(Deep) Induction for GADTs

Patricia Johann Enrico Ghiorzi Daniel Jeffries

{johannp, ghiorzie, jeffriesd}@appstate.edu
Department of Computer Science, Appalachian State University

Abstract

Abstract goes here.

Keywords: Please list keywords from your paper here, separated by commas.

- Weird to code in Agda if we're talking about induction rules for Coq?
- Agda style conventions

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* [13], 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:

```
data Rose : Set → Set where
  rEmpty : Rose A
  rNode  : A → List (Rose A) → Rose A
```

The induction rule Coq automatically generates for rose trees is

```
∀ (a : Set) (P : Rose a → Set) → P REmpty →
  (∀ (x : a) (ts : List (Rose a)) → P (RNode x 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

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

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

outer list structure and to the rose trees sitting inside RNode's list argument. More generally, this enhanced notion of induction should traverse *all* of the structure present in a data type, 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 principles for data types that go well beyond traditional ADTs. Such data types include (truly) nested types [3] [2], generalized algebraic data types (GADTs) [4,19,22,23], more richly indexed families [5], and deep variants of all of these.

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 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:

$$\begin{aligned}&\forall\,(\mathsf{a}:\mathtt{Set})\,(\mathsf{P}:\mathsf{Rose}\,\mathsf{a}\to\mathtt{Set})\,(\mathsf{Q}:\mathsf{a}\to\mathtt{Set})\to\mathsf{P}\,\mathsf{REmpty}\to\\&\quad(\forall\,(\mathsf{x}:\mathsf{a})\,(\mathsf{ts}:\mathsf{List}\,(\mathsf{Rose}\,\mathsf{a}))\to\mathsf{Q}\,\mathsf{x}\to\mathsf{List}^{\wedge}\,\mathsf{P}\,\mathsf{ts}\to\mathsf{P}\,(\mathsf{RNode}\,\mathsf{x}\,\mathsf{ts}))\to\\&\quad\forall\,(\mathsf{x}:\mathsf{Rose}\,\mathsf{a})\to\mathsf{Rose}^{\wedge}\,\mathsf{Q}\,\mathsf{x}\to\mathsf{P}\,\mathsf{x}\end{aligned}$$

Here, List^ (resp., Rose^) 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 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 SPair for the GADT Perhaps Lam data type illustrates better? Perhaps also shows non-inbabitation?

$$\begin{aligned}&\mathsf{data}\;\mathsf{Seq}\,(\mathsf{a}:\mathtt{Set}):\mathtt{Set}\;\mathsf{where}\\&\quad\mathsf{SConst}:\mathsf{a}\to\mathsf{Seq}\,\mathsf{a}\\&\quad\mathsf{SPair}\;:\mathsf{Seq}\,\mathsf{a}\to\mathsf{Seq}\,\mathsf{b}\to\mathsf{Seq}\,(\mathsf{a}\times\mathsf{b})\end{aligned}\tag{1}$$

of sequences only constructs sequences of pairs, rather than sequences of arbitrary type, as does SConst. 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 [14], and a parametric semantics interpreting them as their Church encodings []. As explained in [11], 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. 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 semantics and that parametric semantics for proper GADTs are sufficiently distinct that, by contrast with the situation for ADTs and other nested types [2,8,10], it is not actually possible to define a functorial parametric semantics for them [11].

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

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

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 GADTs directly from their syntax. This framework can serve as a basis for extending modern proof assistants' automatic generation of structural induction rules for ADTs to automatic generation of deep induction rules for GADTs. As for ADTs and other nested types, the structural induction 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 [13] to syntax-only GADTs. Indeed, the derivation of induction rules given there makes crucial use of the functoriality of data types' interpretations from [12], and that is precisely what the interpretation of GADTs as their Church encodings fails to deliver. Instead, we first give the (deep) induction rule and a predicate lifting styled after those of [13] for the simplest and most important GADT, namely the $\mathsf{Equality}$ GADT. We then treat any other GADT by writing it as its so-called *Henry Ford encoding* [4,9,15,21,22] and using the predicate liftings for $\mathsf{Equality}$ and any other constituent data types to appropriately thread the custom predicates through the structures of the GADT. Significantly, since we are treating GADTs as syntax-only, rather than functorial, we cannot handle truly nested GADTs like we handled truly nested types, but this is not much of a restriction, since I've never seen GADTs defined over themselves in applications. This process yields, in particular, the following deep induction rule for Seq: Order of arguments?

$$\forall(\mathsf{P} : \forall(\mathsf{a} : \mathsf{Set}) \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{Seq}\,\mathsf{a} \to \mathsf{Set}) \to \mathsf{CConst}\,\mathsf{P} \to \mathsf{CSPair}\,\mathsf{P} \\ \to \forall(\mathsf{a} : \mathsf{Set})(\mathsf{Q}_\mathsf{a} : \mathsf{a} \to \mathsf{Set})(\mathsf{s}_\mathsf{a} : \mathsf{Seq}\,\mathsf{a}) \to \mathsf{Seq}^\wedge\,\mathsf{a}\,\mathsf{Q}_\mathsf{a}\,\mathsf{s}_\mathsf{a} \to \mathsf{P}\,\mathsf{a}\,\mathsf{Q}_\mathsf{a}\,\mathsf{s}_\mathsf{a}$$

where CConst is the function Why λ?

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

associated with the Const constructor and CSPair is the function

$$\lambda(\mathsf{P} : \forall(\mathsf{a} : \mathsf{Set}) \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{Seq}\,\mathsf{a} \to \mathsf{Set}) \to \forall(\mathsf{a}\,\mathsf{b}\,\mathsf{c} : \mathsf{Set})(\mathsf{Q}_\mathsf{a} : \mathsf{a} \to \mathsf{Set})(\mathsf{Q}_\mathsf{b} : \mathsf{b} \to \mathsf{Set})(\mathsf{Q}_\mathsf{c} : \mathsf{c} \to \mathsf{Set}) \\ (\mathsf{s}_\mathsf{b} : \mathsf{Seq}\,\mathsf{b})(\mathsf{s}_\mathsf{c} : \mathsf{Seq}\,\mathsf{c})(\mathsf{e} : \mathsf{Equal}\,\mathsf{a}\,(\mathsf{b} \times \mathsf{c})) \to \mathsf{Equal}^\wedge\mathsf{a}\,(\mathsf{b} \times \mathsf{c})\,\mathsf{Q}_\mathsf{a}\,(\mathsf{Pair}^\wedge\,\mathsf{b}\,\mathsf{c}\,\mathsf{Q}_\mathsf{b}\,\mathsf{Q}_\mathsf{c})\,\mathsf{e} \\ \to \mathsf{P}\,\mathsf{b}\,\mathsf{Q}_\mathsf{b}\,\mathsf{s}_\mathsf{b} \to \mathsf{P}\,\mathsf{c}\,\mathsf{Q}_\mathsf{c}\,\mathsf{s}_\mathsf{c} \to \mathsf{PaQ}_\mathsf{a}(\mathsf{SPair}\,\mathsf{b}\,\mathsf{c}\,\mathsf{e}\,\mathsf{s}_\mathsf{b}\,\mathsf{s}_\mathsf{c})$$

associated with the SPair constructor,

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 [**?**].

1.1 Related Work

Ullrich's plugin?

Tassi's Elpi

Peter Selinger and Fu Peng

2 Deep induction for ADTs and nested types

2.1 Induction principles for ADTs and nested types

A structural induction principle 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 principles 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 $\mathsf{x} : \mathsf{A}$ to the set of proofs that $\mathsf{P}\,\mathsf{x}$ holds. For example, the

induction principle for the standard data type

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

of lists is

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

As mentioned above, 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 List^ as described in the introduction, it is

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

Structural induction can also 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 PTree (A × A) to construct data of type PTree A. Thus, it is clear that the instances of PTree at various indices cannot be defined independently, and that the entire inductive family of types must therefore be defined at once. This intertwinedness of the instances of nested types is reflected in their structural induction rules, which 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

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

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 A × 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 PTree A asserting that Q holds for every element of type A contained in its perfect tree argument.

It is not possible to extend structural induction to *truly* nested types, i.e., to nested types whose recursive occurrences appear below their own type constructors. Indeed, defining any induction rule for such a type requires that we can lift the predicate argument to that rule over the type itself. The observation that any induction rule for a truly nested type would thus necessarily have to be a deep induction rule was, in fact, the original motivation for the introduction of deep induction in [13].

The quintessential example of a truly nested type is that of bushes [3]:

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

Its (deep) induction rule is

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

where $\mathsf{Bush}^{\wedge} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{Bush}\,\mathsf{A} \to \mathsf{Set}$ is the folllowing 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:

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

It is worth noting that for those nested types admitting both deep induction and structural induction, the former generalizes the latter. Indeed, the structural induction rule for a nested type is recoverable from its deep induction rule by taking the custom predicates on their data of primitive types to be constantly True-valued predicates. Such an instantiation ensures that a deep induction rule only inspects the top-level structure of its argument data, rather than the contents of that structur. A concrete example of such a derivation is given in Section 4.1.

3 GADTs and Their Encodings

While a constructor of a nested type can take *as arguments* data whose types involve occurrences of the type instantiated at indices other than the one being defined, its *return type* still has to be at the same instance as the one being defined. For example, all constructors of $\mathsf{PTree}\,\mathsf{A}$ have to return an element of type $\mathsf{PTree}\,\mathsf{A}$. By contrast, *generalized algebraic data types* (GADTs) [4,22,23] relax this restriction by allowing their constructors both to take as arguments *and return as results* data whose types involve type instances of the GADT other than the one being defined.

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

$$\begin{aligned}&\mathsf{data}\ \mathsf{Equal} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{eRefl} : \mathsf{Equal}\,\mathsf{A}\,\mathsf{A}\end{aligned} \tag{2}$$

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. In this case $\mathsf{Equal}\,\mathsf{a}\,\mathsf{b}$ contains a single data element; 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 of sequences from the introduction 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 understood in terms of the equality GADT by rewriting it as its Henry Ford encoding [4,9,15,21,22] as follows:

$$\begin{aligned}&\mathsf{data}\ \mathsf{Seq} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{sConst} : \mathsf{A} \to \mathsf{Seq}\,\mathsf{A}\\ &\quad \mathsf{sPair}\ \ : \forall(\mathsf{B}\,\mathsf{C} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{C}) \to \mathsf{Seq}\,\mathsf{B} \to \mathsf{Seq}\,\mathsf{C} \to \mathsf{Seq}\,\mathsf{A}\end{aligned} \tag{3}$$

Here, the requirement that the sPair constructor produces an instance of Seq at a product type has been replaced with the requirement that the instance of Seq returned by sPair is *equal* to some product type. This

encoding is particularly convenient when viewing GADTs as their Church encodings [1,24].

An example of a deep GADT is given by an encoding of simply typed lambda terms. More robust variations on the simply typed lambda calculus are, of course, possible. But since the particular calculus we have chosen is rich enough to illustrate all of the essential aspects of deep induction for GADTs, we keep the types and terms to a minimum here for the purposes of exposition.

Types are either booleans, arrow types, or list types. We represent types via the following GADT: Wrong naming conventions throughout this example!

$$\begin{aligned}&\mathsf{data}\ \mathsf{LType} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{tBool} : \forall(\mathsf{B} : \mathsf{Set}) \to \mathsf{LType}\,\mathsf{Bool}\\ &\quad \mathsf{tArr}\ \ : \forall(\mathsf{B}\,\mathsf{C} : \mathsf{Set}) \to \mathsf{LType}\,\mathsf{B} \to \mathsf{LType}\,\mathsf{C} \to \mathsf{LType}\,(\mathsf{B} \to \mathsf{C})\\ &\quad \mathsf{tList}\ : \forall(\mathsf{B} : \mathsf{Set}) \to \mathsf{LType}\,\mathsf{B} \to \mathsf{LType}\,(\mathsf{List}\,\mathsf{B})\end{aligned} \tag{4}$$

Terms are either variables, abstractions, applications, or lists of such terms. We represent terms via

$$\begin{aligned}&\mathsf{data}\ \mathsf{LTerm} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{var}\ \ : \mathsf{String} \to \mathsf{LType}\,\mathsf{A} \to \mathsf{LTerm}\,\mathsf{A}\\ &\quad \mathsf{abs}\ \ : \forall(\mathsf{B}\,\mathsf{C} : \mathsf{Set}) \to \mathsf{String} \to \mathsf{LType}\,\mathsf{B} \to \mathsf{LTerm}\,\mathsf{C} \to \mathsf{LTerm}\,(\mathsf{B} \to \mathsf{C})\\ &\quad \mathsf{app}\ \ : \forall(\mathsf{B} : \mathsf{Set}) \to \mathsf{LTerm}(\mathsf{B} \to \mathsf{A}) \to \mathsf{LTerm}\,\mathsf{B} \to \mathsf{LTerm}\,\mathsf{A}\\ &\quad \mathsf{listC}\ : \forall(\mathsf{B} : \mathsf{Set}) \to \mathsf{List}\,(\mathsf{LTerm}\,\mathsf{B}) \to \mathsf{LTerm}\,(\mathsf{List}\,\mathsf{B})\end{aligned} \tag{5}$$

The type parameter for LTerm tracks the types of simply typed lambda calculus terms. For example, the type LTerm A contains lambda terms of type A. Variables are tagged with their types by the constructors var and abs. There, LType ensures that variables can only be tagged with legal types, e.g., Bool, Bool → Bool, List (Bool → Bool), etc. The presence of LType in these constructors thus guarantees that all lambda terms produced by var, abs, app, and listC are well-typed.

The Henry Ford encodings representing these GADTs in terms of the equality GADT are:

$$\begin{aligned}&\mathsf{data}\ \mathsf{LType} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{tBool} : \forall(\mathsf{B} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,\mathsf{Bool} \to \mathsf{LType}\,\mathsf{A}\\ &\quad \mathsf{tArr}\ \ : \forall(\mathsf{B}\,\mathsf{C} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C}) \to \mathsf{LType}\,\mathsf{B} \to \mathsf{LType}\,\mathsf{C} \to \mathsf{LType}\,\mathsf{A}\\ &\quad \mathsf{tList}\ : \forall(\mathsf{B} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,(\mathsf{List}\ \mathsf{B}) \to \mathsf{LType}\,\mathsf{B} \to \mathsf{LType}\,\mathsf{A}\end{aligned} \tag{6}$$

and

$$\begin{aligned}&\mathsf{data}\ \mathsf{LTerm} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{var}\ \ : \mathsf{String} \to \mathsf{LType}\,\mathsf{A} \to \mathsf{LTerm}\,\mathsf{A}\\ &\quad \mathsf{abs}\ \ : \forall(\mathsf{B}\,\mathsf{C} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C}) \to \mathsf{String} \to \mathsf{LType}\,\mathsf{B} \to \mathsf{LTerm}\,\mathsf{C} \to \mathsf{LTerm}\,\mathsf{A}\\ &\quad \mathsf{app}\ \ : \forall(\mathsf{B} : \mathsf{Set}) \to \mathsf{LTerm}(\mathsf{B} \to \mathsf{A}) \to \mathsf{LTerm}\,\mathsf{B} \to \mathsf{LTerm}\,\mathsf{A}\\ &\quad \mathsf{listC}\ : \forall(\mathsf{B} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,(\mathsf{List}\,\mathsf{B}) \to \mathsf{List}\,(\mathsf{LTerm}\,\mathsf{B}) \to \mathsf{LTerm}\,\mathsf{A}\end{aligned} \tag{7}$$

We will revisit these GADTs in Sections ?? and ??.

4 (Deep) induction for GADTs

Reasoning about GADTs can generally be quite difficult, even more so than for ADTs and nested types. To help with that, we want to extend the deep induction principle seen in Section 2.1 to GADTs.

Induction rules, and specifically deep induction rules for nested types, are traditionally derived using the functorial semantics of data types in the setting of a parametric model [12]. In particular, relational parametricity is used to validate the induction principle because induction is, itself, a form of unary parametricity, where binary relations have been replaced with predicates, which are essentially unary relations. Is there a reference for this?

Unfortunately, this approach cannot possibly be employed to prove a deep induction rule for GADTs, as these types do not allow for a functorial interpretation, at least in a parametric model [11].

Nevertheless, this paper shows how to extend deep induction to some GADTs. We will first demonstrate how to derive the deep induction rule for some example cases, and then provide a general principle that works for generic GADTs, as long as these do not feature nesting in their definition.

4.1 (Deep) induction for Equal

As a first example, we derive the induction rule for the Equal type from example **??**. This will provide a simple case study that will inform the investigation of more complex GADTs. Moreover, since we define GADTs using the Equal type, as for example in Equation 1, this example will be instrumental in stating and deriving the induction rule of other GADTs.

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

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

defined as

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

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

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

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

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

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

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

$$\mathsf{dIndEqual}\,\mathsf{P}\,\mathsf{crefl}\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}_{\mathsf{A}}\,\mathsf{Q}_{\mathsf{B}}\,\mathsf{e}\,\mathsf{liftE} : \mathsf{P}\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}_{\mathsf{A}}\,\mathsf{Q}_{\mathsf{B}}\,\mathsf{e}$$

for $\mathsf{crefl} : \mathsf{dIndRefl}\,\mathsf{P}$, $\mathsf{Q}_{\mathsf{A}} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{Q}_{\mathsf{B}} : \mathsf{B} \to \mathsf{Set}$, $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,\mathsf{B}$ and $\mathsf{liftE} : \mathsf{Equal}^{\wedge}\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}_{\mathsf{A}}\,\mathsf{Q}_{\mathsf{B}}\,\mathsf{e}$. By pattern-matching, we only need to consider the case where $\mathsf{A} = \mathsf{B}$ and $\mathsf{e} = \mathsf{refl}$, and define

$$\mathsf{dIndEqual}\,\mathsf{P}\,\mathsf{crefl}\,\mathsf{A}\,\mathsf{A}\,\mathsf{Q}_{\mathsf{A}}\,\mathsf{Q}'_{\mathsf{A}}\,\mathsf{refl}\,\mathsf{liftE} = \mathsf{crefl}\,\mathsf{A}\,\mathsf{Q}_{\mathsf{A}}\,\mathsf{Q}'_{\mathsf{A}}\,\mathsf{liftE}$$

Having provided a well-defined term for it, we have shown that the induction rule for Equal is sound.

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

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

As is the case for ADTs and nested types, the structural induction rule for Equal is a consequence of the deep induction rule. Indeed, we can define $\mathsf{indEqual}$ as

$$\mathsf{indEqual}\,\mathsf{Q}\,\mathsf{srefl}\,\mathsf{A}\,\mathsf{B}\,\mathsf{e} = \mathsf{dIndEqual}\,\mathsf{P}\,\mathsf{srefl}\,\mathsf{A}\,\mathsf{B}\,\mathsf{K}_\top^\mathsf{A}\,\mathsf{K}_\top^\mathsf{B}\,\mathsf{e}\,\mathsf{liftE}$$

where $\mathsf{Q} : \forall(\mathsf{A}\,\mathsf{B} : \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,\mathsf{B} \to \mathsf{Set}$, $\mathsf{srefl} : \forall(\mathsf{C} : \mathsf{Set}) \to \mathsf{P}\,\mathsf{C}\,\mathsf{C}\,\mathsf{refl}$ and $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,\mathsf{B}$, and

- $\mathsf{P} : \forall(\mathsf{A}\,\mathsf{B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to (\mathsf{B} \to \mathsf{Set}) \to \mathsf{Equal}\,\mathsf{A}\,\mathsf{B} \to \mathsf{Set}$ is defined as $\mathsf{P}\,\mathsf{A}\,\mathsf{B}\,\mathsf{Q}_\mathsf{A}\,\mathsf{Q}_\mathsf{B}\,\mathsf{e} = \mathsf{Q}\,\mathsf{A}\,\mathsf{B}\,\mathsf{e}$;
- $\mathsf{K}_\top^\mathsf{A}$ and $\mathsf{K}_\top^\mathsf{B}$ are the constantly $\top$-valued predicates on, respectively, A and B;
- $\mathsf{liftE} : \mathsf{Equal}^\wedge\,\mathsf{A}\,\mathsf{B}\,\mathsf{K}_\top^\mathsf{A}\,\mathsf{K}_\top^\mathsf{B}\,\mathsf{e}$ is defined by pattern matching, i.e., in case $\mathsf{A} = \mathsf{B}$ and $\mathsf{e} = \mathsf{refl}$, it is defined as $\mathsf{liftE}\,\mathsf{a} = \mathsf{refl} : \mathsf{Equal}\,\mathsf{A}\,\mathsf{A}$ for every $\mathsf{a} : \mathsf{A}$.

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

4.2 (Deep) induction for Seq

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

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

which is given by pattern-matching as

$$\mathsf{Seq}^\wedge\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,(\mathsf{const}\,\mathsf{a}) = \mathsf{Q}_\mathsf{A}\,\mathsf{a}$$

where $\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set}$ and $\mathsf{a} : \mathsf{A}$, and

$$\begin{aligned}&\mathsf{Seq}^\wedge\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,(\mathsf{sPair}\,\mathsf{B}\,\mathsf{C}\,\mathsf{e}\,\mathsf{s}_\mathsf{B}\,\mathsf{s}_\mathsf{C})\\ &\quad = \textstyle\sum[\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set}][\mathsf{Q}_\mathsf{C} : \mathsf{C} \to \mathsf{Set}]\ \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{B} \times \mathsf{C})\,\mathsf{Q}_\mathsf{A}\,(\mathsf{Q}_\mathsf{B} \times \mathsf{Q}_\mathsf{C})\,\mathsf{e} \times \mathsf{Seq}^\wedge\,\mathsf{B}\,\mathsf{Q}_\mathsf{B}\,\mathsf{s}_\mathsf{B} \times \mathsf{Seq}^\wedge\,\mathsf{C}\,\mathsf{Q}_\mathsf{C}\,\mathsf{s}_\mathsf{C}\end{aligned}$$

where $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{C})$, $\mathsf{s}_\mathsf{B} : \mathsf{Seq}\,\mathsf{B}$ and $\mathsf{s}_\mathsf{C} : \mathsf{Seq}\,\mathsf{C}$.

Finally, let $\mathsf{dIndConst}$ be the function

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

associated to the const constructor, and let $\mathsf{dIndSPair}$ be the function

$$\begin{aligned}&\lambda(\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{Seq}\,\mathsf{A} \to \mathsf{Set})\\ &\quad \to \forall(\mathsf{A}\,\mathsf{B}\,\mathsf{C} : \mathsf{Set})(\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set})(\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set})(\mathsf{Q}_\mathsf{C} : \mathsf{C} \to \mathsf{Set})(\mathsf{s}_\mathsf{B} : \mathsf{Seq}\,\mathsf{B})(\mathsf{s}_\mathsf{C} : \mathsf{Seq}\,\mathsf{C})(\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \times \mathsf{C}))\\ &\quad \to \mathsf{Equal}^\wedge\mathsf{A}\,(\mathsf{B} \times \mathsf{C})\,\mathsf{Q}_\mathsf{A}\,(\mathsf{Pair}^\wedge\,\mathsf{B}\,\mathsf{C}\,\mathsf{Q}_\mathsf{B}\,\mathsf{Q}_\mathsf{C})\,\mathsf{e} \to \mathsf{P}\,\mathsf{B}\,\mathsf{Q}_\mathsf{B}\,\mathsf{s}_\mathsf{B} \to \mathsf{P}\,\mathsf{C}\,\mathsf{Q}_\mathsf{C}\,\mathsf{s}_\mathsf{C} \to \mathsf{PAQ}_\mathsf{A}(\mathsf{SPair}\,\mathsf{B}\,\mathsf{C}\,\mathsf{e}\,\mathsf{s}_\mathsf{B}\,\mathsf{s}_\mathsf{C})\end{aligned}$$

associated to the sPair constructor.

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

$$\begin{aligned}&\forall(\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{Seq}\,\mathsf{A} \to \mathsf{Set}) \to \mathsf{CConst}\,\mathsf{P} \to \mathsf{CSPair}\,\mathsf{P}\\ &\quad \to \forall(\mathsf{A} : \mathsf{Set})(\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set})(\mathsf{s}_\mathsf{A} : \mathsf{Seq}\,\mathsf{A}) \to \mathsf{Seq}^\wedge\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,\mathsf{s}_\mathsf{A} \to \mathsf{P}\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,\mathsf{s}_\mathsf{A}\end{aligned}$$

To validate the induction rule, we define a term $\mathsf{dIndSeq}$ for the above type. We have to define

$$\mathsf{dIndSeq}\,\mathsf{P}\,\mathsf{cconst}\,\mathsf{cspair}\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,\mathsf{s}_\mathsf{A}\,\mathsf{liftA} : \mathsf{P}\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,\mathsf{s}_\mathsf{A}$$

where $\mathsf{P} : \forall(\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{Seq}\,\mathsf{A} \to \mathsf{Set}$, $\mathsf{cconst} : \mathsf{cConst}\,\mathsf{P}$, $\mathsf{cspair} : \mathsf{cSPair}\,\mathsf{P}$, $\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{s}_\mathsf{A} : \mathsf{Seq}\,\mathsf{A}$ and $\mathsf{liftA} : \mathsf{Seq}^\wedge\,\mathsf{A}\,\mathsf{Q}\,\mathsf{s}_\mathsf{A}$, and we proceed by pattern-matching on s_A. Let $\mathsf{s}_\mathsf{A} = \mathsf{const}\,\mathsf{a}$ for $\mathsf{a} : \mathsf{A}$, and define

$$\mathsf{dIndSeq}\,\mathsf{P}\,\mathsf{cconst}\,\mathsf{cspair}\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,(\mathsf{const}\,\mathsf{a})\,\mathsf{liftA} = \mathsf{cconst}\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,\mathsf{a}\,\mathsf{liftA}$$

Notice that $\mathsf{Seq}^\wedge\,\mathsf{A}\,\mathsf{Q_A}\,(\mathsf{const\,a}) = \mathsf{Q_A\,a}$, and thus $\mathsf{liftA} : \mathsf{Q_A\,a}$, making the right-hand-side in the above expression type-check. Now, let $\mathsf{s_A} = \mathsf{sPair\,B\,C\,e\,s_B\,s_C}$ for $\mathsf{e} : \mathsf{Equal\,A\,(B \times C)}$, $\mathsf{s_B} : \mathsf{Seq\,B}$ and $\mathsf{s_C} : \mathsf{Seq\,C}$, and define

$$\mathsf{dIndSeq\,P\,cconst\,cspair\,A\,Q_A\,(sPair\,B\,C\,e\,s_B\,s_C)\,(Q_B, Q_C, liftE, liftB, liftC) = cspair\,A\,B\,C\,Q_A\,Q_B\,Q_C\,s_B\,s_C\,e\,liftE\,p_B\,p_C}$$

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

- $\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}$ and $\mathsf{Q_C} : \mathsf{C} \to \mathsf{Set}$;
- $\mathsf{liftE} : \mathsf{Equal}^\wedge\,\mathsf{A\,(B \times C)\,Q_A\,(Q_B \times Q_C)\,e}$;
- $\mathsf{liftB} : \mathsf{Seq}^\wedge\,\mathsf{B\,Q_B\,s_B}$ and $\mathsf{liftC} : \mathsf{Seq}^\wedge\,\mathsf{C\,Q_C\,s_C}$;

and $\mathsf{p_B}$ and $\mathsf{p_C}$ are defined as follows:

$$\mathsf{p_B} = \mathsf{DISeq\,P\,cconst\,cspair\,B\,Q_B\,s_B\,liftB} : \mathsf{P\,B\,Q_B\,s_B}$$
$$\mathsf{p_C} = \mathsf{DISeq\,P\,cconst\,cspair\,C\,Q_C\,s_C\,liftC} : \mathsf{P\,C\,Q_C\,s_C}$$

4.3 (Deep) induction for LTerm

We now present an induction rule for the LTerm type defined in Example **??**. For that, we need liftings for both LType and LTerm. We will also need a lifting of predicates for arrow types, since arrow types appear in LType and LTerm. The lifting for arrow types

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

is defined as

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

Now we can define the lifting for LType, which has type

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

and is defined by pattern-matching as

$$\mathsf{LType}^\wedge\,\mathsf{A\,Q_A\,(tBool\,B\,e)} = \textstyle\sum[\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}]\,\mathsf{Equal}^\wedge\,\mathsf{A\,B\,Q_A}\,\mathsf{K}_\top^{\mathsf{Bool}}\,\mathsf{e}$$

where $\mathsf{Q_A} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{K}_\top^{\mathsf{Bool}}$ is the constantly $\top$-valued predicate on Bool and $\mathsf{e} : \mathsf{Equal\,A\,Bool}$;

$$\mathsf{LType}^\wedge\,\mathsf{A\,Q_A\,(tArr\,B\,C\,e\,T_B\,T_C)}$$
$$= \textstyle\sum[\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}][\mathsf{Q_C} : \mathsf{C} \to \mathsf{Set}]\,\mathsf{Equal}^\wedge\,\mathsf{A\,(B \to C)\,Q_A\,(Arr}^\wedge\,\mathsf{B\,C\,Q_B\,Q_C)\,e} \times \mathsf{LType}^\wedge\,\mathsf{B\,Q_B\,T_B} \times \mathsf{LType}^\wedge\,\mathsf{C\,Q_C\,T_C}$$

where $\mathsf{Q_A} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{e} : \mathsf{Equal\,A\,(B \to C)}$, $\mathsf{T_B} : \mathsf{LType\,B}$ and $\mathsf{T_C} : \mathsf{LType\,C}$; and

$$\mathsf{LType}^\wedge\,\mathsf{A\,Q_A\,(tList\,B\,e\,T_B)} = \textstyle\sum[\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}]\,\mathsf{Equal}^\wedge\,\mathsf{A\,(List\,B)\,Q_A\,(List}^\wedge\,\mathsf{B\,Q_B)\,e} \times \mathsf{LType}^\wedge\,\mathsf{B\,Q_B\,T_B}$$

where $\mathsf{Q_A} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{e} : \mathsf{Equal\,A\,(List\,B)}$, and $\mathsf{T_B} : \mathsf{LType\,B}$. We can define the lifting for LTerm. Again, the lifting

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

is defined by pattern-matching as

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

where $\mathsf{s} : \mathsf{String}$ and $\mathsf{T_A} : \mathsf{LType\,A}$;

$$\mathsf{LTerm}^\wedge\,\mathsf{A\,Q_A\,(abs\,B\,C\,e\,s\,T_B\,t_C)}$$
$$= \textstyle\sum[\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}][\mathsf{Q_C} : \mathsf{C} \to \mathsf{Set}]\,\mathsf{Equal}^\wedge\,\mathsf{A\,(B \to C)\,Q_A\,(Arr}^\wedge\,\mathsf{B\,C\,Q_B\,Q_C)\,e} \times \mathsf{LType}^\wedge\,\mathsf{B\,Q_B\,T_B} \times \mathsf{LTerm}^\wedge\,\mathsf{C\,Q_C\,t_C}$$

where $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{B} \to \mathsf{C})$, $\mathsf{s} : \mathsf{String}$, $\mathsf{T_B} : \mathsf{LType}\,\mathsf{B}$, and $\mathsf{t_C} : \mathsf{LTerm}\,\mathsf{C}$;

$$\mathsf{LTerm^\wedge}\,\mathsf{A}\,\mathsf{Q_A}\,(\mathsf{app}\,\mathsf{B}\,\mathsf{t_{BA}}\,\mathsf{t_B}) = \sum[\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}]\,\mathsf{LTerm^\wedge}\,(\mathsf{B} \to \mathsf{A})\,(\mathsf{Arr^\wedge}\,\mathsf{B}\,\mathsf{A}\,\mathsf{Q_B}\,\mathsf{Q_A})\,\mathsf{t_{BA}} \times \mathsf{LTerm^\wedge}\,\mathsf{B}\,\mathsf{Q_B}\,\mathsf{t_B}$$

where $\mathsf{t_{BA}} : \mathsf{LTerm}\,(\mathsf{B} \to \mathsf{A})$ and $\mathsf{t_B} : \mathsf{LTerm}\,\mathsf{B}$; and

$$\mathsf{LTerm^\wedge}\,\mathsf{A}\,\mathsf{Q_A}\,(\mathsf{listC}\,\mathsf{B}\,\mathsf{e}\,\mathsf{ts}) = \sum[\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}]\,\mathsf{Equal^\wedge}\,\mathsf{A}\,(\mathsf{List}\,\mathsf{B})\,\mathsf{Q_A}\,(\mathsf{List^\wedge}\,\mathsf{B}\,\mathsf{Q_B})\,\mathsf{e} \times \mathsf{List^\wedge}\,(\mathsf{LTerm}\,\mathsf{B})\,(\mathsf{LTerm^\wedge}\,\mathsf{B}\,\mathsf{Q_B})\,\mathsf{ts}$$

where $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{List}\,\mathsf{B})$ and $\mathsf{ts} : \mathsf{List}\,(\mathsf{LTermB})$. Notice that we use the lifting for List in the case for the listC constructor.

With these liftings, we can define the deep induction principle for LTerm. Let $\mathsf{dIndVar}$ be the function

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

associated to the var constructor. Let $\mathsf{dIndAbs}$ be the function

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

associated to the abs constructor. Let $\mathsf{dIndApp}$ be the function

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

associated to the app constructor. Let $\mathsf{dIndListC}$ be the function

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

associated to the listC constructor.

The type of the deep induction principle for LTerm is

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

To prove the induction principle, we define a term $\mathsf{dIndLTerm}$ for it. We have to define

$$\mathsf{dIndLTerm}\,\mathsf{P}\,\mathsf{cvar}\,\mathsf{cabs}\,\mathsf{capp}\,\mathsf{clistc}\,\mathsf{A}\,\mathsf{Q_A}\,\mathsf{t_A}\,\mathsf{liftA} : \mathsf{P}\,\mathsf{A}\,\mathsf{Q_A}\,\mathsf{t_A}$$

where $\mathsf{cvar} : \mathsf{dIndVar}$, $\mathsf{cabs} : \mathsf{dIndAbs}$, $\mathsf{capp} : \mathsf{dIndApp}$, $\mathsf{clistc} : \mathsf{dIndListC}$, $\mathsf{Q_A} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{t_A} : \mathsf{LTerm}\,\mathsf{A}$, and $\mathsf{liftA} : \mathsf{LTerm^\wedge}\,\mathsf{A}\,\mathsf{Q_A}\,\mathsf{t_A}$. As before, we prove the induction principle by pattern matching on $\mathsf{t_A}$. For the var constructor case, let $\mathsf{t_A} = \mathsf{var}\,\mathsf{s}\,\mathsf{T_A}$ and define

$$\mathsf{dIndLTerm}\,\mathsf{P}\,\mathsf{cvar}\,\mathsf{cabs}\,\mathsf{capp}\,\mathsf{clistc}\,\mathsf{A}\,\mathsf{Q_A}\,(\mathsf{var}\,\mathsf{s}\,\mathsf{T_A})\,\mathsf{liftA} = \mathsf{cvar}\,\mathsf{A}\,\mathsf{Q_A}\,\mathsf{s}\,\mathsf{T_A}\,\mathsf{liftA}$$

for $\mathsf{s} : \mathsf{String}$ and $\mathsf{T_A} : \mathsf{LType}\,\mathsf{A}$. Notice that $\mathsf{LTerm^\wedge}\,\mathsf{A}\,\mathsf{Q_A}\,(\mathsf{var}\,\mathsf{s}\,\mathsf{T_A}) = \mathsf{LType^\wedge}\,\mathsf{A}\,\mathsf{Q_A}\,\mathsf{T_A}$, so $\mathsf{liftA} : \mathsf{LType^\wedge}\,\mathsf{A}\,\mathsf{Q_A}\,\mathsf{T_A}$. For the abs case, let $\mathsf{t_A} = \mathsf{abs}\,\mathsf{B}\,\mathsf{C}\,\mathsf{e}\,\mathsf{s}\,\mathsf{T_B}\,\mathsf{t_C}$ and define

$$\begin{aligned}&\mathsf{dIndLTerm}\,\mathsf{P}\,\mathsf{cvar}\,\mathsf{cabs}\,\mathsf{capp}\,\mathsf{clistc}\,\mathsf{A}\,\mathsf{Q_A}\,(\mathsf{abs}\,\mathsf{B}\,\mathsf{C}\,\mathsf{e}\,\mathsf{s}\,\mathsf{T_B}\,\mathsf{t_C})\,(\mathsf{Q_B}, \mathsf{Q_C}, \mathsf{liftE}, \mathsf{lift_{T_B}}, \mathsf{lift_{t_C}})\\ &\quad = \mathsf{cabs}\,\mathsf{A}\,\mathsf{B}\,\mathsf{C}\,\mathsf{Q_A}\,\mathsf{Q_B}\,\mathsf{Q_C}\,\mathsf{e}\,\mathsf{s}\,\mathsf{T_B}\,\mathsf{t_C}\,\mathsf{liftE}\,\mathsf{lift_{T_B}}\,\mathsf{p_C}\end{aligned}$$

where $(\mathsf{Q_B}, \mathsf{Q_C}, \mathsf{liftE}, \mathsf{lift_{T_B}}, \mathsf{lift_{t_C}}) : \mathsf{LTerm^\wedge\ A\ Q_A\ (abs\ B\ C\ e\ s\ T_B\ t_C)}$, i.e.,

- $\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}$;
- $\mathsf{Q_C} : \mathsf{C} \to \mathsf{Set}$;
- $\mathsf{liftE} : \mathsf{Equal^\wedge\ A\ (B \to C)\ Q_A\ (Arr^\wedge\ B\ C\ Q_B\ Q_C)\ e}$;
- $\mathsf{lift_{T_B}} : \mathsf{LType^\wedge\ B\ Q_B\ T_B}$;
- $\mathsf{lift_{t_C}} : \mathsf{LTerm^\wedge\ C\ Q_C\ T_C}$;

and $\mathsf{p_C}$ is defined as:

$$\mathsf{p_C = dIndLTerm\ P\ cvar\ cabs\ capp\ clistc\ C\ Q_C\ t_C\ lift_{t_C} : P\ C\ Q_C\ t_C}$$

For the app case, let $\mathsf{t_A = app\ B\ t_{BA}\ t_B}$ and define

$$\mathsf{dIndLTerm\ P\ cvar\ cabs\ capp\ clistc\ A\ Q_A\ (app\ B\ t_{BA}\ t_B)\ (Q_B, list_{t_{BA}}, list_{t_B}) = capp\ A\ B\ Q_A\ Q_B\ t_{BA}\ t_B\ p_{BA}\ p_B}$$

where $(\mathsf{Q_B}, \mathsf{lift_{t_{BA}}}, \mathsf{list_{t_B}}) : \mathsf{LTerm^\wedge\ A\ Q_A\ (app\ B\ t_{BA}\ t_B)}$, i.e.,

- $\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}$;
- $\mathsf{lift_{t_{BA}}} : \mathsf{LTerm^\wedge\ (B \to A)\ (Arr^\wedge\ B\ A\ Q_B\ Q_A)\ t_{BA}}$;
- $\mathsf{lift_{t_B}} : \mathsf{LTerm^\wedge\ B\ Q_B\ t_B}$;

and $\mathsf{p_{BA}}$ and $\mathsf{p_B}$ and are defined as:

$$\mathsf{p_{BA} = dIndLTerm\ P\ cvar\ cabs\ capp\ clistc\ (B \to A)\ (Arr^\wedge\ B\ A\ Q_B\ Q_A)\ t_{BA}\ lift_{t_{BA}} : P\ (B \to A)\ (Arr^\wedge\ B\ A\ Q_B\ Q_A)\ t_{BA}}$$
$$\mathsf{p_B = dIndLTerm\ P\ cvar\ cabs\ capp\ clistc\ B\ Q_B\ t_B\ lift_{t_B} : P\ B\ Q_B\ t_B}$$

Finally, for the listC case, let $\mathsf{t_A = listC\ B\ e\ ts}$ and define

$$\mathsf{dIndLTerm\ P\ cvar\ cabs\ capp\ clistc\ A\ Q_A\ (listC\ B\ e\ ts)\ (Q_B, liftE, lift_{List}) = clistc\ A\ B\ Q_A\ Q_B\ e\ ts\ liftE\ p_{List}}$$

where $(\mathsf{Q_B}, \mathsf{liftE}, \mathsf{lift_{List}}) : \mathsf{LTerm^\wedge\ A\ Q_A\ (listC\ B\ e\ ts)}$, i.e.,

- $\mathsf{Q_B} : \mathsf{B} \to \mathsf{Set}$;
- $\mathsf{listE} : \mathsf{Equal^\wedge\ A\ (List\ B)\ Q_A\ (List^\wedge\ B\ Q_B)\ e}$;
- $\mathsf{lift_{List}} : \mathsf{List^\wedge\ (LTerm\ B)\ (LTerm^\wedge\ B\ Q_B)\ ts}$;

and $\mathsf{p_{List}}$ is defined as:

$$\mathsf{p_{List} = liftListMap\ (LTerm\ B)\ (LTerm^\wedge\ B\ Q_B)\ (P\ B\ Q_B)\ p_{ts}\ ts\ lift_{List} : List^\wedge\ (LTerm\ B)\ (P\ B\ Q_B)\ ts}$$

and $\mathsf{p_{ts}}$ is defined as:

$$\mathsf{p_{ts} = dIndLTerm\ P\ cvar\ cabs\ capp\ clistc\ B\ Q_B : PredMap\ (LTerm\ B)\ (LTerm^\wedge\ B\ Q_B)\ (P\ B\ Q_B)}$$

where

$$\mathsf{liftListMap} : \forall\ (\mathsf{A} : \mathsf{Set}) \to (\mathsf{Q_A\ Q'_A} : \mathsf{A} \to \mathsf{Set}) \to \mathsf{PredMap\ A\ Q_A\ Q'_A} \to \mathsf{PredMap\ (List\ A)\ (List^\wedge\ A\ Q_A)\ (List^\wedge\ A\ Q'_A)}$$

takes a morphism of predicates and produces a morphism of lifted predicates. PredMap is the type constructor that produces the type of morphisms between predicates, and it is defined as follows:

$$\mathsf{PredMap} : \forall\ (\mathsf{A} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to (\mathsf{A} \to \mathsf{Set}) \to \mathsf{Set}$$
$$\mathsf{PredMap\ A\ Q_A\ Q'_A} = \forall\ (\mathsf{a} : \mathsf{A}) \to \mathsf{Q_A\ a} \to \mathsf{Q'_A\ a}$$

4.4 General case

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

$$\mathsf{data\ G} : \mathsf{Set}^\alpha \to \mathsf{Set\ where}$$
$$\mathsf{c_i\ :\ F_i\ G\ \overline{B} \to G(\overline{K_i\ \overline{B}})}$$

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

- $\mathsf{F_i} = \mathsf{F_i'} \times \mathsf{F_i''}$ where $\mathsf{F_i'}$ and $\mathsf{F_i''}$ have the same signature as $\mathsf{F_i}$ and are built recursively from the same induction rules.
- $\mathsf{F_i} = \mathsf{F_i'} + \mathsf{F_i''}$ where $\mathsf{F_i'}$ and $\mathsf{F_i''}$ have the same signature as $\mathsf{F_i}$ and are built recursively from the same induction rules.
- $\mathsf{F_i} = \mathsf{F_i'} \to \mathsf{F_i''}$ where $\mathsf{F_i'}$ does not contain the recursive variable, i.e., $\mathsf{F_i'} : \mathsf{Set}^\beta \to \mathsf{Set}$ is a type constructor of arity β, and $\mathsf{F_i''}$ has the same signature as $\mathsf{F_i}$ and is built recursively from the same induction rules.
- $\mathsf{F_i}\,\mathsf{G}\,\overline{\mathsf{B}} = \mathsf{G}(\overline{\mathsf{F_j}\,\overline{\mathsf{B}}})$ where none of the α-many $\mathsf{F_j}$ contains the recursive variable, i.e., $\mathsf{F_j} : \mathsf{Set}^\beta \to \mathsf{Set}$ is a type constructor of arity β for each $\mathsf{j} = 1, \ldots, \alpha$. Such restriction is necessary to prevent nesting, as that would break the induction rule as discussed in Section 5.
- $\mathsf{F_i}\,\mathsf{G}\,\overline{\mathsf{B}} = \mathsf{H}\,\overline{\mathsf{B}}$ where H is a type constructor of arity β not containing the recursive variable, i.e., $\mathsf{H} : \mathsf{Set}^\beta \to \mathsf{Set}$. Notice that this covers the case in which $\mathsf{F_i}$ is a closed type, so, in particular, the unit and empty types, $\mathsf{1}$ and $\mathsf{0}$, and the case in which $\mathsf{F_i}$ is a variable.
- $\mathsf{F_i}\,\mathsf{G}\,\overline{\mathsf{B}} = \mathsf{H}(\overline{\mathsf{F_k}\,\mathsf{G}\,\overline{\mathsf{B}}})$ where H is a γ-ary type constructor not containing the recursive variable, i.e., $\mathsf{H} : \mathsf{Set}^\gamma \to \mathsf{Set}$, and $\mathsf{F_k}$ has the same signature as $\mathsf{F_i}$ and is built recursively from the same induction rules, for every $\mathsf{k} = 1 \ldots \gamma$. Moreover, we require that H is not a GADT itself (but we allow it to be an ADT or even a nested type). This way we know that H admits functorial semantics [13], and thus there is a map function for $\mathsf{H}^\wedge$,

$$\mathsf{HLMap} : \forall(\overline{\mathsf{C} : \mathsf{Set}})(\overline{\mathsf{Q_C}\ \mathsf{Q_C'} : \mathsf{C} \to \mathsf{Set}}) \to \overline{\mathsf{PredMap}\,\mathsf{C}\,\mathsf{Q_C}\,\mathsf{Q_C'}} \to \mathsf{PredMap}\,(\mathsf{H}\,\overline{\mathsf{C}})\,(\mathsf{H}^\wedge\,\overline{\mathsf{C}}\,\overline{\mathsf{Q_C}})\,(\mathsf{H}^\wedge\,\overline{\mathsf{C}}\,\overline{\mathsf{Q_C'}})$$

  A concrete way to define HLMap is to proceed by induction on the structure of the type H, and give an inductive definition when H is an ADT or a nested type. Such details are not essential to the present discussion, and thus we omit them.

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

$$\mathsf{F_i}\,\mathsf{G}\,\overline{\mathsf{B}} := \mathsf{F_i'}\,\mathsf{G}\,\overline{\mathsf{B}} \times \mathsf{F_i''}\,\mathsf{G}\,\overline{\mathsf{B}} \mid \mathsf{F_i'}\,\mathsf{G}\,\overline{\mathsf{B}} + \mathsf{F_i''}\,\mathsf{G}\,\overline{\mathsf{B}} \mid \mathsf{F_i'}\,\overline{\mathsf{B}} \to \mathsf{F_i''}\,\mathsf{G}\,\overline{\mathsf{B}} \mid \mathsf{G}(\overline{\mathsf{F_j}\,\overline{\mathsf{B}}}) \mid \mathsf{H}\,\overline{\mathsf{B}} \mid \mathsf{H}(\overline{\mathsf{F_k}\,\mathsf{G}\,\overline{\mathsf{B}}})$$

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

Consider a generic GADT as defined above,

$$\begin{aligned} &\mathsf{data}\ \mathsf{G} : \mathsf{Set} \to \mathsf{Set}\ \mathsf{where} \\ &\quad \mathsf{c} : \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}} \to \mathsf{G}(\mathsf{K}\,\overline{\mathsf{B}}) \end{aligned} \tag{8}$$

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

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

which is the form we shall use from now on.

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

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

as

$$\mathsf{G}^\wedge\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}\,(\mathsf{c}\,\overline{\mathsf{B}}\,\mathsf{e}\,\mathsf{x}) = \sum[\overline{\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set}}]\,\mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{K}\,\overline{\mathsf{B}})\,\mathsf{Q}_\mathsf{A}\,(\mathsf{K}^\wedge\,\overline{\mathsf{B}}\,\overline{\mathsf{Q}_\mathsf{B}})\,\mathsf{e} \times \mathsf{F}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q}_\mathsf{B}}\,\mathsf{x}$$

where $\mathsf{Q}_\mathsf{A} : \mathsf{A} \to \mathsf{Set}$ and $\mathsf{c}\,\overline{\mathsf{B}}\,\mathsf{e}\,\mathsf{x} : \mathsf{G}\,\mathsf{A}$, i.e., $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{K}\,\overline{\mathsf{B}})$ and $\mathsf{x} : \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}}$. As already mentioned before, we also assume to have liftings for F,

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

and for K,

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

Finally, associate the function

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

to the constructor c.

The induction rule for G is

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

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

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

where $\mathsf{cc} : \mathsf{dIndC}\,\mathsf{P}$ and

- $\mathsf{c}\,\overline{\mathsf{B}}\,\mathsf{e}\,\mathsf{x} : \mathsf{G}\,\mathsf{A}$, i.e., $\mathsf{e} : \mathsf{Equal}\,\mathsf{A}\,(\mathsf{K}\,\overline{\mathsf{B}})$, and $\mathsf{x} : \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}}$;
- $(\overline{\mathsf{Q}_\mathsf{B}}, \mathsf{liftE}, \mathsf{liftF}) : \mathsf{G}^\wedge\,\mathsf{A}\,\mathsf{Q}_\mathsf{A}(\mathsf{c}\,\overline{\mathsf{B}}\,\mathsf{e}\,\mathsf{x})$, i.e., $\mathsf{Q}_\mathsf{B} : \mathsf{B} \to \mathsf{Set}$ for each B, $\mathsf{liftE} : \mathsf{Equal}^\wedge\,\mathsf{A}\,(\mathsf{K}\,\overline{\mathsf{B}})\,\mathsf{Q}_\mathsf{A}\,(\mathsf{K}^\wedge\,\overline{\mathsf{B}}\,\overline{\mathsf{Q}_\mathsf{B}})\,\mathsf{e}$, and $\mathsf{liftF} : \mathsf{F}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q}_\mathsf{B}}\,\mathsf{x}$.

Finally, the morphism of predicates

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

is defined by structural induction on F as follows:

- Case $\mathsf{F} = \mathsf{F}_1 \times \mathsf{F}_2$ where F_1 and F_2 have the same signature as F. We have that

  $$\mathsf{F}^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q}_\mathsf{B}} = \mathsf{Pair}^\wedge(\mathsf{F}_1\,\mathsf{G}\,\overline{\mathsf{B}})(\mathsf{F}_2\,\mathsf{G}\,\overline{\mathsf{B}})(\mathsf{F}_1^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q}_\mathsf{B}})(\mathsf{F}_2^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q}_\mathsf{B}})$$

  By inductive hypothesis, there exist morphisms of predicates

  $$\begin{aligned}&\mathsf{p}_1 : \mathsf{PredMap}\,(\mathsf{F}_1\,\mathsf{G}\,\overline{\mathsf{B}})\,((\mathsf{F}_1)^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q}_\mathsf{B}})((\mathsf{F}_1)^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q}_\mathsf{B}}) \\ &\mathsf{p}_2 : \mathsf{PredMap}\,(\mathsf{F}_2\,\mathsf{G}\,\overline{\mathsf{B}})\,((\mathsf{F}_2)^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q}_\mathsf{B}})((\mathsf{F}_2)^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{P}\,\overline{\mathsf{Q}_\mathsf{B}})\end{aligned}$$

  Thus, we define $\mathsf{p}(\mathsf{x}_1, \mathsf{x}_2)(\mathsf{liftF}_1, \mathsf{liftF}_2) = (\mathsf{p}_1\,\mathsf{x}_1\,\mathsf{liftF}_1, \mathsf{p}_2\,\mathsf{x}_2\,\mathsf{liftF}_2)$ for $\mathsf{x}_1 : \mathsf{F}_1\,\mathsf{G}\,\overline{\mathsf{B}}$, $\mathsf{liftF}_1 : \mathsf{F}_1^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q}_\mathsf{B}}\,\mathsf{x}_1$, $\mathsf{x}_2 : \mathsf{F}_2\,\mathsf{G}\,\overline{\mathsf{B}}$ and $\mathsf{liftF}_2 : \mathsf{F}_2^\wedge\,\mathsf{G}\,\overline{\mathsf{B}}\,\mathsf{G}^\wedge\,\overline{\mathsf{Q}_\mathsf{B}}\,\mathsf{x}_2$.
- Case $\mathsf{F} = \mathsf{F}_1 + \mathsf{F}_2$ where F_1 and F_2 have the same signature as F. Analogous to case $\mathsf{F} = \mathsf{F}_1 \times \mathsf{F}_2$.

- Case $F = F_1 \to F_2$ where F_1 does not contain the recursive variable, i.e., $F_1 : \mathsf{Set}^\beta \to \mathsf{Set}$, and F_2 has the same signature as F. We have that

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

  where $x : F\, G\, \overline{B} = F_1\, \overline{B} \to F_2\, G\, \overline{B}$. By inductive hypothesis, there exist a morphism of predicates

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

  Thus, we define $p\, x\, \mathsf{liftF} : F^\wedge\, G\, \overline{B}\, P\, \overline{Q_B}\, x$ for $\mathsf{liftF} : F^\wedge\, G\, \overline{B}\, G^\wedge\, \overline{Q_B}\, x$ as $p\, x\, \mathsf{liftF}\, z\, \mathsf{liftF}_1 = p_2(x\, z)(\mathsf{liftF}\, z\, \mathsf{liftF}_1)$ for $z : F_1\, \overline{B}$ and $\mathsf{liftF}_1 : F_1^\wedge\, \overline{B}\, \overline{Q_B}\, z$. Notice that F_1 not containing the recursive variable is a necessary restriction, as 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 $F_1^\wedge\, \overline{B}\, \overline{Q_B}\, z$.
- Case $F\, G\, \overline{B} = G(F'\, \overline{B})$ where F' does not contain the recursive variable, i.e., $F' : \mathsf{Set}^\beta \to \mathsf{Set}$. Thus, $F^\wedge\, G\, \overline{B}\, P\, \overline{Q_B} = P(F'\, \overline{B})(F'^\wedge\, \overline{B}\, \overline{Q_B})$. So, p is defined as

$$p = \mathsf{dIndG}\, P\, \mathsf{cc}\, (F'\, \overline{B})\, (F'^\wedge\, \overline{B}\, \overline{Q_B})$$

- Case $F\, G\, \overline{B} = H\, \overline{B}$ where H is a β-ary type constructor not containing the recursive variable, i.e., $H : \mathsf{Set}^\beta \to \mathsf{Set}$. In such case, $p : \mathsf{PredMap}(H\, \overline{B})(H^\wedge\, \overline{B}\, \overline{Q_B})(H^\wedge\, \overline{B}\, \overline{Q_B})$ is just the identity morphism of predicates.
- Case $F\, G\, \overline{B} = H(\overline{F_k\, G\, \overline{B}})$ where H is a γ-ary type constructor not containing the recursive variable, i.e., $H : \mathsf{Set}^\gamma \to \mathsf{Set}$, and F_k has the same signature as F, for every $k = 1 \ldots \gamma$. Moreover, we assume that H has an associated predicate-lifting function,

$$H^\wedge : \forall (\overline{C : \mathsf{Set}}) \to (\overline{C \to \mathsf{Set}}) \to H\, \overline{C} \to \mathsf{Set}$$

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

$$\forall (\overline{C : \mathsf{Set}})(\overline{Q_C\ Q'_C : C \to \mathsf{Set}}) \to \mathsf{PredMap}\, \overline{C}\, \overline{Q_C}\, \overline{Q'_C} \to \mathsf{PredMap}\, (H\, \overline{C})\, (H^\wedge\, \overline{C}\, \overline{Q_C})\, (H^\wedge\, \overline{C}\, \overline{Q'_C})$$

  That means that H cannot be a GADT, as GADTs have no functorial semantics [11]. and incur in the issue exposed in Section 5, but it can be an ADT or even a nested type as those types have functorial semantics [12,11]. Thus,

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

  By induction hypothesis, there is a morphism of predicates

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

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

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

5 Induction for GADTs with nesting

In the previous sections, we derive induction rules for examples of GADTs that do not feature nesting, in the sense that their constructors contain no nested calls of the recursive variable, as truly nested types (such as Bush, Equation 2.1) do. Since both nested types and GADTs without nesting admit induction rules, as seen in the previous sections, it is just natural to expect that GADTs with nesting would as well. Surprisingly, that is not the case: indeed, the induction principle generally relies on (unary) parametricity of the semantic interpretation, and in the case of nested types it also relies on functorial semantics [13], but GADTs cannot admit both functorial and parametric semantics at the same time [11]. In this section we show how induction for GADTs featuring nesting goes wrong by analyzing the following concrete example of such a type.

$$\begin{aligned} &\mathsf{data}\, G\, (a : \mathsf{Set}) : \mathsf{Set}\, \mathsf{where} \\ &\quad C :\ G(G\, a) \to G(a \times a) \end{aligned} \tag{9}$$

The constructor C can be rewritten as

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

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

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

is defined as

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

where $\mathsf{Q}_{\mathsf{a}} : \mathsf{a} \to \mathsf{Set}$, $\mathsf{e} : \mathsf{Equal}\,\mathsf{a}\,(\mathsf{b} \times \mathsf{b})$ and $\mathsf{x} : \mathsf{G}(\mathsf{G}\,\mathsf{b})$. Finally, let CC be the function

$$\begin{aligned}\lambda(\mathsf{P} : \forall(\mathsf{a} : \mathsf{Set}) \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{G}\,\mathsf{a} \to \mathsf{Set})\\ \to \forall(\mathsf{a}\,\mathsf{b} : \mathsf{Set})(\mathsf{Q}_{\mathsf{a}} : \mathsf{a} \to \mathsf{Set})(\mathsf{Q}_{\mathsf{b}} : \mathsf{b} \to \mathsf{Set})(\mathsf{e} : \mathsf{Equal}\,\mathsf{a}\,(\mathsf{b} \times \mathsf{b}))(\mathsf{x} : \mathsf{G}\,(\mathsf{G}\,\mathsf{b}))\\ \to \mathsf{Equal}^{\wedge}\,\mathsf{a}\,(\mathsf{b} \times \mathsf{b})\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{Pair}^{\wedge}\,\mathsf{b}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{b}}\,\mathsf{Q}_{\mathsf{b}})\,\mathsf{e} \to \mathsf{P}\,(\mathsf{G}\,\mathsf{b})\,(\mathsf{P}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{b}})\,\mathsf{x} \to \mathsf{P}\,\mathsf{a}\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{C}\,\mathsf{b}\,\mathsf{e}\,\mathsf{x})\end{aligned}$$

associated to the C constructor.

The induction rule for G is

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

Consistently with the previous examples, to validate the induction rule we try to define a term of the above type, DIG, as

$$\mathsf{DIG}\,\mathsf{P}\,\mathsf{cc}\,\mathsf{a}\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{C}\,\mathsf{b}\,\mathsf{e}\,\mathsf{x})\,(\mathsf{Q}_{\mathsf{b}}, \mathsf{L}_{\mathsf{E}}, \mathsf{L}_{\mathsf{G}}) = \mathsf{cc}\,\mathsf{a}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{a}}\,\mathsf{Q}_{\mathsf{b}}\,\mathsf{e}\,\mathsf{x}\,\mathsf{L}_{\mathsf{E}}\,\mathsf{p}$$

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

- $\mathsf{C}\,\mathsf{b}\,\mathsf{e}\,\mathsf{x} : \mathsf{G}\,\mathsf{a}$, i.e., $\mathsf{e} : \mathsf{Equal}\,\mathsf{a}\,(\mathsf{b} \times \mathsf{b})$ and $\mathsf{x} : \mathsf{G}(\mathsf{G}\,\mathsf{b})$;
- $(\mathsf{Q}_{\mathsf{b}}, \mathsf{L}_{\mathsf{E}}, \mathsf{L}_{\mathsf{G}}) : \mathsf{G}^{\wedge}\,\mathsf{a}\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{C}\,\mathsf{b}\,\mathsf{e}\,\mathsf{x})$, i.e., $\mathsf{Q}_{\mathsf{b}} : \mathsf{b} \to \mathsf{Set}$, $\mathsf{L}_{\mathsf{E}} : \mathsf{Equal}^{\wedge}\,\mathsf{a}\,(\mathsf{b} \times \mathsf{b})\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{Pair}^{\wedge}\,\mathsf{b}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{b}}\,\mathsf{Q}_{\mathsf{b}})\,\mathsf{e}$, and $\mathsf{L}_{\mathsf{G}} : \mathsf{G}^{\wedge}\,(\mathsf{G}\,\mathsf{b})\,(\mathsf{G}^{\wedge}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{b}})\,\mathsf{x}$.

We still need to define $\mathsf{p} : \mathsf{P}\,(\mathsf{G}\,\mathsf{b})\,(\mathsf{P}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{b}})\,\mathsf{x}$. We do so by using the induction rule and letting

$$\mathsf{p} = \mathsf{DIG}\,\mathsf{P}\,\mathsf{cc}\,(\mathsf{G}\,\mathsf{b})\,(\mathsf{P}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{b}})\,\mathsf{x}\,\mathsf{q}$$

where we still need to provide $\mathsf{q} : \mathsf{G}^{\wedge}\,(\mathsf{G}\,\mathsf{b})\,(\mathsf{P}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{b}})\,\mathsf{x}$. If we had the map function of $\mathsf{G}^{\wedge}$,

$$\mathsf{GLMap} : \forall(\mathsf{a} : \mathsf{Set})(\mathsf{Q}_{\mathsf{a}}\ \mathsf{Q}'_{\mathsf{a}} : \mathsf{a} \to \mathsf{Set}) \to \mathsf{PredMap}\,\mathsf{a}\,\mathsf{Q}_{\mathsf{a}}\,\mathsf{Q}'_{\mathsf{a}} \to \mathsf{PredMap}\,(\mathsf{G}\,\mathsf{a})\,(\mathsf{G}^{\wedge}\,\mathsf{a}\,\mathsf{Q}_{\mathsf{a}})\,(\mathsf{G}^{\wedge}\,\mathsf{a}\,\mathsf{Q}'_{\mathsf{a}})$$

then we would be able to define

$$\mathsf{q} = \mathsf{GLMap}\,(\mathsf{G}\,\mathsf{b})\,(\mathsf{G}^{\wedge}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{b}})\,(\mathsf{P}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{b}})\,(\mathsf{DIG}\,\mathsf{P}\,\mathsf{cc}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{b}})\,\mathsf{x}\,\mathsf{L}_{\mathsf{G}}$$

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

$$\mathsf{GLMap}\,\mathsf{a}\,\mathsf{Q}_{\mathsf{a}}\,\mathsf{Q}'_{\mathsf{a}}\,\mathsf{M}\,(\mathsf{C}\,\mathsf{b}\,\mathsf{e}\,\mathsf{x})\,(\mathsf{Q}_{\mathsf{b}}, \mathsf{L}_{\mathsf{E}}, \mathsf{L}_{\mathsf{G}}) = (\mathsf{Q}'_{\mathsf{b}}, \mathsf{L}'_{\mathsf{E}}, \mathsf{L}'_{\mathsf{G}})$$

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

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

$(\mathsf{Q}_{\mathsf{b}}, \mathsf{L}_{\mathsf{E}}, \mathsf{L}_{\mathsf{G}}) : \mathsf{G}^{\wedge}\,\mathsf{a}\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{C}\,\mathsf{b}\,\mathsf{e}\,\mathsf{x})$, i.e.,

- $\mathsf{Q}_{\mathsf{b}} : \mathsf{b} \to \mathsf{Set}$;
- $\mathsf{L}_{\mathsf{E}} : \mathsf{Equal}^{\wedge}\,\mathsf{a}\,(\mathsf{b} \times \mathsf{b})\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{Pair}^{\wedge}\,\mathsf{b}\,\mathsf{b}\,\mathsf{Q}_{\mathsf{b}}\,\mathsf{Q}_{\mathsf{b}})\,\mathsf{e}$;

- $\mathsf{L_G} : \mathsf{G^\wedge}\,(\mathsf{G}\,\mathsf{b})\,(\mathsf{G^\wedge}\,\mathsf{b}\,\mathsf{Q_b})\,\mathsf{x}$;

and $(\mathsf{Q'_b}, \mathsf{L'_E}, \mathsf{L'_G}) : \mathsf{G^\wedge}\,\mathsf{a}\,\mathsf{Q'_a}\,(\mathsf{C}\,\mathsf{b}\,\mathsf{e}\,\mathsf{x})$, i.e.,

- $\mathsf{Q'_b} : \mathsf{b} \to \mathsf{Set}$;
- $\mathsf{L'_E} : \mathsf{Equal^\wedge}\,\mathsf{a}\,(\mathsf{b} \times \mathsf{b})\,\mathsf{Q'_a}\,(\mathsf{Pair^\wedge}\,\mathsf{b}\,\mathsf{b}\,\mathsf{Q'_b}\,\mathsf{Q'_b})\,\mathsf{e}$;
- $\mathsf{L'_G} : \mathsf{G^\wedge}\,(\mathsf{G}\,\mathsf{b})\,(\mathsf{G^\wedge}\,\mathsf{b}\,\mathsf{Q'_b})\,\mathsf{x}$;

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

At a deeper level, the fundamental issue is that the Equal type does not have functorial semantics, so that having morphisms $\mathsf{A} \to \mathsf{A'}$ and $\mathsf{B} \to \mathsf{B'}$ 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-completion semantics. Since we are interested in induction rules, we considered 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 nestes types, indeed, GADTs do not admit a semantic interpretation that is both parametric and functorial [11].

6 Applications

Can get rid of Maybe using non-empty lists and postulates

In this section we use deep induction for the LTerm GADT to extract the type from a lambda term. We have a predicate

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

that takes a lambda term and produces its type (using Maybe to represent potential failure). We want to show this predicate is satisfied for every element of $\mathsf{LTerm}\,\mathsf{a}$. Because of the ListC constructor, 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 deep induction, we want to prove:

$$\mathsf{getTypeProof} : \forall\,(\mathsf{a} : \mathsf{Set}) \to (\mathsf{t} : \mathsf{LTerm}\,\mathsf{a}) \to \mathsf{getType}\,\mathsf{a}\,\mathsf{t}$$

which we prove by

$$\mathsf{getTypeProof}\,\mathsf{a}\,\mathsf{t} = \mathsf{DILTerm}\,(\lambda\,\mathsf{b}\,\mathsf{Q_b}\,\mathsf{t} \to \mathsf{getType}\,\mathsf{b}\,\mathsf{t})\,\mathsf{gtVar}\,\mathsf{gtABs}\,\mathsf{gtApp}\,\mathsf{gtListC}\,\mathsf{a}\;\mathsf{K1}\;\mathsf{t}\,(\mathsf{LTerm^\wedge K1}\,\mathsf{a}\,\mathsf{t})$$

where $\mathsf{K1} : \mathsf{a} \to \mathsf{Set}$ is the constantly true predicate:

$$\mathsf{K1}\,\mathsf{x} = \top$$

and $\mathsf{LTerm^\wedge K1}\,\mathsf{a}\,\mathsf{t} : \mathsf{LTerm^\wedge}\,\mathsf{a}\,\mathsf{K1}\,\mathsf{t}$. Notice that there is no space in $\mathsf{LTerm^\wedge K1}$, because

$$\mathsf{LTerm^\wedge K1} : \forall(\mathsf{a} : \mathsf{Set})(\mathsf{t} : \mathsf{LTermA}) \to \mathsf{LTerm^\wedge}\,\mathsf{a}\,\mathsf{K1}\,\mathsf{t}$$

is a function that we will define. In addition to defining $\mathsf{LTerm^\wedge K1}$, we also have to give a proof for each constructor $\mathsf{Var}, \mathsf{Abs}, \mathsf{App}, \mathsf{ListC}$:

$$\mathsf{gtVar} : \forall(\mathsf{a} : \mathsf{Set})(\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set})(\mathsf{s} : \mathsf{String})(\mathsf{T_a} : \mathsf{LType}\,\mathsf{a}) \to \mathsf{LType^\wedge}\,\mathsf{a}\,\mathsf{Q_a}\,\mathsf{T_a} \to \mathsf{Maybe}\,(\mathsf{LType}\,\mathsf{a})$$

$$\begin{aligned}\mathsf{gtAbs} : \forall(\mathsf{a\,b\,c} : \mathsf{Set})(\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set})(\mathsf{Q_b} : \mathsf{b} \to \mathsf{Set})(\mathsf{Q_c} : \mathsf{c} \to \mathsf{Set})(\mathsf{e} : \mathsf{Equal\,a\,(b \to c)})(\mathsf{s} : \mathsf{String}) \\ (\mathsf{T_b} : \mathsf{LType\,b})(\mathsf{t_c} : \mathsf{LTerm\,c}) \to \mathsf{Equal^\wedge\,a\,(b \to c)\,Q_a\,(Arr^\wedge\,b\,c\,Q_b\,Q_c)\,e} \to \mathsf{LType^\wedge\,b\,Q_b\,T_b} \\ \to \mathsf{Maybe\,(LType\,c)} \to \mathsf{Maybe\,(LType\,a)}\end{aligned}$$

$$\begin{aligned}\mathsf{gtApp} : \forall(\mathsf{a\,b} : \mathsf{Set})(\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set})(\mathsf{Q_b} : \mathsf{b} \to \mathsf{Set})(\mathsf{t_{ba}} : \mathsf{LTerm\,(b \to a)})(\mathsf{t_b} : \mathsf{LTerm\,b}) \\ \to \mathsf{Maybe\,(LType\,(b \to a))} \to \mathsf{Maybe\,(LType\,b)} \to \mathsf{Maybe\,(LType\,a)}\end{aligned}$$

$$\begin{aligned}\mathsf{gtListC} : \forall(\mathsf{a\,b} : \mathsf{Set})(\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set})(\mathsf{Q_b} : \mathsf{b} \to \mathsf{Set})(\mathsf{e} : \mathsf{Equal\,a\,(List\,b)})(\mathsf{ts} : \mathsf{List\,(LTerm\,b)}) \\ \to \mathsf{Equal^\wedge\,a\,(List\,b)\,Q_a\,(List^\wedge\,b\,Q_b)\,e} \to \mathsf{List^\wedge\,(LTerm\,b)\,(getType\,b)\,ts} \to \mathsf{Maybe\,(LType\,a)}\end{aligned}$$

For variables we simply return the type $\mathsf{T_a}$, and the cases for abstraction and application are similar. The interesting case is gtListC, in which we have to use the results of $(\mathsf{List^\wedge\,(LTerm\,b)\,(getType\,b)\,ts})$ in order to extract the type of one of the terms in the list. To define gtListC we pattern-match on the list of terms ts.

If ts is the empty list (denoted by []), we cannot extract a type, so we return nothing.
Maybe handle this case differently, by using non-empty lists, for example

$$\mathsf{gtListC\,a\,b\,Q_a\,Q_b\,e\,[]\,L_e L_{ts} = nothing}$$

If ts is a non-empty list, we pattern match on $\mathsf{L_{ts}}$ and use the result to construct the type we need:

$$\begin{aligned}\mathsf{gtListC\,a\,b\,Q_a\,Q_b\,e\,(t :: ts)\,L_e(nothing, L_{ts}) = nothing} \\ \mathsf{gtListC\,a\,b\,Q_a\,Q_b\,e\,(t :: ts)\,L_e(just\,T_b, L_{ts}) = just\,(TList\,b\,e\,T_b)}\end{aligned}$$

where $\mathsf{e} : \mathsf{Equal\,a\,(List\,b)}$ and $\mathsf{T_b} : \mathsf{LType\,b}$.

6.1 Defining LTerm^K1

Maybe this section can be deleted by assuming $\mathsf{LTerm^\wedge\,a\,K1\,t = K1}$. Maybe we can say this derives from parametricity. Currently we say that $\mathsf{K1 \times K1 = K1}$ 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 to define getTypeProof is a function

$$\mathsf{LTerm^\wedge K1} : \forall(\mathsf{a} : \mathsf{Set}) \to (\mathsf{t} : \mathsf{LTermA}) \to \mathsf{LTerm^\wedge\,a\,K1\,t}$$

that provides a proof of $\mathsf{LTerm^\wedge\,a\,K1\,t}$ for any term $\mathsf{t} : \mathsf{LTerm\,a}$. Because LTerm^ is defined in terms of LType^, Arr^, and List^, we will need analogous functions for these liftings as well. We only give the definition of LTerm^K1, but the definitions for LType^, Arr^, and List^ are analogous.

LTerm^K1 is defined by pattern matching on the lambda term t. For the Var case, let $\mathsf{t} = (\mathsf{Var\,s\,T_a})$ and define

$$\mathsf{LTerm^\wedge K1\,a\,(Var\,s\,T_a) = LType^\wedge K1\,a\,T_a}$$

For the Abs case, let $\mathsf{t} = (\mathsf{Abs\,b\,c\,e\,s\,T_b\,t_c})$ and recall the definition of LTerm^ for the Abs constructor, instantiating the predicate $\mathsf{Q_a}$ to K1:

$$\begin{aligned}\mathsf{LTerm^\wedge\,a\,K1\,(Abs\,b\,c\,e\,s\,T_b\,t_c)} \\ = \exists(\mathsf{Q_b} : \mathsf{b} \to \mathsf{Set})(\mathsf{Q_c} : \mathsf{c} \to \mathsf{Set}) \to \mathsf{Equal^\wedge\,a\,(b \to c)\,K1\,(Arr^\wedge\,b\,c\,Q_b\,Q_c)\,e} \\ \times \mathsf{LType^\wedge\,b\,Q_b\,T_b} \times \mathsf{LTerm^\wedge\,c\,Q_c\,t_c}\end{aligned}$$

so to define the Abs case of LTerm^K1, we need a proof of

$$\mathsf{Equal^\wedge\,a\,(b \to c)\,K1\,(Arr^\wedge\,b\,c\,Q_b\,Q_c)\,e}$$

i.e., that $\mathsf{K1}$ is (extensionally) equal to the lifting $(\mathsf{Arr}^\wedge\,\mathsf{b}\,\mathsf{c}\,\mathsf{Q_b}\,\mathsf{Q_c})$ for some predicates $\mathsf{Q_b}, \mathsf{Q_c}$. The only reasonable choice for $\mathsf{Q_b}$ and $\mathsf{Q_c}$ is to let both be $\mathsf{K1}$, which means we need a proof of:

$$\mathsf{Equal}^\wedge\,\mathsf{a}\,(\mathsf{b}\to\mathsf{c})\,\mathsf{K1}\,(\mathsf{Arr}^\wedge\,\mathsf{b}\,\mathsf{c}\,\mathsf{K1}\,\mathsf{K1})\,\mathsf{e}$$

Since we are working with proof-relevant predicates (i.e., functions into Set rather than functions into Bool), the lifting $(\mathsf{Arr}^\wedge\,\mathsf{b}\,\mathsf{c}\,\mathsf{K1}\,\mathsf{K1})$ of $\mathsf{K1}$ to arrow types is not identical to $\mathsf{K1}$ 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

$$\mathsf{Equal}^\wedge\mathsf{ArrK1} : \mathsf{Equal}^\wedge\,\mathsf{a}\,(\mathsf{b}\to\mathsf{c})\,\mathsf{K1}\,(\mathsf{Arr}^\wedge\,\mathsf{b}\,\mathsf{c}\,\mathsf{K1}\,\mathsf{K1})\,\mathsf{e}$$

and define the Abs case of $\mathsf{LTerm}^\wedge\mathsf{K1}$ as

$$\mathsf{LTerm}^\wedge\mathsf{K1}\,\mathsf{a}\,(\mathsf{Abs}\,\mathsf{b}\,\mathsf{c}\,\mathsf{e}\,\mathsf{s}\,\mathsf{T_b}\,\mathsf{t_c}) = (\mathsf{K1}, \mathsf{K1}, \mathsf{Equal}^\wedge\mathsf{ArrK1}, \mathsf{LType}^\wedge\mathsf{K1}\,\mathsf{b}\,\mathsf{T_b}\,, \mathsf{LTerm}^\wedge\mathsf{K1}\,\mathsf{c}\,\mathsf{t_c})$$

For the App case, let $\mathsf{t} = (\mathsf{App}\,\mathsf{b}\,\mathsf{t_{ba}}\,\mathsf{t_b})$ and just as we did for the Abs case, recall the definition of $\mathsf{LTerm}^\wedge\,\mathsf{a}\,(\mathsf{App}\,\mathsf{b}\,\mathsf{t_{ba}}\,\mathsf{t_b})$ with all of the predicates instantiated with $\mathsf{K1}$:

$$\mathsf{LTerm}^\wedge\,(\mathsf{b}\to\mathsf{a})\,(\mathsf{Arr}^\wedge\,\mathsf{b}\,\mathsf{a}\,\mathsf{K1}\,\mathsf{K1})\,\mathsf{t_{ba}} \times \mathsf{LTerm}^\wedge\,\mathsf{b}\,\mathsf{K1}\,\mathsf{t_b}$$

The second component can be given using $\mathsf{LTerm}^\wedge\mathsf{K1}$, and we can define the first component using a proof of

$$\mathsf{LTerm}^\wedge\,(\mathsf{b}\to\mathsf{a})\,\mathsf{K1}\;\mathsf{t_{ba}}$$

and a map-like function

$$\mathsf{LTerm}^\wedge\mathsf{EqualMap} : \forall\,\{\mathsf{a}:\mathsf{Set}\}\to(\mathsf{Q_a}\,\mathsf{Q'_a} : \mathsf{a}\to\mathsf{Set})\to(\mathsf{Equal}^\wedge\,\mathsf{a}\,\mathsf{a}\,\mathsf{Q_a}\,\mathsf{Q'_a}\,\mathsf{rfl}) \to \mathsf{PredMap}\,(\mathsf{LTerm}^\wedge\,\mathsf{a}\,\mathsf{Q_a})\,(\mathsf{LTerm}^\wedge\,\mathsf{a}\,\mathsf{Q'_a})$$

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. But it is worth noting that while a true HLMap function, which takes a *morphism* of predicates instead of a proof of equality, cannot be defined for GADTs in general, an analogue of $\mathsf{LTerm}^\wedge\mathsf{EqualMap}$ should be definable for every GADT. These analogues of $\mathsf{LTerm}^\wedge\mathsf{EqualMap}$ will be required to define $\mathsf{G}^\wedge\mathsf{K1}$ whenever G has a constructor of the form $(\mathsf{c} : \mathsf{G}\,(\mathsf{F}\,\mathsf{b})\to\mathsf{G}\,(\mathsf{K}\,\mathsf{b}))$.

Using $\mathsf{LTerm}^\wedge\mathsf{EqualMap}$, we can define the App case of $\mathsf{LTerm}^\wedge\mathsf{K1}$ as

$$\mathsf{LTerm}^\wedge\mathsf{K1}\,\mathsf{a}\,(\mathsf{App}\,\mathsf{b}\;\mathsf{t_{ba}}\,\mathsf{t_b}) = (\mathsf{K1}, \mathsf{L_{Arr^\wedge\,K1}}, \mathsf{LTerm}^\wedge\mathsf{K1}\,\mathsf{b}\,\mathsf{t_b})$$

where $\mathsf{L_{Arr^\wedge\,K1}} : \mathsf{LTerm}^\wedge\,(\mathsf{b}\to\mathsf{a})\,(\mathsf{Arr}^\wedge\,\mathsf{b}\,\mathsf{a}\,\mathsf{K1}\,\mathsf{K1})\,\mathsf{t_{ba}}$ is defined as

$$\mathsf{L_{Arr^\wedge\,K1}} = \mathsf{LTerm}^\wedge\mathsf{EqualMap}\,\mathsf{K1}\,(\mathsf{Arr}^\wedge\,\mathsf{b}\,\mathsf{a}\,\mathsf{K1}\,\mathsf{K1})\;\mathsf{Equal}^\wedge\mathsf{ArrK1}\;\;\mathsf{t_{ba}}\;\mathsf{L_{K1}}$$

where $\mathsf{L_{K1}} = \mathsf{LTerm}^\wedge\mathsf{K1}\,(\mathsf{b}\to\mathsf{a})\,\mathsf{t_{ba}} : \mathsf{LTerm}^\wedge\,(\mathsf{b}\to\mathsf{a})\,\mathsf{K1}\,\mathsf{t_{ba}}$.

Finally, we define the ListC case for $\mathsf{LTerm}^\wedge\mathsf{K1}$. Let $\mathsf{t} = (\mathsf{ListC}\,\mathsf{b}\,\mathsf{e}\,\mathsf{ts})$ 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 $\mathsf{K1}$:

$$\mathsf{Equal}^\wedge\,\mathsf{a}\,(\mathsf{List}\,\mathsf{b})\,\mathsf{K1}\,(\mathsf{List}^\wedge\,\mathsf{b}\,\mathsf{K1})\,\mathsf{e} \times \mathsf{List}^\wedge\,(\mathsf{LTerm}\,\mathsf{b})\,(\mathsf{LTerm}^\wedge\,\mathsf{b}\,\mathsf{K1})\,\mathsf{ts}$$

We can give the first component by assuming a proof $\mathsf{Equal}^\wedge\mathsf{ListK1} : \mathsf{Equal}^\wedge\,\mathsf{a}\,(\mathsf{List}\,\mathsf{b})\,\mathsf{K1}\,(\mathsf{List}^\wedge\,\mathsf{b}\,\mathsf{K1})\,\mathsf{e}$, but for the second component we again have multiple liftings nested together. In this case, we can get a proof of

$$\mathsf{List}^\wedge\,(\mathsf{LTerm}\,\mathsf{b})\,(\mathsf{LTerm}^\wedge\,\mathsf{b}\,\mathsf{K1})\,\mathsf{ts}$$

using

$$\mathsf{List}^\wedge\mathsf{map} : \forall\,(\mathsf{a}:\mathsf{Set})\to(\mathsf{Q_a}\,\mathsf{Q'_a} : \mathsf{a}\to\mathsf{Set})\to\mathsf{PredMap}\,\mathsf{Q_a}\,\mathsf{Q'_a}\to\mathsf{PredMap}\,(\mathsf{List}^\wedge\,\mathsf{a}\,\mathsf{Q_a})\,(\mathsf{List}^\wedge\,\mathsf{a}\,\mathsf{Q'_a})$$

to map a morphism of predicates

$$\mathsf{PredMap}\,(\mathsf{K1})\,(\mathsf{LTerm\hat{}}\,\mathsf{b}\,\mathsf{K1})$$

to a morphism of lifted predicates

$$\mathsf{PredMap}\,(\mathsf{List\hat{}}\,(\mathsf{LTerm}\,\mathsf{b})\,\mathsf{K1})\,(\mathsf{List\hat{}}\,(\mathsf{LTerm}\,\mathsf{b})\,(\mathsf{LTerm\hat{}}\,\mathsf{b}\,\mathsf{K1}))$$

We define the ListC case of LTerm^K1 as

$$\mathsf{LTerm\hat{}K1}\,\mathsf{a}\,(\mathsf{ListC}\,\mathsf{b}\,\mathsf{e}\,\mathsf{ts}) = (\mathsf{K1},\ \mathsf{Equal\hat{}ListK1},\ \mathsf{L}_{\mathsf{List\hat{}LTerm\hat{}K1}})$$

where $\mathsf{L}_{\mathsf{List\hat{}LTerm\hat{}K1}} : \mathsf{List\hat{}}\,(\mathsf{LTerm}\,\mathsf{b})\,(\mathsf{LTerm\hat{}}\,\mathsf{b}\,\mathsf{K1})\,\mathsf{ts}$

$$\mathsf{L}_{\mathsf{List\hat{}LTerm\hat{}K1}} = \mathsf{List\hat{}map}\,(\mathsf{LTerm}\,\mathsf{b})\,\mathsf{K1}\,(\mathsf{LTerm\hat{}}\,\mathsf{b}\,\mathsf{K1})\,\mathsf{m}_{\mathsf{K1}}\,\mathsf{ts}\,(\mathsf{List\hat{}K1}\,(\mathsf{LTerm}\,\mathsf{b})\,\mathsf{ts})$$

and $\mathsf{m}_{\mathsf{K1}} : \mathsf{PredMap}\,(\mathsf{K1})\,(\mathsf{LTerm\hat{}}\,\mathsf{b}\,\mathsf{K1})$

$$\mathsf{m}_{\mathsf{K1}}\,\mathsf{t}\,* = \mathsf{LTerm\hat{}K1}\,\mathsf{b}\,\mathsf{t}$$

where $*$ is the single element of $(\mathsf{K1}\,\mathsf{t})$. The use of List^map is required in the ListC case because ListC takes an argument of type List (LTerm b). The same technique can be used to define G^K1 whenever G has a constructor of the form $(\mathsf{c} : \mathsf{F}\,(\mathsf{G}\,\mathsf{a}) \to \mathsf{G}\,(\mathsf{K}\,\mathsf{b}))$. We only allow constructors of this form when F is a nested type or ADT, so we are guaranteed to have a F^map function.

6.2 Liftings of K1

To provide a proof of $\mathsf{G\hat{}}\,\mathsf{a}\,\mathsf{K1}\,\mathsf{t}$ for every term $\mathsf{t} : \mathsf{G}\,\mathsf{a}$, we need to know that the lifting of K1 by a type H is extensionally equal to K1 on H. For example, we might need a proof that $\mathsf{Pair\hat{}}\,\mathsf{a}\,\mathsf{b}\,\mathsf{K1}\,\mathsf{K1}$ is equal to the predicate K1 on pairs. Given a pair $(\mathsf{x},\mathsf{y}) : \mathsf{a} \times \mathsf{b}$, we have

$$\begin{aligned} &\mathsf{Pair\hat{}}\,\mathsf{a}\,\mathsf{b}\,\mathsf{K1}\,\mathsf{K1}(\mathsf{x},\mathsf{y}) \\ &= \mathsf{K1}\,\mathsf{x} \times \mathsf{K1}\,\mathsf{y} \\ &= \top \times \top \end{aligned}$$

while

$$\mathsf{K1}\,(\mathsf{x},\mathsf{y}) = \top$$

and while these types are not equal they are clearly isomorphic. So for simplicity of presentation, we assume $(\mathsf{F\hat{}}\,\mathsf{a}\,\mathsf{K1})$ is equal to K1 for every nested type and ADT F.

7 Conclusion/Related work

Mention Patricia/Neil2008 paper

No induction with primitive representation (reference Haskell Symposium paper and [12] and paper Patricia Neil Clement 2010)

8 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

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] 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.

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

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

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

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

[13] 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.

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

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

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

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

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

[19] 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?

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

[21] 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.

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

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

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