,\mathsf{B}\,\mathsf{C}\,\mathsf{K}_\top\,\mathsf{K}_\top$ of $\mathsf{K}_\top$ to arrow types is not identical to $\mathsf{K}_\top$ on arrow types, but the predicates are (extensionally) isomorphic. We discuss this issue in more detail at the end of the section. For now, we assume a proof

$$\mathsf{EqualLArrKT} : \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C})\,\mathsf{K}_\top\,(\mathsf{Arr}^\wedge\,\mathsf{B}\,\mathsf{C}\,\mathsf{K}_\top\,\mathsf{K}_\top)\,\mathsf{e}$$

  and define the abs case of $\mathsf{LTermLKT}$ as

$$\mathsf{LTermLKT}\,\mathsf{A}\,(\mathsf{abs}\,\mathsf{B}\,\mathsf{C}\,\mathsf{e}\,\mathsf{s}\,\mathsf{T}\,\mathsf{t}') = (\mathsf{K}_\top, \mathsf{K}_\top, \mathsf{EqualLArrKT}, \mathsf{LTypeLKT}\,\mathsf{B}\,\mathsf{T}, \mathsf{LTermLKT}\,\mathsf{C}\,\mathsf{t}')$$

- For the app case, let $\mathsf{t} = \mathsf{app}\,\mathsf{B}\,\mathsf{t}'\,\mathsf{t}''$ for $\mathsf{t}' : \mathsf{LTerm}\,(\mathsf{B} \to \mathsf{A})$ and $\mathsf{t}'' : \mathsf{LTerm}\,\mathsf{B}$, and, just as we did for the abs case, recall the definition of $\mathsf{LTerm}^\wedge\,\mathsf{A}\,\mathsf{K}_\top\,(\mathsf{app}\,\mathsf{B}\,\mathsf{t}'\,\mathsf{t}'')$ with all of the predicates instantiated with $\mathsf{K}_\top$:

$$\mathsf{LTerm}^\wedge\,(\mathsf{B} \to \mathsf{A})\,(\mathsf{Arr}^\wedge\,\mathsf{B}\,\mathsf{A}\,\mathsf{K}_\top\,\mathsf{K}_\top)\,\mathsf{t}' \times \mathsf{LTerm}^\wedge\,\mathsf{B}\,\mathsf{K}_\top\,\mathsf{t}''$$

  The second component can be given using $\mathsf{LTermLKT}$, and we can define the first component using a proof of

$$\mathsf{LTerm}^\wedge\,(\mathsf{B} \to \mathsf{A})\,\mathsf{K}_\top\,\mathsf{t}'$$

  and a map-like function

$$\mathsf{LTermLEqualMap} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{Q}\,\mathsf{Q}' : \mathsf{A} \to \mathsf{Set}) \to \mathsf{Equal}^\wedge\,\mathsf{A}\,\mathsf{A}\,\mathsf{Q}\,\mathsf{Q}'\,\mathsf{refl} \to \mathsf{PredMap}\,(\mathsf{LTerm}\,\mathsf{A})\,(\mathsf{LTerm}^\wedge\,\mathsf{A}\,\mathsf{Q})\,(\mathsf{LTerm}^\wedge\,\mathsf{A}\,\mathsf{Q}')$$

  that takes two (extensionally) equal predicates with the same carrier and produces a morphism of predicates between their liftings. The definition is straightforward enough, so we omit the details. Using $\mathsf{LTermLEqualMap}$, we can define the app case of $\mathsf{LTermLKT}$ as

$$\mathsf{LTermLKT}\,\mathsf{A}\,(\mathsf{app}\,\mathsf{B}\,\mathsf{t}'\,\mathsf{t}'') = (\mathsf{K}_\top, \mathsf{LTermLArr}, \mathsf{LTermLKT}\,\mathsf{B}\,\mathsf{t}'')$$

  where $\mathsf{LTermLArr} : \mathsf{LTerm}^\wedge\,(\mathsf{B} \to \mathsf{A})\,(\mathsf{Arr}^\wedge\,\mathsf{B}\,\mathsf{A}\,\mathsf{K}_\top\,\mathsf{K}_\top)\,\mathsf{t}'$ is defined as

$$\mathsf{LTermLArr} = \mathsf{LTermLEqualMap}\,\mathsf{K}_\top\,(\mathsf{Arr}^\wedge\,\mathsf{B}\,\mathsf{A}\,\mathsf{K}_\top\,\mathsf{K}_\top)\;\mathsf{EqualLArrKT}\,\mathsf{t}'\,\mathsf{L}_{\mathsf{K}_\top}$$

  where $\mathsf{L}_{\mathsf{K}_\top} = \mathsf{LTermLKT}\,(\mathsf{B} \to \mathsf{A})\,\mathsf{t}' : \mathsf{LTerm}^\wedge\,(\mathsf{B} \to \mathsf{A})\,\mathsf{K}_\top\,\mathsf{t}'$.
- For the listC case, let $\mathsf{t} = \mathsf{listC}\,\mathsf{B}\,\mathsf{e}\,\mathsf{ts}$ for $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{List}\,\mathsf{B})$ and $\mathsf{ts} : \mathsf{List}\,(\mathsf{LTerm}\,\mathsf{B})$, and recall the definition of $\mathsf{LTerm}^\wedge\,\mathsf{A}\,(\mathsf{listC}\,\mathsf{B}\,\mathsf{e}\,\mathsf{ts})$ with all of the predicates instantiated to $\mathsf{K}_\top$:

$$\mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{List}\,\mathsf{B})\,\mathsf{K}_\top\,(\mathsf{List}^\wedge\,\mathsf{B}\,\mathsf{K}_\top)\,\mathsf{e} \times \mathsf{List}^\wedge\,(\mathsf{LTerm}\,\mathsf{B})\,(\mathsf{LTerm}^\wedge\,\mathsf{B}\,\mathsf{K}_\top)\,\mathsf{ts}$$

  We can give the first component by assuming a proof $\mathsf{EqualLListKT} : \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{List}\,\mathsf{B})\,\mathsf{K}_\top\,(\mathsf{List}^\wedge\,\mathsf{B}\,\mathsf{K}_\top)\,\mathsf{e}$, but for the second component we again have multiple liftings nested together. In this case, we can get a proof of

$$\mathsf{List}^\wedge\,(\mathsf{LTerm}\,\mathsf{B})\,(\mathsf{LTerm}^\wedge\,\mathsf{B}\,\mathsf{K}_\top)\,\mathsf{ts}$$

using $\mathsf{liftListMap}$, as seen in Section ?? (add back-reference), to map a morphism of predicates

$$\mathsf{PredMap}\,(\mathsf{LTerm}\,\mathsf{B})\,(\mathsf{K}_\top)\,(\mathsf{LTerm}^\wedge\,\mathsf{B}\,\mathsf{K}_\top)$$

to a morphism of lifted predicates

$$\mathsf{PredMap}\,(\mathsf{List}\,(\mathsf{LTerm}\,\mathsf{B}))\,(\mathsf{List}^\wedge\,(\mathsf{LTerm}\,\mathsf{B})\,\mathsf{K}_\top)\,(\mathsf{List}^\wedge\,(\mathsf{LTerm}\,\mathsf{B})\,(\mathsf{LTerm}^\wedge\,\mathsf{B}\,\mathsf{K}_\top))$$

We then define the listC case of $\mathsf{LTermLKT}$ as

$$\mathsf{LTermLKT}\,\mathsf{A}\,(\mathsf{listC}\,\mathsf{B}\,\mathsf{e}\,\mathsf{ts}) = (\mathsf{K}_\top,\ \mathsf{EqualLListKT},\ \mathsf{L}_{\mathsf{ListLLTermLKT}})$$

where $\mathsf{L}_{\mathsf{ListLLTermLKT}} : \mathsf{List}^\wedge\,(\mathsf{LTerm}\,\mathsf{B})\,(\mathsf{LTerm}^\wedge\,\mathsf{B}\,\mathsf{K}_\top)\,\mathsf{ts}$ is defined as

$$\mathsf{L}_{\mathsf{ListLLTermLKT}} = \mathsf{liftListMap}\,(\mathsf{LTerm}\,\mathsf{B})\,\mathsf{K}_\top\,(\mathsf{LTerm}^\wedge\,\mathsf{B}\,\mathsf{K}_\top)\,\mathsf{m}_{\mathsf{K}_\top}\,\mathsf{ts}\,(\mathsf{ListLKT}\,(\mathsf{LTerm}\,\mathsf{B})\,\mathsf{ts})$$

and $\mathsf{m}_{\mathsf{K}_\top} : \mathsf{PredMap}\,(\mathsf{LTerm}\,\mathsf{B})\,(\mathsf{K}_\top)\,(\mathsf{LTerm}^\wedge\,\mathsf{B}\,\mathsf{K}_\top)$ is defined as

$$\mathsf{m}_{\mathsf{K}_\top}\,\mathsf{t}'\,\mathsf{tt} = \mathsf{LTermLKT}\,\mathsf{B}\,\mathsf{t}'$$

where $\mathsf{t}' : \mathsf{LTerm}\,\mathsf{B}$ and tt is the single element of $\mathsf{K}_\top\,\mathsf{t}'$. The use of $\mathsf{liftListMap}$ is required in the listC case because listC takes an argument of type $\mathsf{List}\,(\mathsf{LTerm}\,\mathsf{B})$.

These are general considerations, and are not required to conclude the above argument. The above techniques can be used to define a function $\mathsf{GLKT} : \forall\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{x} : \mathsf{G}\,\mathsf{A}) \to \mathsf{G}^\wedge\,\mathsf{A}\,\mathsf{K}_\top\,\mathsf{x}$ for an arbitrary GADT G, as defined in Section ?? (add back-reference). 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 the lifting of $\mathsf{K}_\top$ by any type constructor F is extensionally equal to $\mathsf{K}_\top$ on F. For example, we might need a proof that $\mathsf{Pair}^\wedge\,\mathsf{A}\,\mathsf{B}\,\mathsf{K}_\top\,\mathsf{K}_\top$ is equal to the predicate $\mathsf{K}_\top$ on $\mathsf{A} \times \mathsf{B}$. Given a pair $(\mathsf{a}, \mathsf{b}) : \mathsf{A} \times \mathsf{B}$, we have $\mathsf{Pair}^\wedge\,\mathsf{A}\,\mathsf{B}\,\mathsf{K}_\top\,\mathsf{K}_\top(\mathsf{a}, \mathsf{b}) = \mathsf{K}_\top\,\mathsf{a} \times \mathsf{K}_\top\,\mathsf{b} = \top \times \top$, whereas $\mathsf{K}_\top\,(\mathsf{a}, \mathsf{b}) = \top$. While these types are not equal, they are clearly isomorphic. So, for simplicity of presentation, we assume $\mathsf{F}^\wedge\,\mathsf{A}\,\mathsf{K}_\top$ is equal to $\mathsf{K}_\top$ for every nested type and ADT F. Moreover, remember that, whenever G has a constructor of the form $\mathsf{c} : \mathsf{F}\,(\mathsf{G}\,\mathsf{A}) \to \mathsf{G}\,(\mathsf{K}\,\mathsf{B})$, F is only allowed to be a nested type or an ADT, and we are guaranteed to have a $\mathsf{liftFMap}$ function.

8 Conclusion

9 TODO

- find correct entcsmacro file (current one is for 2018). Maybe ask Ana Sokolova (anas@cs.uni-salzburg.at).
- reference (correctly) Haskell Symposium paper
- reference inspiration for STLC GADT : https://www.seas.upenn.edu/ cis194/spring15/lectures/11-stlc.html
- Data type vs. data structure
- Weird to code in Agda if we're talking about induction rules for Coq?
- Agda style conventions
- spacing in data type declarations
- Mention Agda flags that need to be toggled to handle true nesting

References

[1] Atkey, R. *Relational parametricity for higher kinds.* Computer Science Logic, pp. 46-61, 2012.

[2] Bainbridge, E. S., Freyd, P. Scedrov, A., and Scott, P. J. *Functorial polymorphism.* Theoretical Computer Science 70(1), pp. 35-64, 1990.

[3] Bird, R. and Meertens, L. *Nested datatypes.* Proceedings, Mathematics of Program Construction, pp. 52–67, 1998.

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

[5] Coquand, T. and Huet, G. *The calculus of constructions.* Information and Computation 76(2/3), 1988.

[6] Chlipala, A. *Library Inductive Types.* `http://adam.chlipala.net/cpdt/html/InductiveTypes.html`

[7] The Coq Development Team. *The Coq Proof Assistant*, version 8.11.0, January 2020. `https://doi.org/10.5281/zenodo.3744225`

[8] Fu, P. and Selinger, P. Dependently typed folds for nested data types, 2018. `https://arxiv.org/abs/1806.05230`

[9] Ghani, N., Johann, P., Nordvall Forsberg, F., Orsanigo, F., and Revell, T. *Bifibrational functorial semantics for parametric polymorphism.* Proceedings, Mathematical Foundations of Program Semantics, pp. 165-181, 2015.

[10] Hinze, R. *Fun with phantom types.* Proceedings, The Fun of Programming, pp. 245–262, 2003.

[11] Johann, P., Ghiorzi, E., and Jeffries, D. Accompanying Agda code for this paper. `https://cs.appstate.edu/ johannp/FoSSaCS21Code.html`

[12] Johann, P., Ghiorzi, E., and Jeffries, D. *Parametricity for primitive nested types.* Proceedings, Foundations of Software Science and Computation Structures, pp. 324-343, 2021.

[13] Johann, P., Ghiorzi, E., and Jeffries, D. *Parametricity in the presence of GADTs.* Submitted, 2021.

[14] Johann, P. and Polonsky, A. *Higher-kinded data types: Syntax and semantics* Proceedings, Logic in Computer Science 2019. PAGES?

[15] Johann, P. and Polonsky, A. *Deep induction: Induction rules for (truly) nested types.* Proceedings, Foundations of Software Science and Computation Structures, pp. 339-358, 2020.

[16] MacLane, S. *Catgories for the Working Mathematician.* Springer, 1971.

[17] McBride, C. *Dependently Typed Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999.

[18] Minsky, Y. *Why GADTs matter for performance.* `https://blog.janestreet.com/why-gadts-matter-for-performance/`, 2015.

[19] Pasalic, E., and Linger, N. *Meta-programming with typed object-language representations.* Generic Programming and Component Engineering, pp. 136-167, 2004.

[20] Penner, C. *Simpler and safer API design using GADTs.* `https://chrispenner.ca/posts/gadt-design`, 2020.

[21] Peyton Jones, S., Vytiniotis, D., Weirich, S., and Washburn, G. *Simple unification-based type inference for GADTs.* Proceedings, International Conference on Functional Programming, 2006. PAGES?

[22] Pottier, F., and Régis-Gianas, Y. *Stratified type inference for generalized algebraic data types.* Principles of Programming Languages, pp. 232-244, 2006.

[23] Schrijvers, T, Peyton Jones, S. L., Sulzmann, M., and Vytiniotis, D. *Complete and decidable type inference for GADTs.* Proceedings, International Conference on Functional Programming, pp. 341– 352, 2009.

[24] Sheard, T., and Pasalic, E. *Meta-programming with built-in type equality.* Proceedings, Workshop on Logical Frameworks and Meta-languaegs, 2004. PAGES?

[25] Tassi, E.: Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. Proceedings, Interactive Theorem Proving, pp. 1-18, 2019.

[26] Vytiniotis, D., and Weirich, S. *Parametricity, type equality, and higher-order polymorphism.* Journal of Functional Programming 20(2), pp. 175–210, 2010.

[27] Xi, H., Chen, C. and Chen, G. *Guarded recursive datatype constructors.* Proceedings, Principles of Programming Languages, pp. 224–235, 2003.

[28] Ullrich, M. *Generating Induction Principles for Nested Induction Types in MetaCoq.* PhD thesis, Saarland University, 2020.