(Deep) Induction for GADTs

ANONYMOUS AUTHOR(S)

Abstract

1 INTRODUCTION

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) (Ask Daniel which flavor of syntax, paper as literate Agda, naming conventions?)

$$\begin{array}{l} \mathsf{data\ List\ (a : Set) : Set\ where} \\ \qquad \mathsf{Nil} \quad : \mathsf{List\ a} \\ \qquad \mathsf{Cons} : \mathsf{a} \to \mathsf{List\ a} \to \mathsf{List\ a} \end{array} \tag{1}$$

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. A simple example of such a type is given by Bird and Meertens' [Bird and Meertens 1998] prototypical *nested type*

$$\begin{array}{l} \mathsf{data\ PTree\ (a : Set) : Set\ where} \\ \qquad \mathsf{PLeaf} \quad : \mathsf{a} \to \mathsf{PTree\ a} \\ \qquad \mathsf{PNode} : \mathsf{PTree\ (a \times a)} \to \mathsf{PTree\ a} \end{array} \tag{2}$$

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 $(\mathsf{a} \times \mathsf{a})$ 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

$$\begin{array}{l} \mathsf{data\ Bush\ (a : Set) : Set\ where} \\ \qquad \mathsf{BNil} \quad : \mathsf{Bush\ a} \\ \qquad \mathsf{BCons} : \mathsf{a} \to \mathsf{Bush\ (Bush\ a)} \to \mathsf{Bush\ a} \end{array} \tag{3}$$

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

2021. 2475-1421/2021/8-ART1 $15.00
https://doi.org/

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 proving 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 is a function $\mathsf{a} \rightarrow \mathsf{Set}$ (where Set is the type of sets) mapping each $\mathsf{x} : \mathsf{a}$ to the set of proofs that the predicate holds for x. For example, the induction principle for List is

$$\forall (\mathsf{a} : \mathsf{Set})(\mathsf{P} : \mathsf{List\,a} \rightarrow \mathsf{Set}) \rightarrow \mathsf{P\,Nil} \rightarrow \big(\forall (\mathsf{x} : \mathsf{a})(\mathsf{xs} : \mathsf{List\,a}) \rightarrow \mathsf{P\,xs} \rightarrow \mathsf{P\,(Cons\,x\,xs)}\big) \\ \rightarrow \forall (\mathsf{xs} : \mathsf{List\,a}) \rightarrow \mathsf{P\,xs}$$

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 necessary involve a polymorphic predicate. Thus, the induction rule for PTree is

$$\forall (\mathsf{P} : \forall (\mathsf{a} : \mathsf{Set}) \rightarrow \mathsf{PTree\,a} \rightarrow \mathsf{Set}) \rightarrow \big(\forall (\mathsf{a} : \mathsf{Set})(\mathsf{x} : \mathsf{a}) \rightarrow \mathsf{P\,a\,(PLeaf\,x)}\big) \\ \rightarrow \big(\forall (\mathsf{a} : \mathsf{Set})(\mathsf{x} : \mathsf{PTree\,(a \times a)}) \rightarrow \mathsf{P\,(a \times a)\,x} \rightarrow \mathsf{P\,a\,(PNode\,x)}\big) \\ \rightarrow \forall (\mathsf{a} : \mathsf{Set})(\mathsf{x} : \mathsf{PTree\,a}) \rightarrow \mathsf{P\,a\,x}$$

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 [Johann and Polonsky 2020]. 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

$$\forall (\mathsf{P} : \forall (\mathsf{a} : \mathsf{Set}) \rightarrow (\mathsf{a} \rightarrow \mathsf{Set}) \rightarrow \mathsf{Bush\,a} \rightarrow \mathsf{Set}) \rightarrow \big(\forall (\mathsf{a} : \mathsf{Set}) \rightarrow \mathsf{P\,a\,BNil}\big) \\ \rightarrow \big(\forall (\mathsf{a} : \mathsf{Set})(\mathsf{Q} : \mathsf{a} \rightarrow \mathsf{Set})(\mathsf{x} : \mathsf{a})(\mathsf{y} : \mathsf{Bush\,(Bush\,a)}) \\ \rightarrow \mathsf{Q\,x} \rightarrow \mathsf{P\,(Bush\,a)\,(Bush^{\wedge}\,a\,Q)\,y} \rightarrow \mathsf{P\,a\,Q\,(BCons\,x\,y)}\big) \\ \rightarrow \forall (\mathsf{a} : \mathsf{Set})(\mathsf{Q} : \mathsf{a} \rightarrow \mathsf{Set})(\mathsf{x} : \mathsf{Bush\,a}) \rightarrow \mathsf{Bush^{\wedge}\,a\,Q\,x} \rightarrow \mathsf{P\,a\,Q\,x}$$

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

$$\begin{aligned}\mathsf{Bush}^\wedge\, a\, Q\, \mathsf{BNil} &= 1\\ \mathsf{Bush}^\wedge\, a\, Q\, (\mathsf{BCons}\, x\, y) &= Q\, x \times \mathsf{Bush}^\wedge\, (\mathsf{Bush}\, a)\, (\mathsf{Bush}^\wedge\, a\, Q)\, y\end{aligned}$$

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

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

where $\mathsf{List}^\wedge : \forall (a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{List}\, a \to \mathsf{Set}$ lifts a predicate Q on data of type a to a predicate on data of type $\mathsf{List}\, a$ asserting that Q holds for every element of its argument list. Finally, the deep induction rule for PTree is

$$\begin{aligned}&\forall (P : \forall (a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{PTree}\, a \to \mathsf{Set})\\ &\qquad \to \big(\forall (a : \mathsf{Set})(Q : a \to \mathsf{Set})(x : a) \to Q\, x \to P\, a\, Q\, (\mathsf{PLeaf}\, x)\big)\\ &\to \big(\forall (a : \mathsf{Set})(Q : a \to \mathsf{Set})(x : \mathsf{PTree}\, (a \times a)) \to P\, (a \times a)\, (\mathsf{Pair}^\wedge\, a\, Q)\, x \to P\, a\, Q\, (\mathsf{PNode}\, x)\big)\\ &\qquad\qquad \to \forall (a : \mathsf{Set})(Q : a \to \mathsf{Set})(x : \mathsf{PTree}\, a) \to \mathsf{PTree}^\wedge\, a\, Q\, x \to P\, a\, Q\, x\end{aligned}$$

where $\mathsf{Pair}^\wedge : \forall (a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to a \times a \to \mathsf{Set}$ lifts a predicate Q on a to a predicate on pairs of type $a \times a$, so that $\mathsf{Pair}^\wedge\, a\, Q\, (x, y) = Q\, x \times Q\, y$, and $\mathsf{PTree}^\wedge : \forall (a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{PTree}\, a \to \mathsf{Set}$ lifts a predicate Q on data of type a to a predicate on data of type $\mathsf{PTree}\, a$ 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 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 this technique will be 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) [Cheney and Hinze 2003; Sheard and Pasalic 2004; Xi et al. 2003] 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 [Pasalic and Linger 2004; Peyton Jones et al. 2006; Pottier and Régis-Gianas 2006], to improve memory performance [Minsky 2015], and to design APIs [Penner 2020].

As a first and notable example of GADT, we consider the 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

$$\begin{aligned}&\mathsf{data\ Equal} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set\ where}\\ &\qquad \mathsf{Refl} :\ \mathsf{Equal\ c\ c}\end{aligned} \tag{4}$$

Equal has thus a single data element when its two type arguments are the same and no data elements otherwise.

A more complex example for a GADT is

$$\begin{aligned}&\mathsf{data\ Seq} : \mathsf{Set} \to \mathsf{Set\ where}\\ &\qquad \mathsf{Const} :\ \mathsf{a} \to \mathsf{Seq\ a}\\ &\qquad \mathsf{SPair}\ \ :\ \mathsf{Seq\ a} \to \mathsf{Seq\ b} \to \mathsf{Seq}\ (\mathsf{a} \times \mathsf{b})\end{aligned} \tag{5}$$

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 [Cheney and Hinze 2003; Sheard and Pasalic 2004]. For example, we can rewrite the Seq type as

$$\begin{aligned}&\mathsf{data\ Seq}\ (\mathsf{a} : \mathsf{Set}) : \mathsf{Set\ where}\\ &\qquad \mathsf{Const} :\ \mathsf{a} \to \mathsf{Seq\ a}\\ &\qquad \mathsf{SPair}\ \ :\ \exists(\mathsf{b\ c} : \mathsf{Set}).\, \mathsf{Equal\ a}\ (\mathsf{b} \times \mathsf{c}) \to \mathsf{Seq\ b} \to \mathsf{Seq\ c} \to \mathsf{Seq\ a}\end{aligned} \tag{6}$$

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 [Atkey 2012; Vytiniotis and Weirich 2010].

STLC example

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 STLC 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:

$$\begin{aligned}&\mathsf{data\ LType}\ (\mathsf{a} : \mathsf{Set}) : \mathsf{Set\ where}\\ &\qquad \mathsf{TBool} :\ \exists(\mathsf{b} : \mathsf{Set}).\, \mathsf{Equal\ a}\ (\mathsf{KBool\ b}) \to \mathsf{LType\ a}\\ &\qquad \mathsf{TArr}\ \ :\ \exists(\mathsf{b} : \mathsf{Set})(\mathsf{c} : \mathsf{Set}).\, \mathsf{Equal\ a}\ (\mathsf{b} \to \mathsf{c}) \to \mathsf{LType\ b} \to \mathsf{LType\ c} \to \mathsf{LType\ a}\\ &\qquad \mathsf{TList}\ \ :\ \exists(\mathsf{b} : \mathsf{Set}).\, \mathsf{Equal\ a}\ (\mathsf{List\ \ b}) \to \mathsf{LType\ b} \to \mathsf{LType\ a}\end{aligned} \tag{7}$$

The TBool constructor represents the type of booleans in the calculus. It is defined in terms of KBool : $\forall$ (a : Set) $\to$ Set, which simply returns Bool on any input:

$$\mathsf{KBool\ a} = \mathsf{Bool}$$

The lifting $\mathsf{KBool}^{\wedge} : \forall(\mathsf{a} : \mathsf{Set}) \to (\mathsf{a} \to \mathsf{Set}) \to (\mathsf{KBool\ a}) \to \mathsf{Set}$ lifts every predicate to the constantly true predicate:

$$\mathsf{KBool}^{\wedge}\ \mathsf{a}\ \mathsf{Q_a}\ \mathsf{b} = \top$$

The type system also includes an arrow type constructor, TArr , which takes two types b and c and produces the arrow type $\mathsf{b} \to \mathsf{c}$. Similarly, given a type b, we can construct the type List b using the TList constructor.

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.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{data\ LTerm\ (a : Set) : Set\ where}} \\
\quad \mathsf{Var} & : & \mathsf{String} \to \mathsf{LType\ a} \to \mathsf{LTerm\ a} \\
\quad \mathsf{Abs} & : & \exists(\mathsf{b : Set})(\mathsf{c : Set}).\ \mathsf{Equal\ a\ (b \to c)} \to \mathsf{String} \to \mathsf{LType\ b} \to \mathsf{LTerm\ c} \to \mathsf{LTerm\ a} \\
\quad \mathsf{App} & : & \exists(\mathsf{b : Set}).\ \mathsf{LTerm (b \to a)} \to \mathsf{LTerm\ b} \to \mathsf{LTerm\ a} \\
\quad \mathsf{ListC} & : & \exists(\mathsf{b : Set}).\ \mathsf{Equal\ a\ (List\ \ b)} \to \mathsf{List\ (LTerm\ b)} \to \mathsf{LTerm\ a}
\end{array}
\tag{8}
$$

The type parameter for LTerm tracks the types of STLC 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 $\mathsf{b} \to \mathsf{c}$. The App constructor applies a lambda term of type $\mathsf{b} \to \mathsf{a}$ 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 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, $\mathsf{Bool} \to \mathsf{Bool}$, $\mathsf{List\ (Bool \to Bool)}$, 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

As we have seen in Section 2.2, truly nested types do not support a structural induction rule, which is the reason why it was necessary to introduce a deep induction rule supporting them. Consequently, GADTs do not support a structural induction rule either, as they generalize nested types. Still, there is hope for GADTs to support a deep induction rule, like nested types do.

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 [Johann and Polonsky 2019]. 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.

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 GADTs, and then provide a general principle that works for GADTs not featuring nesting in their definition.

4.1 (Deep) induction for Equal

As a first example, we derive the induction rule for the Equal type from Equation 4. 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 G a. the predicate-lifting function for Equal is the function

$$\mathsf{Equal}^{\wedge} : \forall(\mathsf{a\ b : Set}) \to (\mathsf{a} \to \mathsf{Set}) \to (\mathsf{b} \to \mathsf{Set}) \to \mathsf{Equal\ a\ b} \to \mathsf{Set}$$

defined as

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

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 CRefl be the following function associated to the Refl constructor:

$$\lambda(\mathsf{P} : \forall(\mathsf{a}\ \mathsf{b} : \mathsf{Set}) \to (\mathsf{a} \to \mathsf{Set}) \to (\mathsf{b} \to \mathsf{Set}) \to \mathsf{Equal}\ \mathsf{a}\ \mathsf{b} \to \mathsf{Set}) \\ \to \forall(\mathsf{c} : \mathsf{Set})(\mathsf{Q} : \mathsf{c} \to \mathsf{Set})(\mathsf{Q}' : \mathsf{c} \to \mathsf{Set}) \to \mathsf{Equal}^\wedge\ \mathsf{c}\ \mathsf{c}\ \mathsf{Q}\ \mathsf{Q}'\ \mathsf{Refl} \to \mathsf{P}\ \mathsf{c}\ \mathsf{c}\ \mathsf{Q}\ \mathsf{Q}'\ \mathsf{Refl}$$

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

$$\forall(\mathsf{P} : \forall(\mathsf{a}\ \mathsf{b} : \mathsf{Set}) \to (\mathsf{a} \to \mathsf{Set}) \to (\mathsf{b} \to \mathsf{Set}) \to \mathsf{Equal}\ \mathsf{a}\ \mathsf{b} \to \mathsf{Set}) \\ \to \mathsf{CRefl}\ \mathsf{P} \to \forall(\mathsf{a}\ \mathsf{b} : \mathsf{Set})(\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set})(\mathsf{Q_b} : \mathsf{b} \to \mathsf{Set})(\mathsf{e} : \mathsf{Equal}\ \mathsf{a}\ \mathsf{b}) \\ \to \mathsf{Equal}^\wedge\ \mathsf{a}\ \mathsf{b}\ \mathsf{Q_a}\ \mathsf{Q_b}\ \mathsf{e} \to \mathsf{P}\ \mathsf{a}\ \mathsf{b}\ \mathsf{Q_a}\ \mathsf{Q_b}\ \mathsf{e}$$

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 define a term DIEqual of the above type as

$$\mathsf{DIEqual}\ \mathsf{P}\ \mathsf{crefl}\ \mathsf{a}\ \mathsf{a}\ \mathsf{Q_a}\ \mathsf{Q'_a}\ \mathsf{Refl}\ \mathsf{L_E} = \mathsf{crefl}\ \mathsf{a}\ \mathsf{Q_a}\ \mathsf{Q'_a}\ \mathsf{L_E}$$

where $\mathsf{crefl} : \mathsf{CRefl}\ \mathsf{P}$, $\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}$, $\mathsf{Q'_a} : \mathsf{a} \to \mathsf{Set}$ and $\mathsf{L_E} : \mathsf{Equal}^\wedge\ \mathsf{a}\ \mathsf{a}\ \mathsf{Q_a}\ \mathsf{Q'_a}\ \mathsf{Refl}$. 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 SIEqual,

$$\forall(\mathsf{Q} : \forall(\mathsf{a}\ \mathsf{b} : \mathsf{Set}) \to \mathsf{Equal}\ \mathsf{a}\ \mathsf{b} \to \mathsf{Set}) \\ \to \big(\forall(\mathsf{c} : \mathsf{Set}) \to \mathsf{P}\ \mathsf{c}\ \mathsf{c}\ \mathsf{Refl}\big) \to \forall(\mathsf{a}\ \mathsf{b} : \mathsf{Set})(\mathsf{e} : \mathsf{Equal}\ \mathsf{a}\ \mathsf{b}) \to \mathsf{P}\ \mathsf{a}\ \mathsf{b}\ \mathsf{e}$$

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 SIEqual as

$$\mathsf{SIEqual}\ \mathsf{Q}\ \mathsf{srefl}\ \mathsf{a}\ \mathsf{b}\ \mathsf{e} = \mathsf{DIEqual}\ \mathsf{P}\ \mathsf{srefl}\ \mathsf{a}\ \mathsf{b}\ \mathsf{K}^{\mathsf{a}}_1\ \mathsf{K}^{\mathsf{b}}_1\ \mathsf{e}\ \mathsf{L_E}$$

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 $\mathsf{P}\ \mathsf{a}\ \mathsf{b}\ \mathsf{Q_a}\ \mathsf{Q_b}\ \mathsf{e} = \mathsf{Q}\ \mathsf{a}\ \mathsf{b}\ \mathsf{e}$;
- $\mathsf{K}^{\mathsf{a}}_1$ and $\mathsf{K}^{\mathsf{b}}_1$ are the constantly 1-valued predicates on, respectively, a and b;
- $\mathsf{L_E} : \mathsf{Equal}^\wedge\ \mathsf{a}\ \mathsf{b}\ \mathsf{K}^{\mathsf{a}}_1\ \mathsf{K}^{\mathsf{b}}_1\ \mathsf{e}$ is defined by pattern matching, i.e., in case $\mathsf{a} = \mathsf{b}$ and $\mathsf{e} = \mathsf{Refl}$, it is defined as $\mathsf{L_E}\ \mathsf{x} = \mathsf{Refl} : \mathsf{Equal}\ \mathsf{a}\ \mathsf{a}$.

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 Equation 6. 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

$$\mathsf{Seq}^\wedge\ \mathsf{a}\ \mathsf{Q_a}\ (\mathsf{Const}\ \mathsf{x}) = \mathsf{Q_a}\ \mathsf{x}$$

where $Q_a : a \to \mathrm{Set}$ and $x : a$, and

$$
\begin{aligned}
&\mathrm{Seq}^\wedge\ a\ Q_a\ (\mathrm{SPair}\ b\ c\ e\ s_b\ s_c)\\
&= \exists (Q_b : b \to \mathrm{Set})(Q_c : c \to \mathrm{Set}) \to \mathrm{Equal}^\wedge\ a\ (b \times c)\ Q_a\ (Q_b \times Q_c)\ e \times \mathrm{Seq}^\wedge\ b\ Q_b\ s_b \times \mathrm{Seq}^\wedge\ c\ Q_c\ s_c
\end{aligned}
$$

where $e : \mathrm{Equal}\ a\ (b \times c)$, $s_b : \mathrm{Seq}\ b$ and $s_c : \mathrm{Seq}\ c$. We also need to define the lifting of predicates over the polymorphic type of pairs, $\mathrm{Pair} = \forall (b\ c : \mathrm{Set}) \to b \times c$, which is

$$\mathrm{Pair}^\wedge : \forall (b\ c : \mathrm{Set}) \to (b \to \mathrm{Set}) \to (c \to \mathrm{Set}) \to \mathrm{Pair}\ b\ c \to \mathrm{Set}$$

and it is defined as

$$\mathrm{Pair}^\wedge\ b\ c\ Q_b\ Q_c\ (y, z) = Q_b\ y \times Q_c\ z$$

where $Q_b : b \to \mathrm{Set}$, $Q_c : c \to \mathrm{Set}$, $y : b$ and $z : c$.

Finally, let CConst be the function

$$
\begin{aligned}
&\lambda (P : \forall (a : \mathrm{Set}) \to (a \to \mathrm{Set}) \to \mathrm{Seq}\ a \to \mathrm{Set})\\
&\qquad\qquad \to \forall (a : \mathrm{Set})(Q_a : a \to \mathrm{Set})(x : a) \to Q_a\ x \to P\ a\ Q_a\ (\mathrm{Const}\ x)
\end{aligned}
$$

associated to the Const constructor, and let CSPair be the function

$$
\begin{aligned}
&\lambda (P : \forall (a : \mathrm{Set}) \to (a \to \mathrm{Set}) \to \mathrm{Seq}\ a \to \mathrm{Set})\\
&\qquad \to \forall (a\ b\ c : \mathrm{Set})(Q_a : a \to \mathrm{Set})(Q_b : b \to \mathrm{Set})(Q_c : c \to \mathrm{Set})\\
&\quad (s_b : \mathrm{Seq}\ b)(s_c : \mathrm{Seq}\ c)(e : \mathrm{Equal}\ a\ (b \times c)) \to \mathrm{Equal}^\wedge a\ (b \times c)\ Q_a\ (\mathrm{Pair}^\wedge\ b\ c\ Q_b\ Q_c)\ e\\
&\qquad\qquad \to P\ b\ Q_b\ s_b \to P\ c\ Q_c\ s_c \to P a Q_a (\mathrm{SPair}\ b\ c\ e\ s_b\ s_c)
\end{aligned}
$$

associated to the SPair constructor,

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

$$
\begin{aligned}
&\forall (P : \forall (a : \mathrm{Set}) \to (a \to \mathrm{Set}) \to \mathrm{Seq}\ a \to \mathrm{Set}) \to \mathrm{CConst}\ P \to \mathrm{CSPair}\ P\\
&\qquad\qquad \to \forall (a : \mathrm{Set})(Q_a : a \to \mathrm{Set})(s_a : \mathrm{Seq}\ a) \to \mathrm{Seq}^\wedge\ a\ Q_a\ s_a \to P\ a\ Q_a\ s_a
\end{aligned}
$$

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

$$\mathrm{DISeq}\ P\ \mathrm{cconst}\ \mathrm{cspair}\ a\ Q_a\ s_a\ L_a : P\ a\ Q_a\ s_a$$

where $P : \forall (a : \mathrm{Set}) \to (a \to \mathrm{Set}) \to \mathrm{Seq}\ a \to \mathrm{Set}$, $\mathrm{cconst} : \mathrm{CConst}\ P$, $\mathrm{cspair} : \mathrm{CSPair}\ P$, $Q_a : a \to \mathrm{Set}$, $s_a : \mathrm{Seq}\ a$ and $L_a : \mathrm{Seq}^\wedge\ a\ Q\ s_a$, and we proceed by pattern-matching on s_a. Let $s_a = \mathrm{Const}\ x$ for $x : a$, and define

$$\mathrm{DISeq}\ P\ \mathrm{cconst}\ \mathrm{cspair}\ a\ Q_a\ (\mathrm{Const}\ x)\ L_a = \mathrm{cconst}\ a\ Q_a\ x\ L_a$$

Notice that $\mathrm{Seq}^\wedge\ a\ Q_a\ (\mathrm{Const}\ x) = Q_a\ x$, and thus $L_a : Q_a\ x$, making the right-hand-side in the above expression type-check. Now, let $s_a = \mathrm{SPair}\ b\ c\ e\ s_b\ s_c$ for $e : \mathrm{Equal}\ a\ (b \times c)$, $s_b : \mathrm{Seq}\ b$ and $s_c : \mathrm{Seq}\ c$, and define

$$\mathrm{DISeq}\ P\ \mathrm{cconst}\ \mathrm{cspair}\ a\ Q_a\ (\mathrm{SPair}\ b\ c\ e\ s_b\ s_c)\ (Q_b, Q_c, L_e, L_b, L_c) = \mathrm{cspair}\ a\ b\ c\ Q_a\ Q_b\ Q_c\ s_b\ s_c\ e\ L_e\ p_b\ p_c$$

where $(Q_b, Q_c, L_e, L_b, L_c) : \mathrm{Seq}^\wedge\ a\ Q\ (\mathrm{SPair}\ b\ c\ e\ x\ y)$, i.e.,

- $Q_b : b \to \mathrm{Set}$ and $Q_c : c \to \mathrm{Set}$;
- $L_e : \mathrm{Equal}^\wedge\ a\ (b \times c)\ Q_a\ (Q_b \times Q_c)\ e$;
- $L_b : \mathrm{Seq}^\wedge\ b\ Q_b\ s_b$ and $L_c : \mathrm{Seq}^\wedge\ c\ Q_c\ s_c$;

and p_b and p_c are defined as follows:

$$
\begin{aligned}
p_b &= \mathrm{DISeq}\ P\ \mathrm{cconst}\ \mathrm{cspair}\ b\ Q_b\ s_b\ L_b : P\ b\ Q_b\ s_b\\
p_c &= \mathrm{DISeq}\ P\ \mathrm{cconst}\ \mathrm{cspair}\ c\ Q_c\ s_c\ L_c : P\ c\ Q_c\ s_c
\end{aligned}
$$

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, $(T_b : \mathsf{LType}\ b)$ or $(t_c : \mathsf{LTerm}\ c)$.

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

$$\mathsf{Arr}^\wedge : \forall (a\ b : \mathsf{Set}) \to (a \to \mathsf{Set}) \to (b \to \mathsf{Set}) \to (a \to b) \to \mathsf{Set}$$

is defined as:

$$\mathsf{Arr}^\wedge\ a\ b\ Q_a\ Q_b\ f = \forall (x : a) \to Q_a\ x \to Q_b\ (f\ x)$$

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

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

and is defined by pattern-matching as

$$\mathsf{LType}^\wedge\ a\ Q_a\ (\mathsf{TBool}\ b\ e) = \exists (Q_b : b \to \mathsf{Set}) \to \mathsf{Equal}^\wedge\ a\ b\ Q_a\ (\mathsf{KBool}^\wedge\ b\ Q_b)\ e$$

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

$$\begin{aligned}&\mathsf{LType}^\wedge\ a\ Q_a\ (\mathsf{TArr}\ b\ c\ e\ T_b\ T_c)\\ &\quad = \exists (Q_b : b \to \mathsf{Set})(Q_c : c \to \mathsf{Set}) \to \mathsf{Equal}^\wedge\ a\ (b \to c)\ Q_a\ (\mathsf{Arr}^\wedge\ b\ c\ Q_b\ Q_c)\ e\\ &\qquad\qquad \times \mathsf{LType}^\wedge\ b\ Q_b\ T_b \times \mathsf{LType}^\wedge\ c\ Q_c\ T_c\end{aligned}$$

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

$$\begin{aligned}&\mathsf{LType}^\wedge\ a\ Q_a\ (\mathsf{TList}\ b\ e\ T_b)\\ &\quad = \exists (Q_b : b \to \mathsf{Set}) \to \mathsf{Equal}^\wedge\ a\ (\mathsf{List}\ b)\ Q_a\ (\mathsf{List}^\wedge\ b\ Q_b)\ e \times \mathsf{LType}^\wedge\ b\ Q_b\ T_b\end{aligned}$$

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

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

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

is defined by pattern-matching as

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

where $s : \mathsf{String}$ and $T_a : \mathsf{LType}\ a$, and

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

where $e : \mathsf{Equal}\ a\ (b \to c)$, $s : \mathsf{String}$, $T_b : \mathsf{LType}\ b$, and $t_c : \mathsf{LTerm}\ c$, and

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

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

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

where $e : \mathsf{Equal}\ a\ (\mathsf{List}\ b)$ and $ts : \mathsf{List}\ (\mathsf{LTerm} b)$. Notice we use the lifting for List in the case for the ListC constructor.

With these liftings, we can define the deep induction princple for LTerm. Let CVar be the type

$$\lambda(P : \forall(a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{LTerm}\ a \to \mathsf{Set})$$
$$\to \forall(a : \mathsf{Set})(Q_a : a \to \mathsf{Set})(s : \mathsf{String})(T_a : \mathsf{LType}\ a) \to \mathsf{LType}^\wedge\ a\ Q_a\ T_a \to P\ a\ Q_a\ (\mathsf{Var}\ s\ T_a)$$

associated to the Var constructor. Let CAbs be the type

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

associated to the Abs constructor. Let CApp be the type

$$\lambda(P : \forall(a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{LTerm}\ a \to \mathsf{Set})$$
$$\to \forall(a\ b : \mathsf{Set})(Q_a : a \to \mathsf{Set})(Q_b : b \to \mathsf{Set})(t_{ba} : \mathsf{LTerm}\ (b \to a))(t_b : \mathsf{LTerm}\ b)$$
$$\to P\ (b \to a)\ (\mathsf{Arr}^\wedge\ b\ a\ Q_b\ Q_a)\ t_{ba} \to P\ b\ Q_b\ t_b \to P\ a\ Q_a\ (\mathsf{App}\ b\ t_{ba}\ t_b)$$

associated to the App constructor. Let CListC be the type

$$\lambda(P : \forall(a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{LTerm}\ a \to \mathsf{Set})$$
$$\to \forall(a\ b : \mathsf{Set})(Q_a : a \to \mathsf{Set})(Q_b : b \to \mathsf{Set})(e : \mathsf{Equal}\ a\ (\mathsf{List}\ b))(ts : \mathsf{List}\ (\mathsf{LTerm}\ b))$$
$$\to \mathsf{Equal}^\wedge\ a\ (\mathsf{List}\ b)\ Q_a\ (\mathsf{List}^\wedge\ b\ Q_b)\ e \to \mathsf{List}^\wedge\ (\mathsf{LTerm}\ b)(P\ b\ Q_b)\ ts$$
$$\to P\ a\ Q_a\ (\mathsf{ListC}\ b\ e\ ts)$$

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

$$\forall(P : \forall(a : \mathsf{Set}) \to (a \to \mathsf{Set}) \to \mathsf{LTerm}\ a \to \mathsf{Set}) \to \mathsf{CVar}\ P \to \mathsf{CAbs}\ P \to \mathsf{CApp}\ P \to \mathsf{CListC}\ P$$
$$\to \forall(a : \mathsf{Set})(Q_a : a \to \mathsf{Set})(t_a : \mathsf{LTerm}\ a) \to \mathsf{LTerm}^\wedge\ a\ Q_a\ t_a \to P\ a\ Q_a\ t_a$$

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

$$\mathsf{DILTerm}\ P\ \mathit{cvar}\ \mathit{cabs}\ \mathit{capp}\ \mathit{clistc}\ a\ Q_a\ t_a\ L_a : P\ a\ Q_a\ t_a$$

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

$$\mathsf{DILTerm}\ P\ \mathit{cvar}\ \mathit{cabs}\ \mathit{capp}\ \mathit{clistc}\ a\ Q_a\ (\mathsf{Var}\ s\ T_a)\ L_a = \mathit{cvar}\ a\ Q_a\ s\ T_a\ L_a$$

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

$$\mathsf{DILTerm}\ P\ \mathit{cvar}\ \mathit{cabs}\ \mathit{capp}\ \mathit{clistc}\ a\ Q_a\ (\mathsf{Abs}\ b\ c\ e\ s\ T_b\ t_c)\ (Q_b, Q_c, L_e, L_{T_b}, L_{t_c})$$
$$= \mathit{cabs}\ a\ b\ c\ Q_a\ Q_b\ Q_c\ e\ s\ T_b\ t_c\ L_e\ L_{T_b}\ p_c$$

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

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

and pc is defined as:

$$\mathsf{pc} = \mathsf{DILTerm\ P\ cvar\ cabs\ capp\ clistc\ c}\ Q_c\ t_c\ L_{t_c} : \mathsf{P\ c}\ Q_c\ t_c$$

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

$$\mathsf{DILTerm\ P\ cvar\ cabs\ capp\ clistc\ a}\ Q_a\ (\mathsf{App\ b}\ t_{ba}\ t_b)\ (Q_b, L_{t_{ba}}, L_{t_b}) = \mathsf{capp\ a\ b}\ Q_a\ Q_b\ t_{ba}\ t_b\ p_{ba}\ p_b$$

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

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

and p_{ba} and p_b and are defined as:

$$p_{ba} = \mathsf{DILTerm\ P\ cvar\ cabs\ capp\ clistc}\ (\mathsf{b} \to \mathsf{a})\ (\mathsf{Arr}^\wedge\ \mathsf{b\ a}\ Q_b\ Q_a)\ t_{ba}\ L_{t_{ba}} : \mathsf{P}\ (\mathsf{b} \to \mathsf{a})\ (\mathsf{Arr}^\wedge\ \mathsf{b\ a}\ Q_b\ Q_a)\ t_{ba}$$
$$p_b = \mathsf{DILTerm\ P\ cvar\ cabs\ capp\ clistc\ b}\ Q_b\ t_b\ L_{t_b} : \mathsf{P\ b}\ Q_b\ t_b$$

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

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

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

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

and p_{List} is defined as:

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

and p_{ts} is defined as:

$$p_{ts} = \mathsf{DILTerm\ P\ cvar\ cabs\ capp\ clistc\ b}\ Q_b : \mathsf{PredMap}\ (\mathsf{LTerm}^\wedge\ \mathsf{b}\ Q_b)\ (\mathsf{P\ b}\ Q_b)$$

where

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

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

$$\mathsf{PredMap} : \forall\,\{\mathsf{a} : \mathsf{Set}\} \to (Q_a\ Q'_a : \mathsf{a} \to \mathsf{Set}) \to \mathsf{Set}$$
$$\mathsf{PredMap}\ Q_a\ Q'_a = \forall\,(\mathsf{x} : \mathsf{A}) \to Q_a\ \mathsf{x} \to Q'_a\ \mathsf{x}$$

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

$$\mathsf{data\ G}\ (\overline{\mathsf{a} : \mathsf{Set}}) : \mathsf{Set\ where}$$
$$C_i\ :\ F_i\ \mathsf{G}\ \overline{\mathsf{b}} \to \mathsf{G}(\overline{K_i\ \overline{\mathsf{b}}})$$

is defined by a finite number of constructors C_i. In the definition above, F_i is a type constructor with signature $(\mathsf{Set}^\alpha \to \mathsf{Set}) \to \mathsf{Set}^\beta \to \mathsf{Set}$ and each 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: $\overline{\mathsf{a}}$ 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 $\overline{\mathsf{b}}$ of length β. Moreover, notice that the arity of G matches the number of type constructors $\overline{\mathsf{K_i}}$. 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.
- $\mathsf{F_i} = \mathsf{F_i'} \rightarrow \mathsf{F_i''}$ where $\mathsf{F_i'}$ does not contain the recursive variable, i.e., $\mathsf{F_i'} : \mathsf{Set}^{\beta} \rightarrow \mathsf{Set}$ 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.
- $\mathsf{F_i\,G\,\overline{b}} = \mathsf{G}(\overline{\mathsf{F_a\,\overline{b}}})$ where none of the $\mathsf{F_a}$ contains the recursive variable, i.e., $\mathsf{F_a} : \mathsf{Set}^{\beta} \rightarrow \mathsf{Set}$ 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.
- $\mathsf{F_i\,G\,\overline{b}} = \mathsf{H\,\overline{b}}$ where H is a type constructor of arity β not containing the recursive variable, i.e., $\mathsf{H} : \mathsf{Set}^{\beta} \rightarrow \mathsf{Set}$. Notice that this covers the case in which $\mathsf{F_i}$ is a closed type, so, in particular, the unit and empty types, 1 and 0.
- $\mathsf{F_i\,G\,\overline{b}} = \mathsf{H}(\overline{\mathsf{F_c\,G\,\overline{b}}})$ where H is a γ-ary type constructor not containing the recursive variable, i.e., $\mathsf{H} : \mathsf{Set}^{\gamma} \rightarrow \mathsf{Set}$, and $\mathsf{F_c}$ has the same signature as $\mathsf{F_i}$ and is built recursively from the same induction rules, for every $\mathsf{c} = 1 \ldots \gamma$. 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 H admits functorial semantics [Johann and Polonsky 2020], and thus we have a map function for $\mathsf{H}^\wedge$,

$$\mathsf{HLMap} : \forall\overline{(\mathsf{c} : \mathsf{Set})}\,\overline{(\mathsf{Q_c}\ \mathsf{Q_c'} : \mathsf{c} \rightarrow \mathsf{Set})} \rightarrow \overline{\mathsf{PredMap\,c\,Q_c\,Q_c'}} \rightarrow \mathsf{PredMap}\,(\mathsf{H}\,\overline{\mathsf{c}})\,(\mathsf{H}^\wedge\,\overline{\mathsf{c}}\,\overline{\mathsf{Q_c}})\,(\mathsf{H}^\wedge\,\overline{\mathsf{c}}\,\overline{\mathsf{Q_c'}})$$

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

$$\mathsf{PredMap\,c\,Q_c\,Q_c'} = \forall(\mathsf{x} : \mathsf{c}) \rightarrow \mathsf{Q_c\,x} \rightarrow \mathsf{Q_c'\,x}$$

and represents the type of morphisms between predicates. 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):

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

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,

$$\begin{array}{c}\mathsf{data\ G\ (a : Set) : Set\ where} \\ \mathsf{C\ :\ F\,G\,\overline{b} \rightarrow G(K\,\overline{b})}\end{array} \tag{9}$$

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

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

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

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

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,

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

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

$$\begin{aligned}\mathsf{CC} = \lambda(\mathsf{P} : \forall(\mathsf{a} : \mathsf{Set}) \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{G}\,\mathsf{a} \to \mathsf{Set})&\\ \to \forall(\mathsf{a} : \mathsf{Set})(\overline{\mathsf{b} : \mathsf{Set}})(\mathsf{Q}_{\mathsf{a}} : \mathsf{a} \to \mathsf{Set})(\overline{\mathsf{Q}_{\mathsf{b}} : \mathsf{b} \to \mathsf{Set}})(\mathsf{e} : \mathsf{Equal}\,\mathsf{a}\,(\mathsf{K}\,\overline{\mathsf{b}}))(\mathsf{x} : \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{b}})&\\ \to \mathsf{Equal}^{\wedge}\,\mathsf{a}\,(\mathsf{K}\,\overline{\mathsf{b}})\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{K}^{\wedge}\,\overline{\mathsf{b}}\,\overline{\mathsf{Q}_{\mathsf{b}}})\,\mathsf{e} \to \mathsf{F}^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q}_{\mathsf{b}}}\,\mathsf{x} \to \mathsf{P}\,\mathsf{a}\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{C}\,\overline{\mathsf{b}}\,\mathsf{e}\,\mathsf{x})&\end{aligned}$$

to the constructor C.

The induction rule for G is

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

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

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

where $\mathsf{cc} : \mathsf{CC}\,\mathsf{P}$ 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 $\mathsf{x} : \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{b}}$;
- $(\overline{\mathsf{Q}_{\mathsf{b}}}, \mathsf{L}_{\mathsf{E}}, \mathsf{L}_{\mathsf{F}}) : \mathsf{G}^{\wedge}\,\mathsf{a}\,\mathsf{Q}_{\mathsf{a}}(\mathsf{C}\,\overline{\mathsf{b}}\,\mathsf{e}\,\mathsf{x})$, i.e., $\mathsf{Q}_{\mathsf{b}} : \mathsf{b} \to \mathsf{Set}$ for each b, $\mathsf{L}_{\mathsf{E}} : \mathsf{Equal}^{\wedge}\,\mathsf{a}\,(\mathsf{K}\,\overline{\mathsf{b}})\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{K}^{\wedge}\,\overline{\mathsf{b}}\,\overline{\mathsf{Q}_{\mathsf{b}}})\,\mathsf{e}$, and $\mathsf{L}_{\mathsf{F}} : \mathsf{F}^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{G}^{\wedge}\,\overline{\mathsf{Q}_{\mathsf{b}}}\,\mathsf{x}$.

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

$$\mathsf{F}^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}} = \mathsf{Pair}^{\wedge}(\mathsf{F}_1\,\mathsf{G}\,\overline{\mathsf{b}})(\mathsf{F}_2\,\mathsf{G}\,\overline{\mathsf{b}})(\mathsf{F}_1^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}})(\mathsf{F}_2^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}})$$

By inductive hypothesis, there exist morphisms of predicates

$$\mathsf{p}_1 : \mathsf{PredMap}\,(\mathsf{F}_1\,\mathsf{G}\,\overline{\mathsf{b}})\,((\mathsf{F}_1)^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{G}^{\wedge}\,\overline{\mathsf{Q_b}})((\mathsf{F}_1)^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}})$$
$$\mathsf{p}_2 : \mathsf{PredMap}\,(\mathsf{F}_2\,\mathsf{G}\,\overline{\mathsf{b}})\,((\mathsf{F}_2)^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{G}^{\wedge}\,\overline{\mathsf{Q_b}})((\mathsf{F}_2)^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}})$$

Thus, we define $\mathsf{p}(\mathsf{x}_1,\mathsf{x}_2)(\mathsf{L}_1,\mathsf{L}_2) = (\mathsf{p}_1\,\mathsf{x}_1\,\mathsf{L}_1, \mathsf{p}_2\,\mathsf{x}_2\,\mathsf{L}_2)$ for $\mathsf{x}_1 : \mathsf{F}_1\,\mathsf{G}\,\overline{\mathsf{b}}$, $\mathsf{L}_1 : \mathsf{F}_1^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{G}^{\wedge}\,\overline{\mathsf{Q_b}}\,\mathsf{x}_1$, $\mathsf{x}_2 : \mathsf{F}_2\,\mathsf{G}\,\overline{\mathsf{b}}$ and $\mathsf{L}_2 : \mathsf{F}_2^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{G}^{\wedge}\,\overline{\mathsf{Q_b}}\,\mathsf{x}_2$.
- 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 $\mathsf{F} = \mathsf{F}_1 \to \mathsf{F}_2$ where F_1 does not contain the recursive variable, i.e., $\mathsf{F}_1 : \mathsf{Set}^{\beta} \to \mathsf{Set}$, and F_2 has the same signature as F. We have that

$$\mathsf{F}^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}}\,\mathsf{x} = \forall(\mathsf{z} : \mathsf{F}_1\,\overline{\mathsf{b}}) \to \mathsf{F}_1^{\wedge}\,\overline{\mathsf{b}}\,\overline{\mathsf{Q_b}}\,\mathsf{z} \to \mathsf{F}_2^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}}\,(\mathsf{x}\,\mathsf{z})$$

where $\mathsf{x} : \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{b}} = \mathsf{F}_1\,\overline{\mathsf{b}} \to \mathsf{F}_2\,\mathsf{G}\,\overline{\mathsf{b}}$. By inductive hypothesis, there exist a morphism of predicates

$$\mathsf{p}_2 : \mathsf{PredMap}\,(\mathsf{F}_2\,\mathsf{G}\,\overline{\mathsf{b}})(\mathsf{F}_2^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{G}^{\wedge}\,\overline{\mathsf{Q_b}})(\mathsf{F}_2^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}})$$

Thus, we define $\mathsf{p}\,\mathsf{x}\,\mathsf{L_F} : \mathsf{F}^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}}\,\mathsf{x}$ for $\mathsf{L_F} : \mathsf{F}^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{G}^{\wedge}\,\overline{\mathsf{Q_b}}\,\mathsf{x}$ as $\mathsf{p}\,\mathsf{x}\,\mathsf{L_F}\,\mathsf{z}\,\mathsf{L}_1 = \mathsf{p}_2(\mathsf{x}\,\mathsf{z})(\mathsf{L_F}\,\mathsf{z}\,\mathsf{L}_1)$ for $\mathsf{z} : \mathsf{F}_1\,\overline{\mathsf{b}}$ and $\mathsf{L}_1 : \mathsf{F}_1^{\wedge}\,\overline{\mathsf{b}}\,\overline{\mathsf{Q_b}}\,\mathsf{z}$. Notice that F_1 not containing the recursive variable is a necessary restriction, as the proof relies on $\mathsf{F}^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{G}^{\wedge}\,\overline{\mathsf{Q_b}}\,\mathsf{x}$ and $\mathsf{F}^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}}\,\mathsf{x}$ having the same domain $\mathsf{F}_1^{\wedge}\,\overline{\mathsf{b}}\,\overline{\mathsf{Q_b}}\,\mathsf{z}$.
- Case $\mathsf{F}\,\mathsf{G}\,\overline{\mathsf{b}} = \mathsf{G}(\mathsf{F}'\,\overline{\mathsf{b}})$ where F' does not contain the recursive variable, i.e., $\mathsf{F}' : \mathsf{Set}^{\beta} \to \mathsf{Set}$. Thus, $\mathsf{F}^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}} = \mathsf{P}(\mathsf{F}'\,\overline{\mathsf{b}})(\mathsf{F}'^{\wedge}\,\overline{\mathsf{b}}\,\overline{\mathsf{Q_b}})$. So, p is defined as

$$\mathsf{p} = \mathsf{DIG}\,\mathsf{P}\,\mathsf{cc}\,(\mathsf{F}'\,\overline{\mathsf{b}})\,(\mathsf{F}'^{\wedge}\,\overline{\mathsf{b}}\,\overline{\mathsf{Q_b}})$$

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

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

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

$$\forall(\overline{\mathsf{c} : \mathsf{Set}})(\overline{\mathsf{Q_c}\ \mathsf{Q}'_\mathsf{c} : \mathsf{c} \to \mathsf{Set}}) \to \mathsf{PredMap}\,\overline{\mathsf{c}}\,\overline{\mathsf{Q_c}}\,\overline{\mathsf{Q}'_\mathsf{c}} \to \mathsf{PredMap}\,(\mathsf{H}\,\overline{\mathsf{c}})\,(\mathsf{H}^{\wedge}\,\overline{\mathsf{c}}\,\overline{\mathsf{Q_c}})\,(\mathsf{H}^{\wedge}\,\overline{\mathsf{c}}\,\overline{\mathsf{Q}'_\mathsf{c}})$$

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 [Johann et al. 2021; Johann and Polonsky 2019]. Thus,

$$\mathsf{F}^{\wedge}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}} = \mathsf{H}^{\wedge}(\overline{\mathsf{F_c}\,\mathsf{G}\,\overline{\mathsf{b}}})(\overline{\mathsf{F_c^{\wedge}}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}}})$$

By induction hypothesis, there is a morphism of predicates

$$\mathsf{p_c} : \mathsf{PredMap}\,(\mathsf{F_c}\,\mathsf{G}\,\overline{\mathsf{b}})(\mathsf{F_c^{\wedge}}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{G}^{\wedge}\,\overline{\mathsf{Q_b}})(\mathsf{F_c^{\wedge}}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}})$$

for every $\mathsf{c} = 1 \ldots \gamma$. So, p is defined as

$$\mathsf{p} = \mathsf{HLMap}\,(\overline{\mathsf{F_c}\,\mathsf{G}\,\overline{\mathsf{b}}})(\overline{\mathsf{F_c^{\wedge}}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{G}^{\wedge}\,\overline{\mathsf{Q_b}}})(\overline{\mathsf{F_c^{\wedge}}\,\mathsf{G}\,\overline{\mathsf{b}}\,\mathsf{P}\,\overline{\mathsf{Q_b}}})\,\overline{\mathsf{p_c}}$$

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 [Johann and Polonsky 2020], 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.

$$\begin{array}{c}\mathrm{data\ G\ (a : Set) : Set\ where} \\ \mathrm{C :\ G(G\,a) \to G(a \times a)}\end{array} \tag{10}$$

The constructor C can be rewritten as

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

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

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

is defined as

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

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

$$\begin{aligned}&\mathrm{\lambda(P : \forall(a : Set) \to (a \to Set) \to G\ a \to Set)}\\ &\qquad\mathrm{\to \forall(a\ b : Set)(Q_a : a \to Set)(Q_b : b \to Set)(e : Equal\ a\ (b \times b))(x : G\ (G\ b))}\\ &\qquad\quad\mathrm{\to Equal^{\wedge}\ a\ (b \times b)\ Q_a\ (Pair^{\wedge}\ b\ b\ Q_b\ Q_b)\ e \to P\ (G\ b)\ (P\ b\ Q_b)\ x \to P\ a\ Q_a\ (C\ b\ e\ x)}\end{aligned}$$

associated to the C constructor.

The induction rule for G is

$$\begin{aligned}&\mathrm{\forall(P : \forall(a : Set) \to (a \to Set) \to G\ a \to Set) \to CC\ P}\\ &\qquad\qquad\qquad\qquad\mathrm{\to \forall(a : Set)(Q_a : a \to Set)(y : G\ a) \to G^{\wedge}\ a\ Q_a\ y \to P\ a\ Q_a\ y}\end{aligned}$$

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

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

where $\mathrm{cc : CC\ P}$ and

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

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

$$\mathrm{p = DIG\ P\ cc\ (G\ b)\ (P\ b\ Q_b)\ x\ q}$$

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

$$\mathrm{GLMap : \forall(a : Set)(Q_a\ Q'_a : a \to Set) \to PredMap\ a\ Q_a\ Q'_a \to PredMap\ (G\ a)\ (G^{\wedge}\ a\ Q_a)\ (G^{\wedge}\ a\ Q'_a)}$$

then we would be able to define

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

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

$$\mathsf{GLMap}\;a\;Q_a\;Q_a'\;M\;(C\;b\;e\;x)\;(Q_b, L_E, L_G) = (Q_b', L_E', L_G')$$

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

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

$(Q_b, L_E, L_G) : G^\wedge\;a\;Q_a\;(C\;b\;e\;x)$, i.e.,

- $Q_b : b \to \mathsf{Set}$;
- $L_E : \mathsf{Equal}^\wedge\;a\;(b \times b)\;Q_a\;(\mathsf{Pair}^\wedge\;b\;b\;Q_b\;Q_b)\;e$;
- $L_G : G^\wedge\;(G\;b)\;(G^\wedge\;b\;Q_b)\;x$;

and $(Q_b', L_E', L_G') : G^\wedge\;a\;Q_a'\;(C\;b\;e\;x)$, i.e.,

- $Q_b' : b \to \mathsf{Set}$;
- $L_E' : \mathsf{Equal}^\wedge\;a\;(b \times b)\;Q_a'\;(\mathsf{Pair}^\wedge\;b\;b\;Q_b'\;Q_b')\;e$;
- $L_G' : G^\wedge\;(G\;b)\;(G^\wedge\;b\;Q_b')\;x$;

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

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

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:

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

which we prove by

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

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

$$\mathrm{K1}\ x = \top$$

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

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

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

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

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

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

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

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 $(\mathrm{List}^{\wedge}\ (\mathrm{LTerm}\ b)\ (\mathrm{getType}\ b)\ ts)$ 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

i.e., that K1 is (extensionally) equal to the lifting $(\mathsf{Arr^\wedge}\ b\ c\ Q_b\ Q_c)$ 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:

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

Since we are working with proof-relevant predicates (i.e., functions into Set rather than functions into Bool), the lifting (Arr^ b c K1 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

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

and define the Abs case of LTerm^K1 as

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

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

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

The second component can be given using LTerm^K1, and we can define the first component using a proof of

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

and a map-like function

$$\begin{aligned}&\mathsf{LTerm^\wedge EqualMap} : \forall \{a : \mathsf{Set}\} \to (Q_a\ Q_a' : a \to \mathsf{Set}) \to (\mathsf{Equal^\wedge}\ a\ a\ Q_a\ Q_a'\ \mathsf{rfl})\\ &\qquad\qquad \to \mathsf{PredMap}\ (\mathsf{LTerm^\wedge}\ a\ Q_a)\ (\mathsf{LTerm^\wedge}\ a\ Q_a')\end{aligned}$$

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

Using LTerm^EqualMap, we can define the App case of LTerm^K1 as

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

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

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

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

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

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

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

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

using

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

to map a morphism of predicates

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

to a morphism of lifted predicates

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

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

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

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

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

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

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

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

6.2 Liftings of K1

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

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

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 $(\mathsf{F}^\wedge\ \mathsf{a}\ \mathsf{K1})$ 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 [Johann and Polonsky 2019] and paper Patricia Neil Clement 2010)

8 TODO

- reference (correctly) Haskell Symposium paper
- reference inspiration for STLC GADT : https://www.seas.upenn.edu/ cis194/spring15/lectures/11-stlc.html

REFERENCES

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

R. Bird and L. Meertens. 1998. Nested datatypes. In *Mathematics of Program Construction*.

J. Cheney and R. Hinze. 2003. *First-class phantom types*. Technical Report. Cornell University.

P. Johann, E. Ghiorzi, and D. Jeffries. 2021. Parametricity for Primitive Nested Types. In *Foundations of Software Science and Computation Structures*. 324–343.

P. Johann and A. Polonsky. 2019. Higher-kinded data types: Syntax and semantics. In *Logic in Computer Science*. 1–13.

P. Johann and A. Polonsky. 2020. Deep Induction: Induction Rules for (Truly) Nested Types. In *Foundations of Software Science and Computation Structures*. 339–358.

Y. Minsky. 2015. Why GADTs matter for performance. https://blog.janestreet.com/why-gadts-matter-for-performance/. (2015).

E. Pasalic and N. Linger. 2004. Meta-programming with typed object-language representations. In *Generic Programming and Component Engineering*. 136–167.

C. Penner. 2020. Simpler and safer API design using GADTs. https://chrispenner.ca/posts/gadt-design. (2020).

S. L. Peyton Jones, D. Vytiniotis, G. Washburn, and S. Weirich. 2006. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming*. 50–61.

F. Pottier and Y. Régis-Gianas. 2006. Stratified type inference for generalized algebraic data types. In *Principles of Programming Languages*. 232–244.

T. Sheard and E. Pasalic. 2004. Meta-programming with built-in type equality. In *Fourth International Workshop on Logical Frameworks and Meta-Languages*.

D. Vytiniotis and S. Weirich. 2010. Parametricity, type equality, and higher-order polymorphism. (2010).

H. Xi, C. Chen, and G. Chen. 2003. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 224–235.