

Deep Induction for GADTs

Patricia Johann¹ Enrico Ghiorzi² Daniel Jeffries³

*Department of Computer Science
Appalachian State University*

Abstract

Abstract goes here.

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

1 Introduction

Introduction goes here.

2 Deep induction for ADTs and nested types

2.1 Syntax of ADTs and nested types

(Polynomial) algebraic data types (ADTs), both built-in and user-defined, have long been at the core of functional languages such as Haskell, ML, Agda, Epigram, and Idris. ADTs are used extensively in functional programming to structure computations, to express invariants of the data over which computations are defined, and to ensure the type safety of programs specifying those computations. ADTs include unindexed types, such as the type of natural numbers, and types indexed over other types, such as the quintessential example of an ADT, the type of lists (here coded in Agda)

$$\begin{aligned}
 \text{data List} &: \text{Set} \rightarrow \text{Set} \text{ where} \\
 \text{nil} &: \text{List A} \\
 \text{cons} &: \text{A} \rightarrow \text{List A} \rightarrow \text{List A}
 \end{aligned} \tag{1}$$

Notice that all occurrences of `List` in the above encoding are instantiated at the same index `A`. Thus, the instances of `List` at various indices are defined independently from one another. That is a defining feature of ADTs: an ADT defines a *family of inductive types*, one for each index type.

Over time, there has been a notable trend toward data types whose non-regular indexing can capture invariants and other sophisticated properties that can be used for program verification and other applications.

¹ Email: johannp@appstate.edu

² Email: ghiorzie@appstate.edu

³ Email: jeffriesd@appstate.edu

A simple example of such a type is given by Bird and Meertens' [2] prototypical *nested type*

$$\begin{aligned}
 \text{data PTree} &: \text{Set} \rightarrow \text{Set} \text{ where} \\
 \text{pLeaf} &: A \rightarrow \text{PTree } A \\
 \text{pNode} &: \text{PTree } (A \times A) \rightarrow \text{PTree } A
 \end{aligned} \tag{2}$$

of perfect trees, which can be thought of as constraining lists to have lengths that are powers of 2. In the above code, the constructor `PNode` uses data of type `PNode (A × A)` to construct data of type `PNode A`. Thus, it is clear that the instantiations of `PNode` at various indices cannot be defined independently, so that the entire family of types must actually be defined at once. A nested type thus defines not a family of inductive types, but rather an *inductive family of types*.

Nested types include simple nested types, like perfect trees, none of whose recursive occurrences occur below another type constructor, and *truly* nested types, such as the nested type

$$\begin{aligned}
 \text{data Bush} &: \text{Set} \rightarrow \text{Set} \text{ where} \\
 \text{bNil} &: \text{Bush } A \\
 \text{bCons} &: A \rightarrow \text{Bush } (\text{Bush } A) \rightarrow \text{Bush } A
 \end{aligned} \tag{3}$$

of bushes, whose recursive occurrences appear below their own type constructors. Note that, while the constructors of a nested type can contain occurrences of the type instantiated at any index, the return types of its constructors still have to be the same type instance of the type being defined. In other words, all constructors of `PTree A` have to return an element of type `PTree A`, and all constructors of `Bush A` have to return an element of type `Bush A`.

2.2 Induction principles for ADTs and nested types

An induction principle for a data type allows to prove that a predicate holds for every element of that data type, provided that it holds for every element inductively produced by the type's constructors. In this paper, we are interested in induction principles for proof-relevant predicates. A proof-relevant predicate on a type $A : \text{Set}$ is a function $A \rightarrow \text{Set}$ mapping each $x : A$ to the set of proofs that the predicate holds for x . For example, the induction principle for `List` is

$$\forall (A : \text{Set}) (P : \text{List } A \rightarrow \text{Set}) \rightarrow P \text{ Nil} \rightarrow (\forall (a : A) (ls : \text{List } A) \rightarrow P \text{ ls} \rightarrow P (\text{Cons } a \text{ ls})) \rightarrow \forall (ls : \text{List } A) \rightarrow P \text{ ls}$$

Note that the data inside a structure of type `List` is treated monolithically (i.e., ignored) by this induction rule. Indeed, the induction rule inducts over only the top-level structures of data types, leaving any data internal to the top-level structure untouched. Since this kind of induction principle is only concerned with the structure of the type, and unconcerned with the contained data, we will then refer to it as *structural induction*.

We can extend such a structural induction principle to some nested types, such as `PTree`. The only difference from the induction principle for ADTs is that, since a nested type is defined as a whole inductive family of types at once, its induction rule has to necessarily involve a polymorphic predicate. Thus, the induction rule for `PTree` is

$$\begin{aligned}
 &\forall (P : \forall (A : \text{Set}) \rightarrow \text{PTree } A \rightarrow \text{Set}) \rightarrow (\forall (A : \text{Set}) (a : A) \rightarrow P A (\text{PLeaf } a)) \\
 &\rightarrow (\forall (A : \text{Set}) (\text{pt} : \text{PTree } (A \times A)) \rightarrow P (A \times A) \text{ pt} \rightarrow P A (\text{PNode pt})) \rightarrow \forall (A : \text{Set}) (\text{pt} : \text{PTree } A) \rightarrow P A \text{ pt}
 \end{aligned}$$

Structural induction principles cannot be extended to truly nested types, such as `Bush`. Instead, for such data types it is necessary to use a *deep induction* principle [6]. Such a principle, unlike structural induction, inducts over all of the structured data present, by traversing not just the outer structure with a predicate P , but also each data element contained in the data type with a custom predicate Q . This additional predicate is lifted to predicates on any internal structure 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.

For example, the deep induction rule for `Bush` is

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

where $\text{Bush}^\wedge : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{Bush } A \rightarrow \text{Set}$ lifts a predicate Q on data of type A to a predicate on data of type $\text{Bush } A$ asserting that Q holds for every element of type A contained in its argument bush. It is defined as

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

where $Q : A \rightarrow \text{Set}$ and $bb : \text{Bush } (\text{Bush } A)$.

Despite deep induction being motivated by the need to produce an induction principle for truly nested types, it can equally be applied to all other ADTs and nested types. For example, the deep induction principle for `List` is

$$\begin{aligned} & \forall (A : \text{Set})(P : \text{List } A \rightarrow \text{Set})(Q : A \rightarrow \text{Set}) \rightarrow P \text{ Nil} \rightarrow (\forall (a : A)(ls : \text{List } A) \rightarrow Q a \rightarrow P ls \rightarrow P (\text{Cons } a \text{ } ls)) \\ & \rightarrow \forall (ls : \text{List } A) \rightarrow \text{List}^\wedge A Q ls \rightarrow P ls \end{aligned}$$

where $\text{List}^\wedge : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{List } A \rightarrow \text{Set}$ lifts a predicate Q on data of type A to a predicate on data of type $\text{List } A$ asserting that Q holds for every element of its argument list.

As a further example, the deep induction rule for `PTree` is

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

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

Moreover, for types admitting both deep induction and structural induction, the former generalizes the latter. Indeed, structural induction rules can be derived from deep induction rules by always choosing the constantly true predicate as the custom predicate traversing each data element contained in the data type. That way, deep induction only inspects the structure of the data type and not its content, just like structural induction does. A concrete example of such derivation is demonstrated in Section 4.1.

3 Introducing GADTs

As noted in Subsection 2.1, the return types of the constructors of a nested type have to be the same type instance of the type being defined. As a further generalization of ADTs and nested types, *generalized algebraic data types* (GADTs) [3,14,12] relax the restriction on the type instances appearing in a data type definition 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. Applications of GADTs include generic programming, modeling programming languages via higher-order abstract syntax, maintaining invariants in data structures, and expressing constraints in embedded domain-specific languages. GADTs have also been used, e.g., to implement tagless interpreters [8,11,10], to improve memory performance [7], and to design APIs [9].

Example 3.1 As a first and notable example of GADT, we consider the `Equal` type. This GADT is parametrized by two type indices, but it is only possible to construct a data element if the two indices are instantiated at the same type. In Agda, we code it as

$$\begin{aligned} \text{data Equal} &: \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} \text{ where} \\ \text{refl} &: \text{Equal } A \ A \end{aligned} \tag{4}$$

`Equal` has thus a single data element when its two type arguments are the same and no data elements otherwise. The importance of the `Equal` type lies in the fact that we can understand other GADTs in terms of equality, as shown in the examples below.

Example 3.2 A more complex example of GADT is the type of sequences,

$$\begin{aligned} \text{data Seq} &: \text{Set} \rightarrow \text{Set} \text{ where} \\ \text{const} &: A \rightarrow \text{Seq } A \\ \text{sPair} &: \text{Seq } A \rightarrow \text{Seq } B \rightarrow \text{Seq } (A \times B) \end{aligned} \tag{5}$$

which comprises sequences of any type `a` and sequences obtained by pairing the data in two already existing sequences. Such GADTs can be understood in terms of the `Equal` data type [3,12]. For example, we can rewrite the `Seq` type as

$$\begin{aligned} \text{data Seq} &: \text{Set} \rightarrow \text{Set} \text{ where} \\ \text{const} &: A \rightarrow \text{Seq } A \\ \text{sPair} &: \forall (B \ C : \text{Set}) \rightarrow \text{Equal } A \ (B \times C) \rightarrow \text{Seq } B \rightarrow \text{Seq } C \rightarrow \text{Seq } A \end{aligned} \tag{6}$$

where 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 representing GADTs as Church encodings [13,1].

Example 3.3 GADTs are expressive enough to represent, for example, the types and terms of the simply typed lambda calculus. We will use a variant of the simply typed lambda calculus including a base type of booleans, and type constructors for arrow types and list types. We define a GADT `LType` to encode the set of permissible types for this calculus:

$$\begin{aligned} \text{data LType} &: \text{Set} \rightarrow \text{Set} \text{ where} \\ \text{tBool} &: \forall (B : \text{Set}) \rightarrow \text{Equal } A \ \text{Bool} \rightarrow \text{LType } A \\ \text{tArr} &: \forall (B \ C : \text{Set}) \rightarrow \text{Equal } A \ (B \rightarrow C) \rightarrow \text{LType } B \rightarrow \text{LType } C \rightarrow \text{LType } A \\ \text{tList} &: \forall (B : \text{Set}) \rightarrow \text{Equal } A \ (\text{List } B) \rightarrow \text{LType } B \rightarrow \text{LType } A \end{aligned} \tag{7}$$

The type system includes a `tBool` constructor representing the type of booleans; an arrow type constructor, `tArr`, which takes two types `B` and `C` and produces the arrow type `B → C`; and a `tList` constructor which takes a type `B` and constructs the type `List B`.

The term calculus for this type system includes variables, abstraction, application, and also includes an introduction rule that takes a list of lambda terms of type `B` and produces a lambda term of type `List B`.

$$\begin{aligned} \text{data LTerm} &: \text{Set} \rightarrow \text{Set} \text{ where} \\ \text{var} &: \text{String} \rightarrow \text{LType } A \rightarrow \text{LTerm } A \\ \text{abs} &: \forall (B \ C : \text{Set}) \rightarrow \text{Equal } A \ (B \rightarrow C) \rightarrow \text{String} \rightarrow \text{LType } B \rightarrow \text{LTerm } C \rightarrow \text{LTerm } A \\ \text{app} &: \forall (B : \text{Set}) \rightarrow \text{LTerm } (B \rightarrow A) \rightarrow \text{LTerm } B \rightarrow \text{LTerm } A \\ \text{listC} &: \forall (B : \text{Set}) \rightarrow \text{Equal } A \ (\text{List } B) \rightarrow \text{List } (\text{LTerm } B) \rightarrow \text{LTerm } A \end{aligned} \tag{8}$$

The type parameter for `LTerm` tracks the types of simply typed lambda calculus terms. So the type `LTerm A` contains lambda terms of type `A`. Variables are represented as strings and are tagged with their type. The `var`

constructor takes a variable name and a type for the variable and produces a term of the given type. The `abs` constructor takes a variable name, a type B for the variable, and a lambda term of type C for the body of the abstraction, and it produces a term of type $B \rightarrow C$. The `app` constructor applies a lambda term of type $B \rightarrow A$ to a lambda term of type B , producing a lambda term of type A . The `listC` constructor takes a list of lambda terms of type B and produces a lambda term of type $\text{List } B$.

Variables are tagged with their types in the `var` (variable introduction) and `abs` (abstraction) constructors. The role of `LType` in the `var` and `abs` constructors is to enforce that variables can only be tagged with a legal type, e.g., `Bool`, `Bool \rightarrow Bool`, `List (Bool \rightarrow Bool)`, etc. Therefore the presence of `LType` in these constructors ensures that all lambda terms produced by `var`, `abs`, `app`, and `listC` are well-typed.

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 2.2 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 [5]. 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 [?].

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 3.1. 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 5, 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 GA . the predicate-lifting function for `Equal` is the function

$$\text{Equal}^\wedge : \forall (A B : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow (B \rightarrow \text{Set}) \rightarrow \text{Equal } A B \rightarrow \text{Set}$$

defined as

$$\text{Equal}^\wedge A A Q Q' \text{Refl} = \forall (a : A) \rightarrow \text{Equal } (Q a) (Q' a)$$

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:

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

The 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

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

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

$$\text{dIndEqual } P \text{crefl } A B Q_A Q_B e \text{liftE} : P A B Q_A Q_B e$$

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

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

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 ,

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

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 indEqual as

$$\text{indEqual } Q \text{srefl } A B e = \text{dIndEqual } P \text{srefl } A B K_\top^A K_\top^B e \text{liftE}$$

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

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

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 3.2. Again, the first step in deriving the induction rule for **Seq** consists in defining the predicate-lifting function over it,

$$\text{Seq}^\wedge : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{Seq } A \rightarrow \text{Set}$$

which is given by pattern-matching as

$$\text{Seq}^\wedge A Q_A (\text{const } a) = Q_A a$$

where $Q_A : A \rightarrow \text{Set}$ and $a : A$, and

$$\begin{aligned} & \text{Seq}^\wedge A Q_A (\text{sPair } B C e s_B s_C) \\ &= \sum [Q_B : B \rightarrow \text{Set}] [Q_C : C \rightarrow \text{Set}] \text{Equal}^\wedge A (B \times C) Q_A (Q_B \times Q_C) e \times \text{Seq}^\wedge B Q_B s_B \times \text{Seq}^\wedge C Q_C s_C \end{aligned}$$

where $e : \text{Equal } A (B \times C)$, $s_B : \text{Seq } B$ and $s_C : \text{Seq } C$.

Finally, let dIndConst be the function

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

associated to the **const** constructor, and let dIndSPair be the function

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

associated to the **sPair** constructor.

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

$$\begin{aligned} & \forall (P : \forall (A : \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{Seq } A \rightarrow \text{Set}) \rightarrow C \text{Const } P \rightarrow C \text{sPair } P \\ & \rightarrow \forall (A : \text{Set}) (Q_A : A \rightarrow \text{Set}) (s_A : \text{Seq } A) \rightarrow \text{Seq}^\wedge A Q_A s_A \rightarrow P A Q_A s_A \end{aligned}$$

To validate the induction rule, we define a term dIndSeq for the above type. We have to define

$$\text{dIndSeq } P \text{ cconst cspair } A \ Q_A \ s_A \ \text{liftA} : P \ A \ Q_A \ s_A$$

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

$$\text{dIndSeq } P \text{ cconst cspair } A \ Q_A \ (\text{const } a) \ \text{liftA} = \text{cconst } A \ Q_A \ a \ \text{liftA}$$

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

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

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

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

and p_B and p_C are defined as follows:

$$\begin{aligned} p_B &= \text{DlSeq } P \text{ cconst cspair } B \ Q_B \ s_B \ \text{liftB} : P \ B \ Q_B \ s_B \\ p_C &= \text{DlSeq } P \text{ cconst cspair } C \ Q_C \ s_C \ \text{liftC} : P \ C \ Q_C \ s_C \end{aligned}$$

4.3 (Deep) induction for LTerm

Some notational conventions: I am using uppercase letters for elements of LType and lowercase letters for elements of LTerm . For example, $(T_b : \text{LType } b)$ or $(t_c : \text{LTerm } c)$.

Now we will define an induction rule for the LTerm type defined in Equation 8. To define this, 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

$$\text{Arr}^A : \forall (a \ b : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow (b \rightarrow \text{Set}) \rightarrow (a \rightarrow b) \rightarrow \text{Set}$$

is defined as:

$$\text{Arr}^A \ a \ b \ Q_a \ Q_b \ f = \forall (x : a) \rightarrow Q_a \ x \rightarrow Q_b \ (f \ x)$$

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

$$\text{LType}^A : \forall (a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow \text{LType } a \rightarrow \text{Set}$$

and is defined by pattern-matching as

$$\text{LType}^A \ a \ Q_a \ (\text{TBool } b \ e) = \exists (Q_b : b \rightarrow \text{Set}) \rightarrow \text{Equal}^A \ a \ b \ Q_a \ (\text{KBool}^A \ b \ Q_b) \ e$$

where $Q_a : a \rightarrow \text{Set}$ and $e : \text{Equal } a \ (\text{KBool } b)$, and

$$\begin{aligned} \text{LType}^A \ a \ Q_a \ (\text{TArr } b \ c \ e \ T_b \ T_c) \\ = \exists (Q_b : b \rightarrow \text{Set}) (Q_c : c \rightarrow \text{Set}) \rightarrow \text{Equal}^A \ a \ (b \rightarrow c) \ Q_a \ (\text{Arr}^A \ b \ c \ Q_b \ Q_c) \ e \\ \quad \times \text{LType}^A \ b \ Q_b \ T_b \times \text{LType}^A \ c \ Q_c \ T_c \end{aligned}$$

where $Q_a : a \rightarrow \text{Set}$, $e : \text{Equal } a \ (b \rightarrow c)$, $T_b : \text{LType } b$, and $T_c : \text{LType } c$, and

$$\begin{aligned} \text{LType}^A \ a \ Q_a \ (\text{TList } b \ e \ T_b) \\ = \exists (Q_b : b \rightarrow \text{Set}) \rightarrow \text{Equal}^A \ a \ (\text{List } b) \ Q_a \ (\text{List}^A \ b \ Q_b) \ e \times \text{LType}^A \ b \ Q_b \ T_b \end{aligned}$$

where $Q_a : a \rightarrow \text{Set}$, $e : \text{Equal } a \ (\text{List } b)$, and $T_b : \text{LType } b$.

Given the lifting for $LType$, we can define the lifting for $LTerm$. Again, the lifting

$$LTerm^\wedge : \forall(a : Set) \rightarrow (a \rightarrow Set) \rightarrow LTerm\ a \rightarrow Set$$

is defined by pattern-matching as

$$LTerm^\wedge\ a\ Q_a\ (Var\ s\ T_a) = LType^\wedge\ a\ Q_a\ T_a$$

where $s : String$ and $T_a : LType\ a$, and

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

where $e : Equal\ a\ (b \rightarrow c)$, $s : String$, $T_b : LType\ b$, and $t_c : LTerm\ c$, and

$$\begin{aligned} LTerm^\wedge\ a\ Q_a\ (App\ b\ t_{ba}\ t_b) \\ = \exists(Q_b : b \rightarrow Set) \rightarrow LTerm^\wedge\ (b \rightarrow a)\ (Arr^\wedge\ b\ a\ Q_b\ Q_a)\ t_{ba} \times LTerm^\wedge\ b\ Q_b\ t_b \end{aligned}$$

where $t_{ba} : LTerm\ (B \rightarrow A)$ and $t_b : LTerm\ B$, and

$$\begin{aligned} LTerm^\wedge\ a\ Q_a\ (ListC\ b\ e\ ts) \\ = \exists(Q_b : b \rightarrow Set) \rightarrow Equal^\wedge\ a\ (List\ b)\ Q_a\ (List^\wedge\ b\ Q_b)\ e \times List^\wedge\ (LTerm\ b)\ (LTerm^\wedge\ b\ Q_b)\ ts \end{aligned}$$

where $e : Equal\ a\ (List\ b)$ and $ts : List\ (LTerm\ b)$. Notice 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 $CVar$ be the type

$$\begin{aligned} \lambda(P : \forall(a : Set) \rightarrow (a \rightarrow Set) \rightarrow LTerm\ a \rightarrow Set) \\ \rightarrow \forall(a : Set)(Q_a : a \rightarrow Set)(s : String)(T_a : LType\ a) \rightarrow LType^\wedge\ a\ Q_a\ T_a \rightarrow P\ a\ Q_a\ (Var\ s\ T_a) \end{aligned}$$

associated to the Var constructor. Let $CAbs$ be the type

$$\begin{aligned} \lambda(P : \forall(a : Set) \rightarrow (a \rightarrow Set) \rightarrow LTerm\ a \rightarrow Set) \\ \rightarrow \forall(a\ b\ c : Set)(Q_a : a \rightarrow Set)(Q_b : b \rightarrow Set)(Q_c : c \rightarrow Set)(e : Equal\ a\ (b \rightarrow c))(s : String) \\ \rightarrow (T_b : LType\ b) \rightarrow (t_c : LTerm\ c) \rightarrow Equal^\wedge\ a\ (b \rightarrow c)\ Q_a\ (Arr^\wedge\ b\ c\ Q_b\ Q_c)\ e \\ \rightarrow LType^\wedge\ b\ Q_b\ T_b \rightarrow P\ c\ Q_c\ t_c \rightarrow P\ a\ Q_a\ (Abs\ b\ c\ e\ s\ T_b\ t_c) \end{aligned}$$

associated to the Abs constructor. Let $CApp$ be the type

$$\begin{aligned} \lambda(P : \forall(a : Set) \rightarrow (a \rightarrow Set) \rightarrow LTerm\ a \rightarrow Set) \\ \rightarrow \forall(a\ b : Set)(Q_a : a \rightarrow Set)(Q_b : b \rightarrow Set)(t_{ba} : LTerm\ (b \rightarrow a))(t_b : LTerm\ b) \\ \rightarrow P\ (b \rightarrow a)\ (Arr^\wedge\ b\ a\ Q_b\ Q_a)\ t_{ba} \rightarrow P\ b\ Q_b\ t_b \rightarrow P\ a\ Q_a\ (App\ b\ t_{ba}\ t_b) \end{aligned}$$

associated to the App constructor. Let $CListC$ be the type

$$\begin{aligned} \lambda(P : \forall(a : Set) \rightarrow (a \rightarrow Set) \rightarrow LTerm\ a \rightarrow Set) \\ \rightarrow \forall(a\ b : Set)(Q_a : a \rightarrow Set)(Q_b : b \rightarrow Set)(e : Equal\ a\ (List\ b))(ts : List\ (LTerm\ b)) \\ \rightarrow Equal^\wedge\ a\ (List\ b)\ Q_a\ (List^\wedge\ b\ Q_b)\ e \rightarrow List^\wedge\ (LTerm\ b)\ (P\ b\ Q_b)\ ts \\ \rightarrow P\ a\ Q_a\ (ListC\ b\ e\ ts) \end{aligned}$$

associated to the $ListC$ constructor. Then the type of the deep induction principle for $LTerm$ is,

$$\begin{aligned} \forall(P : \forall(a : Set) \rightarrow (a \rightarrow Set) \rightarrow LTerm\ a \rightarrow Set) \rightarrow CVar\ P \rightarrow CAbs\ P \rightarrow CApp\ P \rightarrow CListC\ P \\ \rightarrow \forall(a : Set)(Q_a : a \rightarrow Set)(t_a : LTerm\ a) \rightarrow LTerm^\wedge\ a\ Q_a\ t_a \rightarrow P\ a\ Q_a\ t_a \end{aligned}$$

To prove the induction principle, we define a term DILTerm for it. We have to define

$$\text{DILTerm } P \text{ cvar cabs capp clstc } a \ Q_a \ t_a \ L_a : P \ a \ Q_a \ t_a$$

As before, we prove the induction principle by pattern matching on t_a . For the Var case, let $t_a = (\text{Var } s \ T_a)$ and define

$$\text{DILTerm } P \text{ cvar cabs capp clstc } a \ Q_a \ (\text{Var } s \ T_a) \ L_a = \text{cvar } a \ Q_a \ s \ T_a \ L_a$$

Notice that $\text{LTerm}^a \ a \ Q_a \ (\text{Var } s \ T_a) = \text{LType}^a \ a \ Q_a \ T_a$, so $L_a : \text{LType}^a \ a \ Q_a \ T_a$. For the Abs case, let $t_a = (\text{Abs } b \ c \ e \ T_b \ t_c)$ and define

$$\begin{aligned} \text{DILTerm } P \text{ cvar cabs capp clstc } a \ Q_a \ (\text{Abs } b \ c \ e \ T_b \ t_c) \ (Q_b, Q_c, L_e, L_{T_b}, L_{t_c}) \\ = \text{cabs } a \ b \ c \ Q_a \ Q_b \ Q_c \ e \ T_b \ t_c \ L_e \ L_{T_b} \ p_c \end{aligned}$$

where $(Q_b, Q_c, L_e, L_{T_b}, L_{t_c}) : \text{LTerm}^a \ a \ Q_a \ (\text{Abs } b \ c \ e \ T_b \ t_c)$

- $Q_b : b \rightarrow \text{Set}$
- $Q_c : c \rightarrow \text{Set}$
- $L_e : \text{Equal}^a \ a \ (b \rightarrow c) \ Q_a \ (\text{Arr}^a \ b \ c \ Q_b \ Q_c) \ e$
- $L_{T_b} : \text{LType}^a \ b \ Q_b \ T_b$
- $L_{t_c} : \text{LTerm}^a \ c \ Q_c \ T_c$

and p_c is defined as:

$$p_c = \text{DILTerm } P \text{ cvar cabs capp clstc } c \ Q_c \ t_c \ L_{t_c} : P \ c \ Q_c \ t_c$$

For the App case, let $t_a = (\text{App } b \ t_{ba} \ t_b)$ and define

$$\begin{aligned} \text{DILTerm } P \text{ cvar cabs capp clstc } a \ Q_a \ (\text{App } b \ t_{ba} \ t_b) \ (Q_b, L_{t_{ba}}, L_{t_b}) \\ = \text{capp } a \ b \ Q_a \ Q_b \ t_{ba} \ t_b \ p_{ba} \ p_b \end{aligned}$$

where $(Q_b, L_{t_{ba}}, L_{t_b}) : \text{LTerm}^a \ a \ Q_a \ (\text{App } b \ t_{ba} \ t_b)$

- $Q_b : b \rightarrow \text{Set}$
- $L_{t_{ba}} : \text{LTerm}^a \ (b \rightarrow a) \ (\text{Arr}^a \ b \ a \ Q_b \ Q_a) \ t_{ba}$
- $L_{t_b} : \text{LTerm}^a \ b \ Q_b \ t_b$

and p_{ba} and p_b are defined as:

$$\begin{aligned} p_{ba} &= \text{DILTerm } P \text{ cvar cabs capp clstc } (b \rightarrow a) \ (\text{Arr}^a \ b \ a \ Q_b \ Q_a) \ t_{ba} \ L_{t_{ba}} : P \ (b \rightarrow a) \ (\text{Arr}^a \ b \ a \ Q_b \ Q_a) \ t_{ba} \\ p_b &= \text{DILTerm } P \text{ cvar cabs capp clstc } b \ Q_b \ t_b \ L_{t_b} : P \ b \ Q_b \ t_b \end{aligned}$$

For the ListC case, let $t_a = (\text{ListC } b \ e \ ts)$ and define

$$\text{DILTerm } P \text{ cvar cabs capp clstc } a \ Q_a \ (\text{ListC } b \ e \ ts) \ (Q_b, L_e, L_{\text{List}}) = \text{clstc } a \ b \ Q_a \ Q_b \ e \ ts \ L_e \ p_{\text{List}}$$

where $(Q_b, L_e, L_{\text{List}}) : \text{LTerm}^a \ a \ Q_a \ (\text{ListC } b \ e \ ts)$

- $Q_b : b \rightarrow \text{Set}$
- $L_e : \text{Equal}^a \ a \ (\text{List } b) \ Q_a \ (\text{List}^a \ b \ Q_b) \ e$
- $L_{\text{List}} : \text{List}^a \ (\text{LTerm } b) \ (\text{LTerm}^a \ b \ Q_b) \ ts$

and p_{List} is defined as:

$$p_{\text{List}} = \text{List}^a \ \text{map} \ (\text{LTerm } b) \ (\text{LTerm}^a \ b \ Q_b) \ (P \ b \ Q_b) \ p_{ts} \ ts \ L_{\text{List}} : \text{List}^a \ (\text{LTerm } b) \ (P \ b \ Q_b) \ ts$$

and p_{ts} is defined as:

$$p_{ts} = \text{DILTerm } P \text{ cvar cabs capp clstc } b \ Q_b : \text{PredMap} \ (\text{LTerm}^a \ b \ Q_b) \ (P \ b \ Q_b)$$

where

$$\text{List}^\wedge \text{map} : \forall (a : \text{Set}) \rightarrow (Q_a Q'_a : a \rightarrow \text{Set}) \rightarrow \text{PredMap } Q_a Q'_a \rightarrow \text{PredMap } (\text{List}^\wedge a Q_a) (\text{List}^\wedge a Q'_a)$$

takes a morphism of predicates and produces a morphism of lifted predicates. PredMap gives the type of morphisms of predicates and is defined as:

$$\begin{aligned} \text{PredMap} &: \forall \{a : \text{Set}\} \rightarrow (Q_a Q'_a : a \rightarrow \text{Set}) \rightarrow \text{Set} \\ \text{PredMap } Q_a Q'_a &= \forall (x : A) \rightarrow Q_a x \rightarrow Q'_a x \end{aligned}$$

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

$$\begin{aligned} \text{data } G(\overline{a : \text{Set}}) : \text{Set} \text{ where} \\ C_i : F_i G \overline{b} \rightarrow G(\overline{K_i b}) \end{aligned}$$

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

- $F_i = F'_i \times F''_i$ where F'_i and F''_i have the same signature as F_i and are built recursively from the same induction rules.
- $F_i = F'_i + F''_i$ where F'_i and F''_i have the same signature as F_i and are built recursively from the same induction rules.
- $F_i = F'_i \rightarrow F''_i$ where F'_i does not contain the recursive variable, i.e., $F'_i : \text{Set}^\beta \rightarrow \text{Set}$ is a type constructor of arity β , and F''_i has the same signature as F_i and is built recursively from the same induction rules.
- $F_i G \overline{b} = G(\overline{F_a b})$ where none of the F_a contains the recursive variable, i.e., $F_a : \text{Set}^\beta \rightarrow \text{Set}$ is a type constructor of arity β for each a . Such restriction is necessary to prevent nesting, as that would break the induction rule as discussed in Section 5.
- $F_i G \overline{b} = H \overline{b}$ where H is a type constructor of arity β not containing the recursive variable, i.e., $H : \text{Set}^\beta \rightarrow \text{Set}$. Notice that this covers the case in which F_i is a closed type, so, in particular, the unit and empty types, 1 and 0.
- $F_i G \overline{b} = H(\overline{F_c G b})$ where H is a γ -ary type constructor not containing the recursive variable, i.e., $H : \text{Set}^\gamma \rightarrow \text{Set}$, and F_c has the same signature as F_i and is built recursively from the same induction rules, for every $c = 1 \dots \gamma$. 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 H admits functorial semantics [6], and thus we have a map function for H^\wedge ,

$$\text{HLMMap} : \forall (\overline{c : \text{Set}}) (\overline{Q_c Q'_c : c \rightarrow \text{Set}}) \rightarrow \overline{\text{PredMap } c Q_c Q'_c} \rightarrow \text{PredMap } (H \overline{c}) (H^\wedge \overline{c} \overline{Q_c}) (H^\wedge \overline{c} \overline{Q'_c})$$

where $\text{PredMap} : \forall (c : \text{Set}) \rightarrow (c \rightarrow \text{Set}) \rightarrow (c \rightarrow \text{Set}) \rightarrow \text{Set}$ is defined as

$$\text{PredMap } c Q_c Q'_c = \forall (x : c) \rightarrow Q_c x \rightarrow Q'_c x$$

and represents the type of morphisms between predicates. A concrete way to define HLMMap 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):

$$F_i G \overline{b} := F'_i G \overline{b} \times F''_i G \overline{b} \mid F'_i G \overline{b} + F''_i G \overline{b} \mid F'_i \overline{b} \rightarrow F''_i G \overline{b} \mid G(\overline{F_a b}) \mid H \overline{b} \mid H(\overline{F_c G b})$$

A further requirement that applies to all of the types appearing above, including the types 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,

$$\begin{aligned} \text{data } G(a : \text{Set}) : \text{Set where} \\ C : F G \bar{b} \rightarrow G(K \bar{b}) \end{aligned} \tag{9}$$

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 $(\text{Set} \rightarrow \text{Set}) \rightarrow \text{Set}^\beta \rightarrow \text{Set}$ and each K has signature $\text{Set}^\beta \rightarrow \text{Set}$. The constructor C can be rewritten using the Equal type as

$$C : \exists(\bar{b} : \text{Set}) \rightarrow \text{Equal } a (K \bar{b}) \rightarrow F G \bar{b} \rightarrow G a$$

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

$$G^\wedge : \forall(a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow G a \rightarrow \text{Set}$$

as

$$G^\wedge a Q_a (C \bar{b} e x) = \exists(\bar{Q}_b : b \rightarrow \text{Set}) \rightarrow \text{Equal}^\wedge a (K \bar{b}) Q_a (K^\wedge \bar{b} \bar{Q}_b) e \times F^\wedge G \bar{b} G^\wedge \bar{Q}_b x$$

where $Q_a : a \rightarrow \text{Set}$ and $C \bar{b} e x : G a$, i.e., $e : \text{Equal } a (K \bar{b})$ and $x : F G \bar{b}$. As already mentioned before, we also assume to have liftings for F ,

$$\begin{aligned} F^\wedge : \forall(G : \text{Set}^\alpha \rightarrow \text{Set})(\bar{b} : \text{Set}) \rightarrow (\forall(a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow G a \rightarrow \text{Set}) \\ \rightarrow (\bar{b} \rightarrow \text{Set}) \rightarrow F G \bar{b} \rightarrow \text{Set} \end{aligned}$$

and for K ,

$$K^\wedge : \forall(\bar{b} : \text{Set}) \rightarrow (\bar{b} \rightarrow \text{Set}) \rightarrow K \bar{b} \rightarrow \text{Set}$$

Finally, associate the function

$$\begin{aligned} CC = \lambda(P : \forall(a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow G a \rightarrow \text{Set}) \\ \rightarrow \forall(a : \text{Set})(\bar{b} : \text{Set})(Q_a : a \rightarrow \text{Set})(\bar{Q}_b : b \rightarrow \text{Set})(e : \text{Equal } a (K \bar{b})) (x : F G \bar{b}) \\ \rightarrow \text{Equal}^\wedge a (K \bar{b}) Q_a (K^\wedge \bar{b} \bar{Q}_b) e \rightarrow F^\wedge G \bar{b} P \bar{Q}_b x \rightarrow P a Q_a (C \bar{b} e x) \end{aligned}$$

to the constructor C .

The induction rule for G is

$$\begin{aligned} \forall(P : \forall(a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow G a \rightarrow \text{Set}) \rightarrow CC P \\ \rightarrow \forall(a : \text{Set})(Q_a : a \rightarrow \text{Set})(y : G a) \rightarrow G^\wedge a Q_a y \rightarrow P a Q_a y \end{aligned}$$

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

$$\text{DIG } P cc a Q_a (C \bar{b} e x) (\bar{Q}_b, L_E, L_F) = cc a \bar{b} Q_a \bar{Q}_b e \times L_E (p \times L_F)$$

where $cc : CC P$ and

- $C \bar{b} e x : G a$, i.e., $e : \text{Equal } a (K \bar{b})$, and $x : F G \bar{b}$;
- $(\bar{Q}_b, L_E, L_F) : G^\wedge a Q_a (C \bar{b} e x)$, i.e., $\bar{Q}_b : b \rightarrow \text{Set}$ for each b , $L_E : \text{Equal}^\wedge a (K \bar{b}) Q_a (K^\wedge \bar{b} \bar{Q}_b) e$, and $L_F : F^\wedge G \bar{b} G^\wedge \bar{Q}_b x$.

Finally, the morphism of predicates

$$p : \text{PredMap } (F \bar{G} \bar{b}) (F^\wedge \bar{G} \bar{b} G^\wedge \bar{Q}_b) (F^\wedge \bar{G} \bar{b} P \bar{Q}_b)$$

is defined by structural induction on F as follows:

- Case $F = F_1 \times F_2$ where F_1 and F_2 have the same signature as F . We have that

$$F^\wedge \bar{G} \bar{b} P \bar{Q}_b = \text{Pair}^\wedge (F_1 \bar{G} \bar{b}) (F_2 \bar{G} \bar{b}) (F_1^\wedge \bar{G} \bar{b} P \bar{Q}_b) (F_2^\wedge \bar{G} \bar{b} P \bar{Q}_b)$$

By inductive hypothesis, there exist morphisms of predicates

$$\begin{aligned} p_1 &: \text{PredMap } (F_1 \bar{G} \bar{b}) ((F_1)^\wedge \bar{G} \bar{b} G^\wedge \bar{Q}_b) ((F_1)^\wedge \bar{G} \bar{b} P \bar{Q}_b) \\ p_2 &: \text{PredMap } (F_2 \bar{G} \bar{b}) ((F_2)^\wedge \bar{G} \bar{b} G^\wedge \bar{Q}_b) ((F_2)^\wedge \bar{G} \bar{b} P \bar{Q}_b) \end{aligned}$$

Thus, we define $p(x_1, x_2)(L_1, L_2) = (p_1 x_1 L_1, p_2 x_2 L_2)$ for $x_1 : F_1 \bar{G} \bar{b}$, $L_1 : F_1^\wedge \bar{G} \bar{b} G^\wedge \bar{Q}_b$, $x_2 : F_2 \bar{G} \bar{b}$ and $L_2 : F_2^\wedge \bar{G} \bar{b} G^\wedge \bar{Q}_b$.

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

$$F^\wedge \bar{G} \bar{b} P \bar{Q}_b \times = \forall (z : F_1 \bar{b}) \rightarrow F_1^\wedge \bar{b} \bar{Q}_b z \rightarrow F_2^\wedge \bar{G} \bar{b} P \bar{Q}_b (xz)$$

where $x : F \bar{G} \bar{b} = F_1 \bar{b} \rightarrow F_2 \bar{G} \bar{b}$. By inductive hypothesis, there exist a morphism of predicates

$$p_2 : \text{PredMap } (F_2 \bar{G} \bar{b}) (F_2^\wedge \bar{G} \bar{b} G^\wedge \bar{Q}_b) (F_2^\wedge \bar{G} \bar{b} P \bar{Q}_b)$$

Thus, we define $p \times L_F : F^\wedge \bar{G} \bar{b} P \bar{Q}_b \times$ for $L_F : F^\wedge \bar{G} \bar{b} G^\wedge \bar{Q}_b \times$ as $p \times L_F z L_1 = p_2(xz)(L_F z L_1)$ for $z : F_1 \bar{b}$ and $L_1 : F_1^\wedge \bar{b} \bar{Q}_b z$. Notice that F_1 not containing the recursive variable is a necessary restriction, as the proof relies on $F^\wedge \bar{G} \bar{b} G^\wedge \bar{Q}_b \times$ and $F^\wedge \bar{G} \bar{b} P \bar{Q}_b \times$ having the same domain $F_1^\wedge \bar{b} \bar{Q}_b z$.

- Case $F \bar{G} \bar{b} = G(F' \bar{b})$ where F' does not contain the recursive variable, i.e., $F' : \text{Set}^\beta \rightarrow \text{Set}$. Thus, $F^\wedge \bar{G} \bar{b} P \bar{Q}_b = P(F' \bar{b})(F'^\wedge \bar{b} \bar{Q}_b)$. So, p is defined as

$$p = \text{DIG } P \text{ cc } (F' \bar{b}) (F'^\wedge \bar{b} \bar{Q}_b)$$

- Case $F \bar{G} \bar{b} = H \bar{b}$ where H is a β -ary type constructor not containing the recursive variable, i.e., $H : \text{Set}^\beta \rightarrow \text{Set}$. In such case, $p : \text{PredMap } (H \bar{b}) (H^\wedge \bar{b} \bar{Q}_b) (H^\wedge \bar{b} \bar{Q}_b)$ is just the identity morphism of predicates.
- Case $F \bar{G} \bar{b} = H(\bar{F}_c \bar{G} \bar{b})$ where H is a γ -ary type constructor not containing the recursive variable, i.e., $H : \text{Set}^\gamma \rightarrow \text{Set}$, and F_c has the same signature as F , for every $c = 1 \dots \gamma$. Moreover, we assume that H has an associated predicate-lifting function,

$$H^\wedge : \forall (\bar{c} : \text{Set}) \rightarrow (\bar{c} \rightarrow \text{Set}) \rightarrow H \bar{c} \rightarrow \text{Set}$$

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

$$\forall (\bar{c} : \text{Set}) (\bar{Q}_c \bar{Q}'_c : \bar{c} \rightarrow \text{Set}) \rightarrow \text{PredMap } \bar{c} \bar{Q}_c \bar{Q}'_c \rightarrow \text{PredMap } (H \bar{c}) (H^\wedge \bar{c} \bar{Q}_c) (H^\wedge \bar{c} \bar{Q}'_c)$$

That means that H cannot be a GADT, as GADTs have no functorial semantics [?] 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 [5,4]. Thus,

$$F^\wedge \bar{G} \bar{b} P \bar{Q}_b = H^\wedge (\bar{F}_c \bar{G} \bar{b}) (\bar{F}_c^\wedge \bar{G} \bar{b} P \bar{Q}_b)$$

By induction hypothesis, there is a morphism of predicates

$$p_c : \text{PredMap } (F_c \bar{G} \bar{b}) (F_c^\wedge \bar{G} \bar{b} G^\wedge \bar{Q}_b) (F_c^\wedge \bar{G} \bar{b} P \bar{Q}_b)$$

for every $c = 1 \dots \gamma$. So, p is defined as

$$p = \text{HLMMap}(\overline{F_c G b})(\overline{F_c^\wedge G b G^\wedge Q_b})(\overline{F_c^\wedge G b P Q_b}) \overline{p_c}$$

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 3) 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 [6], but GADTs cannot admit both functorial and parametric semantics at the same time [?]. In this section we show how induction for GADTs featuring nesting goes wrong by analyzing the following concrete example of such a type.

$$\begin{aligned} \text{data } G(a : \text{Set}) : \text{Set where} \\ C : G(Ga) \rightarrow G(a \times a) \end{aligned} \tag{10}$$

The constructor C can be rewritten as

$$C : \exists(b : \text{Set}) \rightarrow \text{Equal } a(b \times b) \rightarrow G(Gb) \rightarrow Ga$$

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

$$G^\wedge : \forall(a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow G \bar{a} \rightarrow \text{Set}$$

is defined as

$$G^\wedge a Q_a (C b e x) = \exists(Q_b : b \rightarrow \text{Set}) \rightarrow \text{Equal}^\wedge a(b \times b) Q_a (\text{Pair}^\wedge b b Q_b Q_b) e \times G^\wedge (Gb) (G^\wedge b Q_b) x$$

where $Q_a : a \rightarrow \text{Set}$, $e : \text{Equal } a(b \times b)$ and $x : G(Gb)$. Finally, let CC be the function

$$\begin{aligned} \lambda(P : \forall(a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow Ga \rightarrow \text{Set}) \\ \rightarrow \forall(a b : \text{Set})(Q_a : a \rightarrow \text{Set})(Q_b : b \rightarrow \text{Set})(e : \text{Equal } a(b \times b))(x : G(Gb)) \\ \rightarrow \text{Equal}^\wedge a(b \times b) Q_a (\text{Pair}^\wedge b b Q_b Q_b) e \rightarrow P(Gb) (P b Q_b) x \rightarrow P a Q_a (C b e x) \end{aligned}$$

associated to the C constructor.

The induction rule for G is

$$\begin{aligned} \forall(P : \forall(a : \text{Set}) \rightarrow (a \rightarrow \text{Set}) \rightarrow Ga \rightarrow \text{Set}) \rightarrow CC P \\ \rightarrow \forall(a : \text{Set})(Q_a : a \rightarrow \text{Set})(y : Ga) \rightarrow G^\wedge a Q_a y \rightarrow P a Q_a y \end{aligned}$$

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

$$DIG P cc a Q_a (C b e x) (Q_b, L_E, L_G) = cc a b Q_a Q_b e \times L_E p$$

where $cc : CC P$ and

- $C b e x : Ga$, i.e., $e : \text{Equal } a(b \times b)$ and $x : G(Gb)$;
- $(Q_b, L_E, L_G) : G^\wedge a Q_a (C b e x)$, i.e., $Q_b : b \rightarrow \text{Set}$, $L_E : \text{Equal}^\wedge a(b \times b) Q_a (\text{Pair}^\wedge b b Q_b Q_b) e$, and $L_G : G^\wedge (Gb) (G^\wedge b Q_b) x$.

We still need to define $p : P(Gb) (P b Q_b) x$. We do so by using the induction rule and letting

$$p = DIG P cc (Gb) (P b Q_b) \times q$$

where we still need to provide $q : G^\wedge (Gb) (P b Q_b) x$. If we had the map function of G^\wedge ,

$$GLMap : \forall(a : \text{Set})(Q_a Q'_a : a \rightarrow \text{Set}) \rightarrow \text{PredMap } a Q_a Q'_a \rightarrow \text{PredMap } (Ga) (G^\wedge a Q_a) (G^\wedge a Q'_a)$$

then we would be able to define

$$q = \text{GLMap } (G \ b) \ (G^\wedge \ b \ Q_b) \ (P \ b \ Q_b) \ (\text{DIG } P \ \text{cc } b \ Q_b) \times L_G$$

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

$$\text{GLMap } a \ Q_a \ Q'_a \ M \ (C \ b \ e \ x) \ (Q_b, L_E, L_G) = (Q'_b, L'_E, L'_G)$$

where $Q_a : a \rightarrow \text{Set}$, $Q'_a : a \rightarrow \text{Set}$, $M : \text{PredMap } a \ Q_a \ Q'_a$, $C \ b \ e \ x : G \ a$, i.e.,

- $e : \text{Equal } a \ (b \times b)$;
- $x : G \ (G \ b)$;
- $(Q_b, L_E, L_G) : G^\wedge \ a \ Q_a \ (C \ b \ e \ x)$, i.e.,
- $Q_b : b \rightarrow \text{Set}$;
- $L_E : \text{Equal}^\wedge \ a \ (b \times b) \ Q_a \ (\text{Pair}^\wedge \ b \ b \ Q_b \ Q_b) \ e$;
- $L_G : G^\wedge \ (G \ b) \ (G^\wedge \ b \ Q_b) \ x$;

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

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

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

At a deeper level, the fundamental issue is that the `Equal` type does not have functorial semantics, so that having morphisms $A \rightarrow A'$ and $B \rightarrow B'$ and a proof that A is equal to A' does not provide a proof that B is equal to 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 nested types, indeed, GADTs do not admit a semantic interpretation that is both parametric and functorial [?].

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

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

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 `LTerm 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:

$$\text{getTypeProof} : \forall (a : \text{Set}) \rightarrow (t : \text{LTerm } a) \rightarrow \text{getType } a \ t$$

which we prove by

$$\text{getTypeProof } a \ t = \text{DILTerm } (\lambda \ b \ Q_b \ t \rightarrow \text{getType } b \ t) \ \text{gtVar} \ \text{gtABs} \ \text{gtApp} \ \text{gtListC} \ a \ K1 \ t \ (\text{LTerm}^\wedge \ K1 \ a \ t)$$

where $K1 : a \rightarrow \text{Set}$ is the constantly true predicate:

$$K1 x = \top$$

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

$$\text{LTerm}^\wedge K1 : \forall (a : \text{Set}) (t : \text{LTerm} A) \rightarrow \text{LTerm}^\wedge a K1 t$$

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

$$\text{gtVar} : \forall (a : \text{Set}) (Q_a : a \rightarrow \text{Set}) (s : \text{String}) (T_a : \text{LType} a) \rightarrow \text{LType}^\wedge a Q_a T_a \rightarrow \text{Maybe} (\text{LType} a)$$

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

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

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

For variables we simply return the type 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 $(\text{List}^\wedge (\text{LTerm} b) (\text{getType} b) ts)$ 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

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

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

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

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

6.1 Defining $\text{LTerm}^\wedge K1$

The last piece of infrastructure we need to define `getTypeProof` is a function

$$\text{LTerm}^\wedge K1 : \forall (a : \text{Set}) \rightarrow (t : \text{LTerm} A) \rightarrow \text{LTerm}^\wedge a K1 t$$

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

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

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

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

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

so to define the **Abs** case of $\text{LTerm}^\wedge K1$, we need a proof of

$$\text{Equal}^\wedge a (b \rightarrow c) K1 (\text{Arr}^\wedge b \text{ c } Q_b \text{ } Q_c) e$$

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

$$\text{Equal}^\wedge a (b \rightarrow c) K1 (\text{Arr}^\wedge b \text{ c } K1 \text{ } K1) e$$

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

$$\text{Equal}^\wedge \text{Arr} K1 : \text{Equal}^\wedge a (b \rightarrow c) K1 (\text{Arr}^\wedge b \text{ c } K1 \text{ } K1) e$$

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

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

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

$$\text{LTerm}^\wedge (b \rightarrow a) (\text{Arr}^\wedge b \text{ a } K1 \text{ } K1) t_{ba} \times \text{LTerm}^\wedge b \text{ } K1 \text{ } t_b$$

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

$$\text{LTerm}^\wedge (b \rightarrow a) K1 t_{ba}$$

and a map-like function

$$\begin{aligned} \text{LTerm}^\wedge \text{EqualMap} : \forall \{a : \text{Set}\} \rightarrow (Q_a \text{ } Q'_a : a \rightarrow \text{Set}) \rightarrow (\text{Equal}^\wedge a \text{ } a \text{ } Q_a \text{ } Q'_a \text{ } \text{rfl}) \\ \rightarrow \text{PredMap} (\text{LTerm}^\wedge a \text{ } Q_a) (\text{LTerm}^\wedge a \text{ } Q'_a) \end{aligned}$$

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

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

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

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

$$L_{\text{Arr}^\wedge K1} = \text{LTerm}^\wedge \text{EqualMap } K1 (\text{Arr}^\wedge b \text{ a } K1 \text{ } K1) \text{Equal}^\wedge \text{Arr} K1 t_{ba} L_{K1}$$

where $L_{K1} = \text{LTerm}^\wedge K1 (b \rightarrow a) t_{ba} : \text{LTerm}^\wedge (b \rightarrow a) K1 t_{ba}$.

Finally, we define the **ListC** case for $\text{LTerm}^\wedge K1$. Let $t = (\text{ListC } b \text{ e } t_s)$ and recall the definition of $\text{LTerm}^\wedge a (\text{ListC } b \text{ e } t_s)$ with all of the predicates instantiated to $K1$:

$$\text{Equal}^\wedge a (\text{List } b) K1 (\text{List}^\wedge b \text{ } K1) e \times \text{List}^\wedge (\text{LTerm}^\wedge b) (\text{LTerm}^\wedge b \text{ } K1) t_s$$

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

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

using

$$\text{List}^\wedge \text{map} : \forall (a : \text{Set}) \rightarrow (Q_a Q'_a : a \rightarrow \text{Set}) \rightarrow \text{PredMap } Q_a Q'_a \rightarrow \text{PredMap } (\text{List}^\wedge a Q_a) (\text{List}^\wedge a Q'_a)$$

to map a morphism of predicates

$$\text{PredMap } (K1) (\text{LTerm}^\wedge b K1)$$

to a morphism of lifted predicates

$$\text{PredMap } (\text{List}^\wedge (\text{LTerm } b) K1) (\text{List}^\wedge (\text{LTerm } b) (\text{LTerm}^\wedge b K1))$$

We define the ListC case of $\text{LTerm}^\wedge K1$ as

$$\text{LTerm}^\wedge K1 a (\text{ListC } b \text{ ts}) = (K1, \text{Equal}^\wedge \text{ListK1}, \text{L}_{\text{List}^\wedge \text{LTerm}^\wedge K1})$$

where $\text{L}_{\text{List}^\wedge \text{LTerm}^\wedge K1} : \text{List}^\wedge (\text{LTerm } b) (\text{LTerm}^\wedge b K1) \text{ ts}$

$$\text{L}_{\text{List}^\wedge \text{LTerm}^\wedge K1} = \text{List}^\wedge \text{map } (\text{LTerm } b) K1 (\text{LTerm}^\wedge b K1) m_{K1} \text{ ts } (\text{List}^\wedge K1 (\text{LTerm } b) \text{ ts})$$

and $m_{K1} : \text{PredMap } (K1) (\text{LTerm}^\wedge b K1)$

$$m_{K1} t * = \text{LTerm}^\wedge K1 b t$$

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

6.2 Liftings of $K1$

To provide a proof of $G^\wedge a K1 t$ for every term $t : G a$, 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 $\text{Pair}^\wedge a b K1 K1$ is equal to the predicate $K1$ on pairs. Given a pair $(x, y) : a \times b$, we have

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

while

$$K1 (x, y) = \top$$

and while these types are not equal they are clearly isomorphic. So for simplicity of presentation, we assume $(F^\wedge a K1)$ 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 [5] 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*, in: *Computer Science Logic*, 2012, pp. 46–61.
- [2] Bird, R. and L. Meertens, *Nested datatypes*, in: *Mathematics of Program Construction*, 1998.
- [3] Cheney, J. and R. Hinze, *First-class phantom types*, Technical report, Cornell University (2003).
- [4] Johann, P., E. Ghiorzi and D. Jeffries, *Parametricity for primitive nested types*, in: *Foundations of Software Science and Computation Structures*, 2021, pp. 324–343.
- [5] Johann, P. and A. Polonsky, *Higher-kinded data types: Syntax and semantics*, in: *Logic in Computer Science*, 2019, pp. 1–13.
- [6] Johann, P. and A. Polonsky, *Deep induction: Induction rules for (truly) nested types*, in: *Foundations of Software Science and Computation Structures*, 2020, pp. 339–358.
- [7] Minsky, Y., *Why GADTs matter for performance*, <https://blog.janestreet.com/why-gadts-matter-for-performance/> (2015).
- [8] Pasalic, E. and N. Linger, *Meta-programming with typed object-language representations*, in: *Generic Programming and Component Engineering*, 2004, pp. 136–167.
- [9] Penner, C., *Simpler and safer API design using GADTs*, <https://chrispenner.ca/posts/gadt-design> (2020).
- [10] Peyton Jones, S. L., D. Vytiniotis, G. Washburn and S. Weirich, *Simple unification-based type inference for gadts*, in: *International Conference on Functional Programming*, 2006, pp. 50–61.
- [11] Pottier, F. and Y. Régis-Gianas, *Stratified type inference for generalized algebraic data types*, in: *Principles of Programming Languages*, 2006, pp. 232–244.
- [12] Sheard, T. and E. Pasalic, *Meta-programming with built-in type equality*, in: *Fourth International Workshop on Logical Frameworks and Meta-Languages*, 2004.
- [13] Vytiniotis, D. and S. Weirich, *Parametricity, type equality, and higher-order polymorphism* (2010).
- [14] Xi, H., C. Chen and G. Chen, *Guarded recursive datatype constructors*, in: *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2003, pp. 224–235.