Deep Induction for Inductive Families (with Applications)[⋆]

Patricia Johann[0000−0002−8075−3904]

Appalachian State University, Boone, NC 28608, USA
johannp@appstate.edu

Abstract. Deep induction provides induction rules for deep data types, i.e., data types that are defined over, or mutually recursively with, (other) such data types. Deep induction is currently defined only for type-indexed types, such as ADTs, nested types, and GADTs. In this paper we show how to extend deep induction from data types with only type indices to data types with term indices as well. Specifically, we extend to inductive families — as found in dependently typed systems such as Agda, Epigram, and Idris — the methodology for deriving sound deep induction rules that was originally developed for nested types and has recently been extended to GADTs.

Keywords: Deep induction · inductive families · proof assistants

1 Introduction

Indexed programming is the practice of programming with indexed types. Perhaps the most common form of indexing indexes types by (other) types. Type-indexed types are found in, e.g., the functional languages Haskell [18] and ML [15]. The essential idea is that a type like List can be indexed by another type that classifies the data it contains. For example, lists of integers, lists of booleans, and lists of lists of data of type t can be modeled by the type-indexed types $\mathsf{List\,Int}$, $\mathsf{List\,Bool}$, and $\mathsf{List\,(List\,t)}$, respectively. More modern programming languages allow types to be indexed not just by types, but also by terms. The essential idea is that a type like List can be indexed by a term that represents, e.g., the length of the list or a proof that it satisfies some property. For instance, lists of length 3 and lists that a proof term p proves are sorted can be modeled by term-indexed types such as $\mathsf{List\,3}$ and $\mathsf{List\,p}$, respectively. (Type- and) term-indexed types are supported as inductive families (IFs) [6] in dependently typed systems such as Agda [1,16], Epigram [13,14], and Idris [8].

Deep induction was introduced in [11] to give induction rules for type-indexed data types that are *deep*, i.e., defined over, or mutually recursively with, (other) such data types. Examples of such data types include, trivially, ordinary

[⋆] This paper is an invited extended version of [10], which appeared at the 2025 Workshop on Logic, Language, Information, and Computation. I am grateful to Ed Morehouse for his contributions to this research, as well as for supporting publication of this extended version of [10] in his absence. This research is supported by National Science Foundation award 2203217.

algebraic data types (ADTs) and nested types; data types, like that of forests from [11] (also called rose trees in [9]), whose recursive occurrences appear below other type constructors; so-called *truly* nested types, like that of bushes from [2] (also called bootstrapped heaps in [17]), whose recursive occurrences can appear below their own type constructors; and generalized algebraic data types (GADTs) [3,24], as found in Haskell and Agda. The aforementioned examples of deep data types are all type indexed, but term-indexed types such as IFs can be deep, too, both in the type-indexed data types that underlie them — i.e., the data types obtained by erasing their term indices — and in the data types (which can also be IFs) that index those underlying data types.

In this paper we show how deep induction can be extended from deep data types that allow only type indexing to those that also allow term indexing. Specifically, we extend to IFs the entire methodology for deriving sound deep induction rules that was developed for nested types in [11] and extended to GADTs in [9]. The structural induction rules currently generated by proof assistants for deep data types induct only over their top-level structures and leave any data internal to that top-level structure untouched. And even those that handle depth in (certain) type-indexed types correctly nevertheless ignore term indices entirely; see the discussion of related work below. Overall, proof assistants currently provide insufficient support for inducting over deep data types. By contrast, deep induction inducts over *all* of the structured data present in a data type. This opens the way for incorporating automatic generation of truly useful induction rules for deep data types, including deep IFs, into state-of-the-art proof assistants.

The remainder of this paper is structured as follows. The rest of this section discusses deep induction for IFs in the context of related work. Section 2 reviews the current state-of-the-art of deep induction for GADTs. These are the most general data types having type indices only. Section 3 illustrates our methodology for extending deep induction from GADTs to *proper* IFs, i.e., IFs that involve term-indexing, and thus are not GADTs. In Section 4 we present our general methodology for deriving deep induction rules for IFs, show that both our methodology and the deep induction rules it delivers generalize those for GADTs, and observe that each concrete instance of a deep induction rule appearing in this paper is derived by instantiating our methodology. Section 5 contains our first application of deep induction for IFs. We show there that if every subterm of a vector-indexed sequence constructed from primitive such sequences indexed by vectors of even length all of whose elements are odd, then the overall sequence is itself indexed by such a vector. As a second application, Section 6 revisits type inference for the small typed lambda calculus performed in [9], but this time the data type defining the calculus is viewed as an IF rather than as a GADT. This allows us to give two deep induction rules for the calculus and to highlight some optimizations of our methodology that allow us to compare the rules given here both with one another and with that from [9]. Section 7 concludes and offers directions for future work. Our Agda implementation containing all of the deep induction rules appearing in this paper (and proof terms that witness their soundness) is available at `https://cs.appstate.edu/johannp/`.

Related Work Deep induction was introduced for nested types in [11] and extended to GADTs in [9]. The methodology for deriving deep induction rules developed in this paper further extends that in [9] to IFs. The relationship between our results and those of [9,11] are discussed in detail throughout this paper.

To the best of our knowledge, all other work on generating induction rules for IFs is either restricted to structural induction or fails to adequately account for depth in term indices. Among the works in the first camp are [4,5,6,19]. The works [22] and [23], on the other hand, derive induction rules that are deep for nested types and some IF's whose underlying data types and indexing types are containers. But since they generate only trivial predicates for types such as the natural numbers, the derived induction rule for, e.g., the data type of vectors (length-indexed lists), effectively ignores vectors' natural number term indices and is therefore reduced to the induction rule for their underlying lists.

2 The State-of-the-Art in Deep Induction

To illustrate the difference between structural induction and deep induction, consider the following data type of lists:[1]

```
data List (a : Set) : Set where
  [] : List a
  _::_  :  a → List a → List a
```

Since the following standard structural induction rule for lists uses a predicate P on entire lists, it essentially ignores the data inside an element of type List a:

$$\begin{array}{l}(\mathsf{P} : \mathsf{List}\,\mathsf{a} \to \mathsf{Set}) \to \\ \quad \mathsf{P}\,[\,] \to \\ \quad ((\mathsf{x} : \mathsf{a}) \to (\mathsf{xs} : \mathsf{List}\,\mathsf{a}) \to \mathsf{P}\,\mathsf{xs} \to \mathsf{P}\,(\mathsf{x} :: \mathsf{xs})) \to \\ \quad (\mathsf{xs} : \mathsf{List}\,\mathsf{a}) \to \mathsf{P}\,\mathsf{xs}\end{array} \qquad (1)$$

By contrast, the deep induction rule for lists traverses not just the outer list structure with P, but also each element of that list with a custom predicate Q:

$$\begin{array}{l}(\mathsf{P} : \mathsf{List}\,\mathsf{a} \to \mathsf{Set}) \to (\mathsf{Q} : \mathsf{a} \to \mathsf{Set}) \to \\ \quad \mathsf{P}\,[\,] \to \\ \quad ((\mathsf{x} : \mathsf{a}) \to (\mathsf{xs} : \mathsf{List}\,\mathsf{a}) \to \mathsf{Q}\,\mathsf{x} \to \mathsf{P}\,\mathsf{xs} \to \mathsf{P}\,(\mathsf{x} :: \mathsf{xs})) \to \\ \quad (\mathsf{xs} : \mathsf{List}\,\mathsf{a}) \to \mathsf{List}^{\wedge}\,\mathsf{Q}\,\mathsf{xs} \to \mathsf{P}\,\mathsf{xs}\end{array} \qquad (2)$$

Here, the lifting $\mathsf{List}^{\wedge}$ lifts its argument predicate Q on data of type a to a predicate on data of type List a by asserting that $\mathsf{List}^{\wedge}\,\mathsf{Q}$ holds of xs : List a

[1] We use Agda syntax for concreteness of exposition in this paper. Specifically, to avoid repetition our development uses Agda's facility for generalizing declared variables whose types are easily inferred. Thus, throughout the paper, implicitly bound occurrences of a, b, c, and d have type Set and implicitly bound occurrences of m and n have type Nat. We emphasize, however, that our methodology is not Agda-specific.

precisely when Q holds for every element of xs. It can be defined in Agda by:

$$\begin{aligned}
&\mathsf{List}^\wedge : (\mathsf{a} \to \mathsf{Set}) \to \mathsf{List\,a} \to \mathsf{Set}\\
&\mathsf{List}^\wedge\,\mathsf{Q}\,[\,] \;=\; \top\\
&\mathsf{List}^\wedge\,\mathsf{Q}\,(\mathsf{x} :: \mathsf{xs}) \;=\; \mathsf{Q\,x} \times \mathsf{List}^\wedge\,\mathsf{Q\,xs}
\end{aligned}$$

The structural induction rule for lists can be recovered by taking the custom predicate Q in their deep induction rule to be the constantly $\top$-valued predicate.

Just as structural induction can be extended to nested types, so can deep induction. Consider, for example, the following type of perfect trees from [2]:

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

Perfect trees can be thought of as lists constrained to have lengths that are powers of 2. Since the constructor pnode of PTree uses data of type $\mathsf{PTree\,(a \times a)}$ to construct data of type $\mathsf{PTree\,a}$, the instances of PTree at various indices cannot be defined independently. Instead, the entire inductive family of types must be defined at once. This intertwinedness of the instances of nested types is reflected in their both their structural and their deep induction rules, which, as explained in [11], must necessarily involve polymorphic predicates rather than the monomorphic predicates that suffice for lists and other ADTs. Indeed, the structural induction rule for perfect trees is given by:

$$\begin{aligned}
&(\mathsf{P} : \{\mathsf{a : Set}\} \to \mathsf{PTree\,a} \to \mathsf{Set}) \to\\
&\quad (\{\mathsf{a : Set}\} \to (\mathsf{x : a}) \to \mathsf{P\,(pleaf\,x)}) \to\\
&\quad (\{\mathsf{a : Set}\} \to (\mathsf{xs : PTree\,(a \times a)}) \to \mathsf{P\,xs} \to \mathsf{P\,(pnode\,xs)}) \to\\
&\quad \{\mathsf{a : Set}\} \to (\mathsf{xs : PTree\,a}) \to \mathsf{P\,xs}
\end{aligned}$$

The deep induction rule for perfect trees similarly uses a polymorphic predicate. This predicate is parameterized over a custom predicate on the data in the perfect tree. The deep induction rule for perfect trees is thus given by:

$$\begin{aligned}
&(\mathsf{P} : \{\mathsf{a : Set}\} \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{PTree\,a} \to \mathsf{Set}) \to\\
&\quad (\{\mathsf{a : Set}\} \to (\mathsf{Q : a \to Set}) \to (\mathsf{x : a}) \to \mathsf{Q\,x} \to \mathsf{P\,Q\,(pleaf\,x)}) \to\\
&\quad (\{\mathsf{a : Set}\} \to (\mathsf{Q : a \to Set}) \to (\mathsf{xs : PTree\,(a \times a)}) \to \mathsf{P}\,(\times^\wedge\,\mathsf{Q\,Q})\,\mathsf{xs} \to \mathsf{P\,Q\,(pnode\,xs)}) \to\\
&\quad (\mathsf{Q : a \to Set}) \to (\mathsf{xs : PTree\,a}) \to \mathsf{PTree}^\wedge\,\mathsf{Q\,xs} \to \mathsf{P\,Q\,xs}
\end{aligned} \tag{3}$$

where the lifting $\times^\wedge : (\mathsf{a} \to \mathsf{Set}) \to (\mathsf{b} \to \mathsf{Set}) \to \mathsf{a} \times \mathsf{b} \to \mathsf{Set}$ is given by $\times^\wedge\,\mathsf{Q_a\,Q_b}\,(\mathsf{x},\mathsf{y}) = \mathsf{Q_a\,x} \times \mathsf{Q_b\,y}$, and the lifting $\mathsf{PTree}^\wedge$ is given by:

$$\begin{aligned}
&\mathsf{PTree}^\wedge : (\mathsf{a} \to \mathsf{Set}) \to \mathsf{PTree\,a} \to \mathsf{Set}\\
&\mathsf{PTree}^\wedge\,\mathsf{Q\,(pleaf\,x)} \;=\; \mathsf{Q\,x}\\
&\mathsf{PTree}^\wedge\,\mathsf{Q\,(pnode\,xs)} \;=\; \mathsf{PTree}^\wedge\,(\times^\wedge\,\mathsf{Q\,Q})\,\mathsf{xs}
\end{aligned}$$

The structural induction rule for perfect trees is obtained by taking Q in (3) to be the constantly $\top$-valued predicate. Similar instantiation shows that the deep induction rule for *any* nested type (indeed, any IF considered in this paper) syntactically generalizes its structural induction rule. The more general syntax of a data type's deep induction rule often lends itself better to proving properties of the data type that involve custom predicates than does that of its structural induction rule. The two rules have exactly the same computational power, however.

On the other hand, deep induction actually *is* central to generating genuinely useful induction rules for some deep data types. For example, the structural induction rule generated by Rocq for the deep data type

$$\begin{array}{l}\mathsf{data\ Forest\ (a : Set) : Set\ where}\\ \quad \mathsf{fempty} : \mathsf{Forest\ a}\\ \quad \mathsf{fnode}\ \ : \mathsf{a} \to \mathsf{List\ (Forest\ a)} \to \mathsf{Forest\ a}\end{array}$$

of forests is

$$\begin{array}{l}(\mathsf{P} : \mathsf{Forest\ a} \to \mathsf{Set}) \to\\ \ \mathsf{P\ fempty} \to\\ \ ((\mathsf{x} : \mathsf{a}) \to (\mathsf{xss} : \mathsf{List\ (Forest\ a)}) \to \mathsf{P\ (fnode\ x\ xss)}) \to\\ \ (\mathsf{xs} : \mathsf{Forest\ a}) \to \mathsf{P\ xs}\end{array} \tag{4}$$

But this is neither the intuitively expected induction rule for them, nor is it expressive enough to prove even basic properties of forests that ought to be amenable to inductive proof. Indeed, to prove that a property P holds for a forest $\mathsf{fnode\ x\ xss}$ using (4), we must do so assuming only that xss is a list of forests and without recourse to any induction hypotheses for the forests in xss. By contrast, the deep induction rule for forests from [11] is:

$$\begin{array}{l}(\mathsf{P} : \mathsf{Forest\ a} \to \mathsf{Set}) \to (\mathsf{Q} : \mathsf{a} \to \mathsf{Set}) \to\\ \ \mathsf{P\ fempty} \to\\ \ ((\mathsf{x} : \mathsf{a}) \to (\mathsf{xss} : \mathsf{List\ (Forest\ a)}) \to \mathsf{Q\ x} \to \mathsf{List}^\wedge\ \mathsf{P\ xss} \to \mathsf{P\ (fnode\ x\ xss)}) \to\\ \ (\mathsf{xs} : \mathsf{Forest\ a}) \to \mathsf{Forest}^\wedge\ \mathsf{Q\ xs} \to \mathsf{P\ xs}\end{array} \tag{5}$$

where the lifting

$$\begin{array}{l}\mathsf{Forest}^\wedge : (\mathsf{a} \to \mathsf{Set}) \to \mathsf{Forest\ a} \to \mathsf{Set}\\ \mathsf{Forest}^\wedge\ \mathsf{Q\ fempty}\ =\ \top\\ \mathsf{Forest}^\wedge\ \mathsf{Q\ (fnode\ x\ xss)}\ =\ \mathsf{Q\ x} \times \mathsf{List}^\wedge\ (\mathsf{Forest}^\wedge\ \mathsf{Q})\ \mathsf{xss}\end{array}$$

provides the missing induction hypotheses for the forests in xss. The deep induction rule (5) is thus of much more use. As expected, the special case when Q is the constantly $\top$-valued predicate again recovers the structural induction rule (4).

In [11], deep induction was also shown to be the key to defining *structural* induction rules for truly nested types. The canonical example of such a type is the following truly nested type Bush of bushes:

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

The deep induction rule for any nested type must account for the potentially different instances at which it is instantiated in its definition; for a truly nested type some of these may be itself. As usual, this is done by parameterizing the predicate P to be proved for all elements of the nested type over custom predicates on the types of data they contain and then suitably instantiating those parameterizing predicates' type arguments at each application site. This gives the following deep induction rule for bushes:

$$\begin{array}{l}(\mathsf{P} : \{\mathsf{a} : \mathsf{Set}\} \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{Bush}\,\mathsf{a} \to \mathsf{Set}) \to \\ \quad (\{\mathsf{a} : \mathsf{Set}\} \to (\mathsf{Q} : \mathsf{a} \to \mathsf{Set}) \to \mathsf{P}\,\mathsf{Q}\,\mathsf{bnil}) \to \\ \quad (\{\mathsf{a} : \mathsf{Set}\} \to (\mathsf{x} : \mathsf{a}) \to (\mathsf{xss} : \mathsf{Bush}\,(\mathsf{Bush}\,\mathsf{a})) \to (\mathsf{Q} : \mathsf{a} \to \mathsf{Set}) \to \\ \qquad \mathsf{Q}\,\mathsf{x} \to \mathsf{P}\,(\mathsf{P}\,\mathsf{Q})\,\mathsf{xss} \to \mathsf{P}\,\mathsf{Q}\,(\mathsf{bcons}\,\mathsf{x}\,\mathsf{xss})) \to \\ \quad (\mathsf{Q} : \mathsf{a} \to \mathsf{Set}) \to (\mathsf{xs} : \mathsf{Bush}\,\mathsf{a}) \to \mathsf{Bush}^{\wedge}\,\mathsf{Q}\,\mathsf{xs} \to \mathsf{P}\,\mathsf{Q}\,\mathsf{xs}\end{array} \qquad (6)$$

As for lists, the role of the predicate lifting $\mathsf{Bush}^{\wedge}$ for bushes is two-fold. It must apply its argument predicate Q to all new data of the primitive type a, as well as propagate Q through the structure of all of the recursive subdata, in its argument bush. This requires applying Q to the datum $\mathsf{x} : \mathsf{a}$ directly, and instantiating the predicate $\mathsf{Bush}^{\wedge}$ on bushes to the type $\mathsf{Bush}\,\mathsf{a}$ itself when handling the recursive argument $\mathsf{xss} : \mathsf{Bush}\,(\mathsf{Bush}\,\mathsf{a})$ in the clause for bcons. The lifting $\mathsf{Bush}^{\wedge}$ is thus:

$$\begin{array}{l}\mathsf{Bush}^{\wedge} : (\mathsf{a} \to \mathsf{Set}) \to \mathsf{Bush}\,\mathsf{a} \to \mathsf{Set} \\ \mathsf{Bush}^{\wedge}\,\mathsf{Q}\,\mathsf{bnil} \;=\; \top \\ \mathsf{Bush}^{\wedge}\,\mathsf{Q}\,(\mathsf{bcons}\,\mathsf{x}\,\mathsf{xss}) = \mathsf{Q}\,\mathsf{x} \times \mathsf{Bush}^{\wedge}\,(\mathsf{Bush}^{\wedge}\,\mathsf{Q})\,\mathsf{xss}\end{array}$$

As for ADTs, the structural induction rule for any nested type — including truly nested types like Bush — can be recovered from its deep induction rule by taking its parameterizing predicates to P to be constantly $\top$-valued predicates on their underlying types. In [11], taking Q in (6) to be the constantly $\top$-valued predicate yielded the first-ever *structural* induction rule for bushes — a full 20 years after the Bush data type was introduced!

Deep Induction for GADTs

Deep induction was further extended to GADTs in [9]. GADTs generalize nested types by allowing data constructors to not only take data at arbitrary instances of the GADT *as arguments*, but also to return such data *as results* as well. A simple example of a GADT is the following data type Seq of sequences:[2]

$$\begin{array}{l}\mathsf{data}\,\mathsf{Seq} : \mathsf{Set} \to \mathsf{Set}\,\mathsf{where} \\ \quad \mathsf{inj} :\; \mathsf{a} \to \mathsf{Seq}\,\mathsf{a} \\ \quad \mathsf{pair} :\; \mathsf{Seq}\,\mathsf{b} \to \mathsf{Seq}\,\mathsf{c} \to \mathsf{Seq}\,(\mathsf{b} \times \mathsf{c})\end{array}$$

Note that Seq's data constructor pair constructs only sequences of data whose types are pair-structured, rather than sequences of any type, as does its data

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

constructor inj. It is fruitful to capture this kind of non-uniformity in the return types of GADTs' data constructors via their so-called *Henry Ford encodings* [3,7,12,20,21]. These encodings use the following equality type from Agda's standard library to, in essence, turn GADTs into nested types:

$$\begin{array}{l} \mathsf{data}\ \_\equiv\_\ (\mathsf{x} : \mathsf{a}) : \mathsf{a} \to \mathsf{Set}\,\mathsf{where} \\ \quad \mathsf{refl} : \mathsf{x} \equiv \mathsf{x} \end{array}$$

The Henry Ford encoding for Seq, for example, replaces the requirement that the data constructor pair produce data at an instance of Seq that is a product type with the requirement that pair produce data at an instance of Seq that is *equal* to a product type. It is:

$$\begin{array}{l} \mathsf{data}\,\mathsf{Seq}\,(\mathsf{a} : \mathsf{Set}) : \mathsf{Set}\ \mathsf{where} \\ \quad \mathsf{inj} :\ \mathsf{a} \to \mathsf{Seq}\,\mathsf{a} \\ \quad \mathsf{pair} :\ (\mathsf{b} \times \mathsf{c}) \equiv \mathsf{a} \to \mathsf{Seq}\,\mathsf{b} \to \mathsf{Seq}\,\mathsf{c} \to \mathsf{Seq}\,\mathsf{a} \end{array} \tag{7}$$

Henry Ford encodings for other GADTs are obtained similarly.

Deep induction rules for GADTs are then defined in [9] using the lifting

$$\begin{array}{l} \equiv^{\wedge} : (\mathsf{a} \to \mathsf{Set}) \to (\mathsf{b} \to \mathsf{Set}) \to \mathsf{a} \equiv \mathsf{b} \to \mathsf{Set} \\ \equiv^{\wedge}\ \mathsf{Q}\,\mathsf{Q}'\,\mathsf{refl}\ =\ (\mathsf{x} : \mathsf{a}) \to \mathsf{Q}\,\mathsf{x} \equiv \mathsf{Q}'\,\mathsf{x} \end{array} \tag{8}$$

for equality types, together with existentially quantified predicates[3] and the original methodology for nested types. Both the predicate liftings and deep induction rules for GADTs are defined via their Henry Ford encodings. The lifting for Seq is:

$$\begin{array}{l} \mathsf{Seq}^{\wedge} : (\mathsf{a} \to \mathsf{Set}) \to \mathsf{Seq}\,\mathsf{a} \to \mathsf{Set} \\ \mathsf{Seq}^{\wedge}\,\mathsf{Q_a}\,(\mathsf{inj}\,\mathsf{x}) = \mathsf{Q_a}\,\mathsf{x} \\ \mathsf{Seq}^{\wedge}\,\mathsf{Q_a}\,(\mathsf{pair}\,\mathsf{p}\,\mathsf{s_b}\,\mathsf{s_c}) = \exists[\mathsf{Q_b}]\exists[\mathsf{Q_c}]\ \equiv^{\wedge}(\times^{\wedge}\,\mathsf{Q_b}\,\mathsf{Q_c})\,\mathsf{Q_a}\,\mathsf{p} \times \mathsf{Seq}^{\wedge}\,\mathsf{Q_b}\,\mathsf{s_b} \times \mathsf{Seq}^{\wedge}\,\mathsf{Q_c}\,\mathsf{s_c} \end{array} \tag{9}$$

It introduces new predicates on the new types introduced by the Henry Ford encoding, and then enforces the necessary connections between them and the predicates on the types present in the original data type declaration. For pair, e.g., it introduces predicates $\mathsf{Q_b}$ and $\mathsf{Q_c}$ on the types b and c introduced by Seq's Henry Ford encoding, and then ensures that $\mathsf{Q_b} \times \mathsf{Q_c}$ and $\mathsf{Q_a}$ are equal. Otherwise it simply performs the usual two tasks of liftings, namely (i) ensuring that any new primitive data (i.e., data of primitive type) used to construct a data element satisfy their predicates, and (ii) ensuring that all of that data element's recursive subdata also satisfy appropriate liftings of those predicates. Given the lifting $\mathsf{Seq}^{\wedge}$, the deep induction rule for Seq is:

$$\begin{array}{l} (\mathsf{P} : \{\mathsf{a} : \mathsf{Set}\} \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{Seq}\,\mathsf{a} \to \mathsf{Set}) \to \\ \quad (\{\mathsf{a} : \mathsf{Set}\} \to (\mathsf{x} : \mathsf{a}) \to (\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to \mathsf{Q_a}\,\mathsf{x} \to \mathsf{P}\,\mathsf{Q_a}\,(\mathsf{inj}\,\mathsf{x})) \to \\ \quad (\{\mathsf{a}\,\mathsf{b}\,\mathsf{c} : \mathsf{Set}\} \to (\mathsf{p} : (\mathsf{b} \times \mathsf{c}) \equiv \mathsf{a}) \to (\mathsf{s_b} : \mathsf{Seq}\,\mathsf{b}) \to (\mathsf{s_c} : \mathsf{Seq}\,\mathsf{c}) \to (\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to \\ \qquad (\mathsf{Q_b} : \mathsf{b} \to \mathsf{Set}) \to (\mathsf{Q_c} : \mathsf{c} \to \mathsf{Set}) \to \mathsf{P}\,\mathsf{Q_b}\,\mathsf{s_b} \to \mathsf{P}\,\mathsf{Q_c}\,\mathsf{s_c} \to \mathsf{P}\,\mathsf{Q_a}\,(\mathsf{pair}\,\mathsf{p}\,\mathsf{s_b}\,\mathsf{s_c})) \to \\ \quad (\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to (\mathsf{s} : \mathsf{Seq}\,\mathsf{a}) \to \mathsf{Seq}^{\wedge}\,\mathsf{Q_a}\,\mathsf{s} \to \mathsf{P}\,\mathsf{Q_a}\,\mathsf{s} \end{array} \tag{10}$$

[3] The suggestive notation $\exists[\mathsf{x}]\,\mathsf{Q}\,\mathsf{x}$ is syntactic sugar for the type of dependent pairs (x, b) with $\mathsf{x} : \mathsf{a}$, $\mathsf{b} : \mathsf{Q}\,\mathsf{x}$, and $\mathsf{Q} : \mathsf{a} \to \mathsf{Set}$.

This Paper

In this paper we extend deep induction from GADTs to IFs. Unlike GADTs, whose indices are always *types*, IFs also allow indices that are *terms*. The predicate in the deep induction rule for an IF must therefore take as input predicates not only on its type indices but also on the types of its term indices. To obtain an IF's deep induction rule, all of these predicates must be appropriately propagated to all of the primitive data in the IF's data elements. Properly accounting for conditions under which term indices must satisfy their predicates is thus the central challenge in extending deep induction from GADTs to proper IFs.

In this paper we meet this challenge precisely. Moreover, we do so in such a way that the deep induction rules for IFs we develop specialize to the rules of [9] for those IFs that can be seen as GADTs (and thus to the rules of [11] for those IFs that can be seen as nested types and ADTs). We consider such specialization to be a minimal success criterion for our deep induction rules for IFs since it ensures that our methodology for producing them is a conservative extension of all those that have come before. Other important success criteria are that our deep induction rules for IFs specialize to the structural induction rules of [6], and that they properly extend the deep induction rules for IFs in [23], which are deep only on their type indices, to be deep on their term indices as well. The former is seen, as usual, by taking the predicates (on *both* the type and term indices) parameterizing the overall predicate to be proved for elements of the IF to be constantly $\top$-valued. This is a second success criterion because the structural induction rule for a given data type should always be a special case of its deep induction rule. The latter is seen by specializing the parameterizing predicates on IFs' *term* indices to constantly $\top$-valued predicates. This is a third success criterion because it guarantees that our deep induction rules for IFs are more general than those found in other conjectured approaches.

Overall, then, this paper gives the first-ever deep induction rules for proper IFs and demonstrates their soundness. But it actually delivers far more: it gives a *general methodology* for deriving sound deep induction rules for IFs that can be instantiated to particular IFs of interest. This methodology can serve as a basis for conservatively extending proof assistants' automatic generation of structural induction rules for IFs to automatic generation of deep induction rules for them.

3 Predicates for Term Indices: The Key Idea

The key to deriving deep induction rules for type-indexed-only data types is to define predicate liftings for them that perform the tasks (i) and (ii) identified just before (10) above. We now show how to generalize liftings for GADTs — i.e., type-indexing-only IFs — to predicate liftings suitable to IFs that also allow *term* indexing. To illustrate our approach, consider the proper IF Vec defining the data type of vectors (or length-indexed lists) over a type a taken (essentially) from Agda's standard library:[4]

$$\begin{array}{ll} \mathsf{data\,Vec\,(a : Set) : Nat \to Set\,where} & \mathsf{data\,Nat : Set\,where} \\ \quad \mathsf{[\,] : \ Vec\,a\,zero} & \quad \mathsf{zero : Nat} \\ \quad \mathsf{\_::\_ \ : a \to Vec\,a\,n \to Vec\,a\,(suc\,n)} & \quad \mathsf{suc : \ Nat \to Nat} \end{array} \tag{11}$$

The data type underlying Vec — i.e., the data type obtained from Vec by erasing its term indices — is the List ADT. Its term indices are of type Nat, and thus do not have interesting traversable structure. Although Vec is a particularly simple proper IF, it cleanly isolates the process of tracking term indices in deriving deep induction rules for IFs. The same principled, uniform methodology we illustrate here delivers deep induction rules for IFs with *both* more complex underlying data types, *and* more complex index types ranging all the way from built-in ones to IFs themselves. For example, the IF of Fin-indexed-sequences in Figure 1, which has the GADT Seq as its underlying data type and the IF Fin of finite sets from Agda's standard library as its index type, has both a maximally general underlying data type and a maximally general index type. Our predicate lifting and deep induction rule for it are given in Figure 1. They are derived using the methodology introduced here using Vec and described more fully in the next section.

Since [ ] constructs vectors of length zero, any useful deep induction rule for vectors must ensure that zero satisfies the predicate on their natural number indices. Moreover, since a vector of length suc n is made from a vector of length n and a new data element of the vector's parameter type, such a rule must also ensure that suc n satisfies this predicate whenever n does. Similar implications must obtain between the term indices of other IFs, so we add to tasks (i) and (ii) identified above for predicate liftings the task of also (iii) ensuring that the[5] term index of every data element constructed using a data constructor of an IF satisfies the predicate on the type of the IF's indices provided the term indices of the element's recursive subdata do. For Vec, this results in the predicate lifting

$$\begin{array}{l} \mathsf{Vec^\wedge : (a \to Set) \to (Nat \to Set) \to Vec\,a\,n \to Set} \\ \mathsf{Vec^\wedge \,\{n = zero\}\, Q_a\, Q_n\, [\,] \ = \ Q_n\, zero} \\ \mathsf{Vec^\wedge \,\{n = suc\,m\}\, Q_a\, Q_n\, (x :: xs) \ = \ Q_a\, x \times Vec^\wedge\, Q_a\, Q_n\, xs \times (Q_n m \to Q_n (suc\, m))} \end{array} \tag{12}$$

and the following deep induction rule for Vec associated to this lifting:

$$\begin{array}{l} \mathsf{(Q_a : a \to Set) \to (Q_n : Nat \to Set) \to (P : \{n : Nat\} \to Vec\,a\,n \to Set) \to} \\ \quad \mathsf{(Q_n\, zero \to P\, [\,]) \to} \\ \quad \mathsf{(\{n : Nat\} \to (x : a) \to (xs : Vec\,a\,n) \to Q_a\, x \to P\, xs \to Q_n\, (suc\, n) \to P\, (x :: xs)) \to} \\ \quad \mathsf{(xs : Vec\,a\,n) \to Vec^\wedge\, Q_a\, Q_n\, xs \to P\, xs} \end{array} \tag{13}$$

[4] Although the names of List's data constructors are also used for those of Vec, no confusion should arise in our exposition since the data type is at play will always be clear from context.

[5] For ease of exposition, we assume throughout that IFs have exactly one term index. The generalization to more than one term index is straightforward, if slightly tedious.

Of course, just as the data in an IF's underlying data type can be structured, so can the data in elements of its indexing data type be structured. Propagation of predicates on both the primitive data in an IF's underlying data type and on the primitive data in its indexing IF's elements are handled in the standard way. This is explicated in [9,11] and also recalled above. The presence or absence of structure in the IF's underlying data type or term indices in no way affects how satisfaction of the predicates on term indices must be preserved in the clauses of the IF's lifting for its data constructors. Indeed, propagation of predicates on primitive data through all of the structure in an IF's underlying data type and indices on the one hand, and preservation of predicate satisfaction for all of that IF's term indices (and, implicitly, preservation of well-formedness of its type indices) on the other, are orthogonal concerns when constructing its deep induction rule.

4 Deriving Liftings and Deep Induction Rules for IFs

That satisfaction of the predicates on a proper IF's term indices must be appropriately preserved to derive deep induction rules for these data types is the key observation of this paper. Detailing and justifying the uniform and principled manner in which this is done is its main technical contribution. This results in a general methodology for defining liftings and deep induction rules for proper IFs that generalize those from [9,11] for data types that are type-indexed only.

Since our methodology will handle an IF's type indices as they are handled in [9,11], the only new thing we need to account for is satisfaction of predicates on its term indices. In the most general situation we consider, an IF can have a GADT as its underlying indexed data type and, recursively, an IF as its indexing data type. In this paper we consider IFs whose underlying GADTs, and whose indexing IFs' underlying GADTs, are of the same form as those in [9], namely:

$$\begin{array}{l} \mathsf{data}\ \mathsf{G} : \mathsf{Set}^{\alpha} \to \mathsf{Set}\ \mathsf{where} \\ \quad \mathsf{c}\ :\ \forall\{\overline{\mathsf{B} : \mathsf{Set}}\} \to \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}} \to \mathsf{G}(\overline{\mathsf{K}\,\overline{\mathsf{B}}}) \end{array} \tag{14}$$

For brevity and clarity we indicate only one data constructor c in (14), even though a GADT can, in general, have any finite number of data constructors, each with a type of the same form as c's above. In (14), F and each K in $\overline{\mathsf{K}}$ are type constructors with signatures $(\mathsf{Set}^{\alpha} \to \mathsf{Set}) \to \mathsf{Set}^{\beta} \to \mathsf{Set}$ and $\mathsf{Set}^{\beta} \to \mathsf{Set}$, respectively. If T is a type constructor with signature $\mathsf{Set}^{\gamma} \to \mathsf{Set}$ then the overline notation denotes a finite list whose length is γ. The number of type constructors in $\overline{\mathsf{K}}$ (resp., $\overline{\mathsf{B}}$) is thus α (resp., β). In addition, the type constructor F must be constructed inductively according to the following grammar, also from [9]:

$$\begin{array}{rl} \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}} := & \mathsf{F}_1\,\mathsf{G}\,\overline{\mathsf{B}} \times \mathsf{F}_2\,\mathsf{G}\,\overline{\mathsf{B}} \mid \mathsf{F}_1\,\mathsf{G}\,\overline{\mathsf{B}} + \mathsf{F}_2\,\mathsf{G}\,\overline{\mathsf{B}} \mid \mathsf{F}_1\,\overline{\mathsf{B}} \to \mathsf{F}_2\,\mathsf{G}\,\overline{\mathsf{B}} \\ & \mid \mathsf{G}\,(\overline{\mathsf{F}_1\,\overline{\mathsf{B}}}) \mid \mathsf{H}\,\overline{\mathsf{B}} \mid \mathsf{H}\,(\overline{\mathsf{F}_1\,\mathsf{G}\,\overline{\mathsf{B}}}) \end{array}$$

As in [9], this grammar is subject to the following restrictions. In the third clause the type constructor F_1 does not use G, so G is omitted from the call to F_1. Similarly, in the fourth clause, none of the α-many type constructors in $\overline{\mathsf{F}_1}$ use G. This prevents nesting, which would make it impossible to give an induction rule for G; see Section 6 of [9] for details. In the fifth and sixth clauses, $\mathsf{H} : \mathsf{Set}^{\gamma} \to \mathsf{Set}$ is the syntactic reflection of some functor, and thus has an associated map function. Note that the fifth clause subsumes the cases in which $\mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}}$ is a closed type or one of the B_i, and that H can be the data type constructor for any (truly) nested type.

Focusing on the same class of GADTs as in [9] guarantees that the techniques of that paper apply to the type indices both of the GADT underlying an IF and of the GADT underlying that IF's indexing IF. We thus need only additionally ensure that each of an IF's data constructors appropriately preserves satisfaction of the predicate on the type of the IF's term indices in order to arrive at a conservative extension to proper IFs of the techniques in [9,11] for deriving deep induction rules for GADTs, and thus at a uniform methodology for deriving deep induction rules for such IFs.

Given an IF D, predicates on its type indices and a predicate on the type of its term indices, the lifting $\mathsf{D}^\wedge$ for D of these predicates includes one clause for each of D's data constructors. The clause for D's data constructor c is constructed via the steps described below. We illustrate each step using the constructors finj and fpair for the IF FSeq from Figure 1 whose underlying GADT Seq has one type parameter a and is term-indexed by elements of the IF Fin. As in the case of FSeq, the GADT underlying the IF of interest may first need to be converted into its Henry Ford encoding to accommodate the predicates on its type indices; see [9] for details. The following steps can then be applied directly to that converted IF, exactly as illustrated below. The liftings from [9] can be understood anew as liftings that accomplish for GADTs all three of the tasks identified below (the last trivially), so our methodology for IFs subsumes that for GADTs in [9] (and hence that for nested types in [11]), in which no term indices are present.

The clause of $\mathsf{D}^\wedge$ for a data constructor c of an IF D first introduces existentially quantified predicates on any types appearing in the clause of the Henry Ford encoding for c other than its type indices, and then enforces the necessary connections between them and the predicates on D's type indices. This is exactly as in (9). The remainder of the clause is constructed as follows:

1. Check that all non-recursive data, other than term indices of recursive subdata and the term index of the constructed term, that are used by c to construct elements of D satisfy the liftings for their types of the predicates on the types appearing in the clause for c. In the definition of FSeq, e.g., the data constructor finj requires the non-recursive datum $\mathsf{x} : \mathsf{a}$ and $\mathsf{i} : \mathsf{Fin}\,\mathsf{n}$. But the latter is the index of the constructed term, so the clause of $\mathsf{FSeq}^\wedge$ for finj requires only the check $\mathsf{Q}_\mathsf{a}\,\mathsf{x}$ here. Similarly, the data constructor fpair requires a non-recursive non-term-index argument $\mathsf{p} : (\mathsf{b} \times \mathsf{c}) \equiv \mathsf{a}$, so the clause of $\mathsf{FSeq}^\wedge$ for fpair requires a corresponding term $\equiv^\wedge (\times^\wedge\,\mathsf{Q}_\mathsf{b}\,\mathsf{Q}_\mathsf{c})\,\mathsf{Q}_\mathsf{a}\,\mathsf{p}$.
2. Check that all recursive subdata of the element of D that c constructs satisfy the lifting being defined of the predicates on the types appearing in the clause for c and the type of its term indices. In the definition of FSeq, e.g., finj takes no recursive subdata as arguments, so this step contributes no terms to the clause of $\mathsf{FSeq}^\wedge$ for finj. On the other hand, fpair requires

recursive arguments $\mathsf{sbi} : \mathsf{FSeq\,b\,i}$ and $\mathsf{scj} : \mathsf{FSeq\,c\,j}$, so the clause of $\mathsf{FSeq}^\wedge$ for fpair requires corresponding terms $\mathsf{FSeq}^\wedge\,\mathsf{Q_b}\,\mathsf{Q_f}\,\mathsf{sbi}$ and $\mathsf{FSeq}^\wedge\,\mathsf{Q_c}\,\mathsf{Q_f}\,\mathsf{scj}$.

3. Check that the term index of the element of D that c constructs satisfies the predicate on its type provided the term indices of the element's recursive subdata do. In the definition of FSeq, e.g., finj constructs an element with term index i from no recursive subdata, so this step contributes the term (i.e., the satisfaction preservation condition with no hypotheses) $\mathsf{Q_f\,i}$ to the clause of $\mathsf{FSeq}^\wedge$ for finj. Similarly, fpair constructs an element with term index $\mathsf{i} +_\mathsf{f} \mathsf{j}$ from recursive subdata with indices i and j, where $+_\mathsf{f}$ is the addition function for elements of Fin defined in Agda's standard library. The clause of $\mathsf{FSeq}^\wedge$ for fpair thus requires the corresponding satisfaction preservation condition $\mathsf{Q_f\,i} \to \mathsf{Q_f\,j} \to \mathsf{Q_f}\,(\mathsf{i} +_\mathsf{f} \mathsf{j})$.

Taken altogether, this gives the clauses of $\mathsf{D}^\wedge$ for c. The clause of $\mathsf{FSeq}^\wedge$ for finj is:

$$\begin{array}{rcll} \mathsf{FSeq}^\wedge\,\mathsf{Q_a}\,\mathsf{Q_f}\,(\mathsf{finj\,x\,i}) & = & \mathsf{Q_a\,x} \times & \text{(Step 1)} \\ & & \mathsf{Q_f\,i} & \text{(Step 3)} \end{array}$$

and that for fpair, e.g., is:

$$\begin{array}{rcll} \mathsf{FSeq}^\wedge\,\mathsf{Q_a}\,\mathsf{Q_f}\,(\mathsf{fpair\,p}\,\{\mathsf{i}\}\,\{\mathsf{j}\}\,\mathsf{sbi\,scj}) & = & \exists[\mathsf{Q_b}]\,\exists[\mathsf{Q_c}] & \\ & & \equiv^\wedge\,(\times^\wedge\,\mathsf{Q_b}\,\mathsf{Q_c})\,\mathsf{Q_a}\,\mathsf{p} \times & \text{(Step 1)} \\ & & \mathsf{FSeq}^\wedge\,\mathsf{Q_b}\,\mathsf{Q_f}\,\mathsf{sbi} \times \mathsf{FSeq}^\wedge\mathsf{Q_c}\,\mathsf{Q_f}\,\mathsf{scj} \times & \text{(Step 2)} \\ & & (\mathsf{Q_f\,i} \to \mathsf{Q_f\,j} \to \mathsf{Q_f}\,(\mathsf{i} +_\mathsf{f} \mathsf{j})) & \text{(Step 3)} \end{array}$$

Once we have the lifting for D, its deep induction rule is derived as follows:

1. The first input to D's deep induction rule is a predicate P to be shown to hold for all elements of D. It must be parameterized by all of the data needed to construct the instance of D that P is a predicate on, as well as by predicates on D's type indices and a predicate on the type of D's term indices. For example, the first input to the deep induction rule for FSeq is a predicate

$$\begin{array}{c} \mathsf{P} : \{\mathsf{a} : \mathsf{Set}\} \to \{\mathsf{n} : \mathsf{Nat}\} \to \{\mathsf{i} : \mathsf{Fin\,n}\} \to (\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to \\ (\mathsf{Q_f} : \{\mathsf{n} : \mathsf{Nat}\} \to \mathsf{Fin\,n} \to \mathsf{Set}) \to \mathsf{FSeq\,a\,i} \to \mathsf{Set} \end{array}$$

that is (implicitly) parameterized by data $\mathsf{a} : \mathsf{Set}$, $\mathsf{n} : \mathsf{Nat}$, and $\mathsf{i} : \mathsf{Fin\,n}$ (which are needed to construct the instance $\mathsf{FSeq\,a\,i}$ that P is a predicate on), as well as by a predicate $\mathsf{Q_a}$ on FSeq's type index a and a predicate $\mathsf{Q_f}$ on the type Fin of FSeq's term indices. Similarly, the first input to the deep induction rule for Vec is a predicate P of type $\{\mathsf{a} : \mathsf{Set}\} \to \{\mathsf{n} : \mathsf{Nat}\} \to (\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to (\mathsf{Q_n} : \mathsf{Nat} \to \mathsf{Set}) \to \mathsf{Vec\,a\,n} \to \mathsf{Set}$ that is (implicitly) parameterized by data $\mathsf{a} : \mathsf{Set}$ and $\mathsf{n} : \mathsf{Nat}$ (which are needed to construct the instance $\mathsf{Vec\,a\,n}$ that P is a predicate on), as well as by predicates $\mathsf{Q_a}$ on Vec's type index a and $\mathsf{Q_n}$ on the type Nat of Vec's term indices. However, in the case of Vec the predicate arguments $\mathsf{Q_a}$ and $\mathsf{Q_n}$ are the same at all call sites, so they can be factored out of P as in (12). This simplification can be applied to the deep induction rules for other IFs, such as that for LTerm in Figure 4, as well.

```
-- type constructor for Fin from Agda's standard library:
data Fin : Nat → Set where
  fz : Fin (suc n)
  fs : Fin n → Fin (suc n)

-- type constructor for FSeq:
data FSeq (a : Set) : Fin n → Set where
  finj : a → (i : Fin n) → FSeq a i
  fpair : {i : Fin m} → {j : Fin n} → FSeq b i → FSeq c j → FSeq (b × c) (i +_f j)

-- the Henry Ford encoding of FSeq:
data FSeq (a : Set) : Fin n → Set where
  finj : a → (i : Fin n) → FSeq a i
  fpair : (b × c) ≡ a → {i : Fin m} → {j : Fin n} → FSeq b i → FSeq c j → FSeq a (i +_f j)

-- the predicate lifting for the Henry Ford encoding of FSeq:
FSeq^ : (a → Set) → ({n : Nat} → Fin n → Set) → {i : Fin n} → FSeq a i → Set
FSeq^ Q_a Q_f (finj x i) = Q_a x × Q_f i
FSeq^ Q_a Q_f (fpair p {i} {j} sbi scj) = ∃ [Q_b] ∃ [Q_c]  ≡^ (×^ Q_b Q_c) Q_a p × FSeq^ Q_b Q_f sbi ×
                                          FSeq^ Q_c Q_f scj × (Q_f i → Q_f j → Q_f (i +_f j))

-- the deep induction rule for the Henry Ford encoding of FSeq:
(P : {a : Set} → {n : Nat} → {i : Fin n} →
   (Q_a : a → Set) → (Q_f : {n : Nat} → Fin n → Set) → FSeq a i → Set) →          (Step 1)
  ({a : Set} → (x : a) → {n : Nat} → (i : Fin n) → (Q_a : a → Set) →
   (Q_f : {n : Nat} → Fin n → Set) → Q_a x → Q_f i → P Q_a Q_f (finj x i)) →       (Step 2)
  ({a b c : Set} → (p : (b × c) ≡ a) → {m n : Nat} → {i : Fin m} → {j : Fin n} →
   (sbi : FSeq b i) → (scj : FSeq c j) → (Q_a : a → Set) → (Q_b : b → Set) →
   (Q_c : c → Set) → (Q_f : {n : Nat} → Fin n → Set) →
   P Q_b Q_f sbi → P Q_c Q_f scj → Q_f (i +_f j) → P Q_a Q_f (fpair p sbi scj)) →  (Step 2)
  (Q_a : a → Set) → (Q_f : {n : Nat} → Fin n → Set) → {i : Fin n} → (s : FSeq a i) →
   FSeq^ Q_a Q_f s → P Q_a Q_f s                                                   (Step 3)
```

Fig. 1. Deep induction rule for FSeq.

2. Include one induction hypothesis in D's deep induction rule for each of its data constructors c. The induction hypothesis for c must:

   (a) take as its first arguments all of the ingredients needed to construct an element of D using c.

   (b) take as additional arguments predicates on the type indices and the type of the term indices appearing in the clause of D^ for c.

   (c) take as further arguments terms checking that each argument of c introducing new data of primitive type satisfies the lifting for its type of P's parameterizing predicates (for those that exist), and that each recursive argument of c satisfies (an appropriate instance of) P.

(d) take as its final arguments terms checking that the term index of the element constructed using c satisfies the predicate for its type parameterizing P.
(e) have as its conclusion that the term constructed using c satisfies P.

For example, the induction hypothesis for finj in the deep induction rule for FSeq is:

$$\begin{aligned}&\{\mathsf{a} : \mathsf{Set}\} \to (\mathsf{x} : \mathsf{a}) \to \{\mathsf{n} : \mathsf{Nat}\} \to (\mathsf{i} : \mathsf{Fin\,n}) \to (\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to\\ &\quad(\mathsf{Q_f} : \{\mathsf{n} : \mathsf{Nat}\} \to \mathsf{Fin\,n} \to \mathsf{Set}) \to \mathsf{Q_a\,x} \to \mathsf{Q_f\,i} \to \mathsf{P\,Q_a\,Q_f\,(finj\,x\,i))}\end{aligned}$$

and that for fpair is:

$$\begin{aligned}&\{\mathsf{a\,b\,c} : \mathsf{Set}\} \to (\mathsf{p} : (\mathsf{b} \times \mathsf{c}) \equiv \mathsf{a}) \to \{\mathsf{m\,n} : \mathsf{Nat}\} \to \{\mathsf{i} : \mathsf{Fin\,m}\} \to \{\mathsf{j} : \mathsf{Fin\,n}\} \to\\ &\quad(\mathsf{sbi} : \mathsf{FSeq\,b\,i}) \to (\mathsf{scj} : \mathsf{FSeq\,c\,j}) \to (\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to (\mathsf{Q_b} : \mathsf{b} \to \mathsf{Set}) \to\\ &\quad(\mathsf{Q_c} : \mathsf{c} \to \mathsf{Set}) \to (\mathsf{Q_f} : \{\mathsf{n} : \mathsf{Nat}\} \to \mathsf{Fin\,n} \to \mathsf{Set}) \to\\ &\quad\mathsf{P\,Q_b\,Q_f\,sbi} \to \mathsf{P\,Q_c\,Q_f\,scj} \to \mathsf{Q_f}\,(\mathsf{i} +_{\mathsf{f}} \mathsf{j}) \to \mathsf{P\,Q_a\,Q_f\,(fpair\,p\,sbi\,scj)}\end{aligned}$$

In the former, the first four arguments come from Step 2a, the next two come from Step 2b, the next comes from Step 2c, the final one comes from Step 2d, and the conclusion comes from Step 2e. In the latter, the first ten arguments come from Step 2a, the next four are from Step 2b, the next two are from Step 2c, the final one comes from Step 2d, and the conclusion comes from Step 2e.

3. Conclude that, given an arbitrary element of D, the ingredients needed to construct it, and instantiations of P's parameterizing predicates for its type indices and the type of its term indices, if the element satisfies D's lifting of these instantiating predicates then it satisfies the instantiation of P to them. For example, the conclusion of the deep induction rule for FSeq is:

$$\begin{aligned}&(\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to (\mathsf{Q_f} : \{\mathsf{n} : \mathsf{Nat}\} \to \mathsf{Fin\,n} \to \mathsf{Set}) \to\\ &\quad\{\mathsf{i} : \mathsf{Fin\,n}\} \to (\mathsf{s} : \mathsf{FSeq\,a\,i}) \to \mathsf{FSeq}^\wedge\,\mathsf{Q_a\,Q_f\,s} \to \mathsf{P\,Q_a\,Q_f\,s}\end{aligned}$$

Exactly this methodology has yielded all of the deep induction rules in this paper. Note, however, that if all of the predicates on an IF's type parameters are the same at all call sites then Henry Ford encodings can be skipped and the methodology can be used with the IF's original, non-Henry-Ford-encoded definition. In addition, the predicates on the IF's type indices and the type of its term indices need not be repeated in the conclusion of its deep induction rule since they will have been factored out of P as discussed above, and will therefore scope over the entire rule already.

The accompanying code file contains additional examples illustrating our methodology. The file contains deep induction rules for IFs with term indices given by primitive types (natural-number-indexed lists, i.e., vectors); ADTs (list-indexed sequences); nested types (perfect-tree-indexed sequences); GADTs (LType-indexed LTerms); and IFs (finite-set-indexed and vector-indexed sequences). Only the first and the final three of these examples appear in the text of this paper, in Sections 3 and 6, and in Figures 2 and 1, respectively. However, the latter two IFs are completely representative: in each case the underlying data type is a GADT, which is the most general kind of type-indexed data type possible, and this GADT is indexed by a (deep, in the case of vector-indexed sequences) IF, which is the most general kind of term-indexed data type possible.

We call to attention some particular features of our methodology.

– As in [9,11], there is no need to reflect predicates as data types. For example, although the set of primes cannot be defined by a data type, the primeness predicate Prime on natural numbers can be lifted to a predicate $\mathsf{List}^\wedge\,\mathsf{Prime}$ characterizing lists of primes, and properties of such lists can be proved by the deep induction rule for lists after instantiating Q to Prime.
– A predicate on a type (either a type index or the type of a term index) appearing in an IF D need not hold for *all* elements of that type, but rather only for those elements that actually *can be* indices of elements of D. This observation was not highlighted in [9] but obtains (for type indices) there as well.
– The previous point is in stark contrast to the methods of [22,23], which use only trivial predicates for types of term indices. This has the effect of reducing the deep induction principle for an IF to that for its underlying GADT. For example, in [23] the deep induction rule for vectors is reduced to that for lists.
– The combining function that makes the term index of an element of D constructed using a data constructor c from the term indices of its recursive subdata determines the term-index predicate satisfaction preservation requirement in the clause of $\mathsf{D}^\wedge$ for c. For example, the term-index predicate satisfaction preservation requirement in the clause of $\mathsf{Vec}^\wedge$ for $\_{::}\_$ is $\mathsf{Q_n}\,\mathsf{m} \to \mathsf{Q_n}\,(\mathsf{suc}\,\mathsf{m})$ precisely because the type of $\_{::}\_$ is (up to variable renaming) $\mathsf{a} \to \mathsf{Vec}\,\mathsf{a}\,\mathsf{m} \to \mathsf{Vec}\,\mathsf{a}\,(\mathsf{suc}\,\mathsf{m})$.

5 Case Study: Preserving Properties of Vector-indexed Sequences

We now show how deep induction can be used to prove a non-trivial property of an IF that is deep over its type indices and also indexed by terms of an(other) IF. Write $\_{++}\_$ for Agda's vector concatenation, and consider the IF VSeq of vector-indexed sequences shown in Figure 2. The IF VSeq is analogous to the IF FSeq of finite-set-indexed sequences in Figure 1. The IF VSeq is type-indexed by its first (explicit) type parameter a, and term-indexed by elements of an IF that is deep over VSeq's second (implicit) type parameter d. The lifting and deep induction rule for VSeq for a "primitive predicate" on its first type parameter a and a "data type predicate" (which is necessarily polymorphic in its term index of type Nat) on the data type of its term indices — i.e., on vectors with data from d — are given there as well. These are obtained similarly to how their analogues for FSeq were obtained, and thus exactly as described and illustrated in Section 4.

We can use the deep induction in Figure 2 to prove that, for every $\mathsf{xs} : \mathsf{Vec}\,\mathsf{Nat}\,\mathsf{n}$ and every $\mathsf{s} : \mathsf{VSeq}\,\mathsf{a}\,\mathsf{xs}$ for some $\mathsf{a} : \mathsf{Set}$, if every subterm of s constructed using vinj is indexed by a vector of even length all of whose elements are odd, then s itself is indexed by such a vector. To state this proposition, we

```
-- type constructor for VSeq:
data VSeq (a : Set) {d : Set} : Vec d n → Set where
  vinj : a → (xs : Vec d n) → VSeq a xs
  vpair : {ys : Vec d m} → {zs : Vec d n} → VSeq b ys → VSeq c zs → VSeq (b × c) (ys ++ zs)

-- the Henry Ford encoding of VSeq:
data VSeq (a : Set) {d : Set} : Vec d n → Set where
  vinj : a → (xs : Vec d n) → VSeq a xs
  vpair : (b × c) ≡ a → {ys : Vec d m} → {zs : Vec d n} → VSeq b ys → VSeq c zs → VSeq a (ys ++ zs)

-- the predicate lifting for the Henry Ford encoding of VSeq:
VSeq^ : (a → Set) → ({n : Nat}  → Vec d n → Set) → {xs : Vec d n} → VSeq a xs → Set
VSeq^ Qa Qv (vinj x xs) = Qa x × Qv xs
VSeq^ Qa Qv (vpair p {ys} {zs} sbys sczs) = ∃[Qb] ∃[Qc] ≡^ (×^ Qb Qc) Qa p × VSeq^ Qb Qv sbys ×
                                             VSeq^ Qc Qv sczs × (Qv ys → Qv zs → Qv (ys ++ zs))

-- the deep induction rule for the Henry Ford encoding of VSeq:
VSeqInd :
  (P : {a d : Set} → {n : Nat} → {xs : Vec d n} →
    (Qa : a → Set) → (Qv : {n : Nat} → Vec d n → Set) → VSeq a xs → Set) →
  ({a d : Set} → (x : a) → {n : Nat} → (xs : Vec d n) → (Qa : a → Set) →
    (Qv : {n : Nat} → Vec d n → Set) → Qa x → Qv xs → P Qa Qv (vinj x xs)) →
  ({a b c d : Set} → (p : (b × c) ≡ a) → {m n : Nat} → {ys : Vec d m} → {zs : Vec d n} →
    (sbys : VSeq b ys) → (sczs : VSeq c zs) →
    (Qa : a → Set) → (Qb : b → Set) → (Qc : c → Set) → (Qv : {n : Nat} → Vec d n → Set) →
    P Qb Qv sbys → P Qc Qv sczs → Qv (ys ++ zs) → P Qa Qv (vpair p sbys sczs)) →
  (Qa : a → Set) → (Qv : {n : Nat} → Vec d n → Set) → {xs : Vec d n} → (s : VSeq a xs) →
    VSeq^ Qa Qv s → P Qa Qv s
```

Fig. 2. Deep induction rule for VSeq.

use the predicate

```
eloe : Vec Nat n → Set
eloe xs = even (length xs) × all odd xs
```

Here, even and odd are the standard predicates on Nat, length computes the length of its vector argument, and the predicate all on vectors with elements of type a checks that each of its elements satisfies a given predicate on a. The proposition prop to be proved can then be stated as:

```
prop : {xs : Vec Nat n} → (s : VSeq a xs) → Tree^(QvOnVec eloe) (leaves s) → eloe xs
```

Here, Tree is the type of binary trees with data only at the leaves, Tree^ checks that every datum in a tree satisfies a given predicate on the type of its elements (this is similar to List^ from Section 2), the function

```
leaves : {xs : Vec d n} → VSeq a xs → Tree (Σ Set id × Σ Nat (Vec d))
```

collects into a binary tree the data-index pairs in the vinj-constructed subterms of a vector-indexed sequence, and the function

```
QvOnVec : ({n : Nat} → Vec d n → Set) → Σ Set id × Σ Nat (Vec d) → Set
```

applies a given predicate on vectors to the vector inside such a pair. Complete implementations of these functions — and indeed the code for the entire application in this section — are given in the appendix. This code is also included in the code file accompanying this paper.

Now, in order to use the deep induction rule for VSeq to prove prop, we need to construct a term of type VSeq^ KT eloe s from prop's argument of type Tree^(QvOnVec eloe) (leaves s).[6] The following function does exactly this:

```
mkVSeq^ : (Qv : {n : Nat} → Vec d n → Set) →
  ({m n : Nat} → (ys : Vec d m) → (zs : Vec d n) → Qv ys → Qv zs → Qv (ys ++ zs)) →
  {xs : Vec d n} → (s : VSeq a xs) → Tree^ (QvOnVec Qv) (leaves s) → VSeq^ KT Qv s
```

Note that the evenness predicate does not hold for all indices of type Nat appearing in indices of type Vec that index elements of VSeq. But it *does* hold for all indices of type Nat appearing in indices of type Vec that index elements of VSeq provided it holds for all subterms of those elements constructed using vinj.

6 Case Study: Type Inference for Lambda Terms

As a second application of deep induction we reconsider a problem introduced in [9]. There it was shown how to use deep induction for GADTs to infer the types of terms in a simple typed lambda calculus. In this section we show how to use deep induction for IFs to solve the same problem in two different ways.

The types and terms of the lambda calculus under consideration are given by:

```
data LType : Set → Set where
  bool : LType Bool
  arr : LType b → LType c → LType (b → c)
  list : LType b → LType (List b)

data LTerm : LType a → Set where
  var : (la : LType a) → String → LTerm la
  abs : (lb : LType b) → (lc : LType c) → String → LTerm lc → LTerm (arr lb lc)
  app : (lb : LType b) → (la : LType a) → LTerm (arr lb la) → LTerm lb → LTerm la
  list : (lb : LType b) → List (LTerm lb) → LTerm (list lb)
                                                                            (15)
```

In [9] LTerm is seen as a GADT that is indexed by the GADT LType, which is itself non-trivially type-indexed, so LTerm is a deep GADT. In fact, this is the

[6] We use the constantly ⊤-valued predicate KT as our predicate on a since using a non-trivial predicate on a here wouldn't introduce anything new over [9].

only possible way to see LTerm as a data type that is type-indexed but not term-indexed. The deep induction rule from [9] for (the Henry Ford encoding of) LTerm regarded as a (deep) GADT can therefore only be used once we have propagated to both LType and LTerm a primitive predicate $\mathsf{Q_a}$ on the type index a that LTerm inherits from (the Henry Ford encoding of) LType. The Henry Ford encodings of the original definitions of LType and LTerm given in (15) are repeated from [9] in Figure 3 below. The liftings of $\mathsf{Q_a}$ to these encodings and the corresponding deep induction rule for LTerm regarded *as a deep GADT* can be found in [9].

However, LTerm can also be seen *as an IF* that has no type indices whatsoever, but is instead indexed only by *terms* of LType. When LTerm is viewed in this way, the predicate on the type LType of LTerms's term indices that is to be used in the lifting and deep induction rule for LTerm can be obtained in two different ways. The first way is by propagating a primitive predicate $\mathsf{Q_a : a \to Set}$ on the type index a to both LType and LTerm, by analogy with how $\mathsf{Q_a}$ was propagated to LType and LTerm when seen as GADTs in [9]. The second way is by giving a predicate $\mathsf{Q : \{a : Set\} \to LType\,a \to Set}$ directly on the entire data type LType. Of course, any such data type predicate must necessarily be polymorphic over LType's type index. We now explore each of these options in turn.

We first consider the liftings and the deep induction rule for LTerm seen as a term-indexed only IF that uses a primitive predicate $\mathsf{Q_a}$ and the Henry Ford encodings of both LType and LTerm. These liftings and their corresponding deep induction rule were mentioned in [10] and included in the code that accompanies that paper, but they were not included in the paper itself. We detail them here.

Since LType is type-indexed by a, the lifting $\mathsf{LType}^\wedge$ of an element predicate $\mathsf{Q_a}$ to LType is identical to that in [9]. This lifting is recalled in Figure 3. On the other hand, the lifting $\mathsf{LTerm}^\wedge$ of an element predicate $\mathsf{Q_a}$ to LTerm is given by:

$$\begin{array}{l}
\mathsf{LTerm}^\wedge : (\mathsf{a} \to \mathsf{Set}) \to \{\mathsf{la} : \mathsf{LType\,a}\} \to \mathsf{LTerm\,la} \to \mathsf{Set}\\
\mathsf{LTerm}^\wedge\,\mathsf{Q_a}\,\{\mathsf{la}\}\,(\mathsf{var\,s}) \;=\; \mathsf{LType}^\wedge\,\mathsf{Q_a}\,\mathsf{la}\\
\mathsf{LTerm}^\wedge\,\mathsf{Q_a}\{\mathsf{la}\}\,(\mathsf{abs\,p\,lb\,lc\,q\,s\,t}) \;=\; \exists\,[\mathsf{Q_b}]\,\exists\,[\mathsf{Q_c}] \equiv^\wedge (\to^\wedge\;\mathsf{Q_b}\,\mathsf{Q_c})\,\mathsf{Q_a}\,\mathsf{p}\,\times\\
\qquad\qquad \mathsf{LType}^\wedge\,\mathsf{Q_b}\,\mathsf{lb}\,\times\,\mathsf{LTerm}^\wedge\,\mathsf{Q_c}\,\mathsf{t}\,\times\,(\mathsf{LType}^\wedge\,\mathsf{Q_c}\,\mathsf{lc} \to \mathsf{LType}^\wedge\,\mathsf{Q_a}\,\mathsf{la})\\
\mathsf{LTerm}^\wedge\,\mathsf{Q_a}\,\{\mathsf{la}\}\,(\mathsf{app\,lb\,f\,x}) \;=\; \exists\,[\mathsf{Q_b}]\,\mathsf{LTerm}^\wedge\,(\to^\wedge\;\mathsf{Q_b}\,\mathsf{Q_a})\,\mathsf{f}\,\times\,\mathsf{LTerm}^\wedge\,\mathsf{Q_b}\,\mathsf{x}\,\times\\
\qquad\qquad (\mathsf{LType}^\wedge\,(\to^\wedge\;\mathsf{Q_b}\,\mathsf{Q_a})\,(\mathsf{arr\,refl\,lb\,la}) \to \mathsf{LType}^\wedge\,\mathsf{Q_b}\,\mathsf{lb} \to \mathsf{LType}^\wedge\,\mathsf{Q_a}\,\mathsf{la})\\
\mathsf{LTerm}^\wedge\,\mathsf{Q_a}\,\{\mathsf{la}\}\,(\mathsf{list\,p\,lb\,q\,t}) \;=\; \exists\,[\mathsf{Q_b}] \equiv^\wedge (\mathsf{List}^\wedge\,\mathsf{Q_b})\,\mathsf{Q_a}\,\mathsf{p}\,\times\,\mathsf{LType}^\wedge\,\mathsf{Q_b}\,\mathsf{lb}\,\times\\
\qquad\qquad \mathsf{List}^\wedge\,(\mathsf{LTerm}^\wedge\,\mathsf{Q_b})\,\mathsf{t}\,\times\,(\mathsf{LType}^\wedge\,(\mathsf{List}^\wedge\,\mathsf{Q_b})\,(\mathsf{list\,refl\,lb}) \to \mathsf{LType}^\wedge\,\mathsf{Q_a}\,\mathsf{la})
\end{array} \tag{16}$$

Here, $\equiv^\wedge$ is the lifting for $\equiv$ given in (8), and $\to^\wedge$ is defined by

$$\begin{array}{l}
\to^\wedge : (\mathsf{a} \to \mathsf{Set}) \to (\mathsf{b} \to \mathsf{Set}) \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{Set}\\
\to^\wedge\;\mathsf{Q}\,\mathsf{Q}'\,\mathsf{f} \;=\; (\mathsf{x} : \mathsf{a}) \to \mathsf{Q\,x} \to \mathsf{Q}'\,(\mathsf{f\,x})
\end{array}$$

In addition, the single term $\mathsf{Type}^\wedge\,\mathsf{Q_a}\,\mathsf{la}$ in the clause of $\mathsf{LTerm}^\wedge$ for var comes from Step 3 of the methodology for constructing the clauses of the lifting given on page 11, and the other steps contribute no terms. In the clause of $\mathsf{LTerm}^\wedge$ for abs, the first two terms come from Step 1, the third comes from Step 2, and the fourth comes from Step 3. In the clause of $\mathsf{LTerm}^\wedge$ for app, the first and second terms come from Step 2, and the third comes from Step 3. And, finally, in the clause of $\mathsf{LTerm}^\wedge$ for list, the first two terms come from Step 1, the third comes from Step 2, and the fourth comes from Step 3. Note that the second term here is necessary to accommodate the case when t is empty.

The lifting of $\mathsf{Q_a}$ to the IF LTerm in (16) appears at first glance to differ from the lifting of $\mathsf{Q_a}$ to the GADT LTerm in [9] because the clauses of the lifting in (16) include term-index predicate satisfaction requirements while those of [9] do not. However, it is not hard to see that they are, in fact, identical. Indeed, the inductive nature of any IF D's definition ensures that it is always possible to project out from a proof that an element d of D satisfies the lifting of a predicate Q to its type a proof that d's index also satisfies the lifting of Q to D's index type. In fact, the accompanying code file already contains two such projections, namely the functions $\mathsf{QfOnIndex}$ and $\mathsf{QvOnIndex}$ that are local to the deep induction rules in Figures 1 and 2, respectively, and necessary to construct their proof witnesses. For LTerm, the following function $\mathsf{QprojIndex}$ asserts that if an LTerm t satisfies the lifting $\mathsf{LTerm}^\wedge\,\mathsf{Q_a}$ of a primitive predicate $\mathsf{Q_a} : \mathsf{a} \to \mathsf{set}$, then t's index satisfies the lifting $\mathsf{LType}^\wedge\,\mathsf{Q_a}$:

```
QprojIndex : (Qa : a → Set) → {la : LType a} → (t : LTerm la) →
                 LTerm^ Qa t → LType^ Qa la
QprojIndex Qa (var s) hQa = hQa
QprojIndex Qa (abs p lb lc q s t) (Qb, Qc, e, ^Qblb, ^Qt, hQa) =
                 hQa (QprojIndex Qc t ^Qt)
QprojIndex Qa (app lb f x) (Qb, ^Qf, ^Qx, hQa) =
                 hQa (^QOnIndex (→^ Qb Qa) f ^Qf) (QprojIndex Qb x ^Qx)
QprojIndex Qa (list p lb q t) (Qb, e, ^Qblb, ^Qt, hQa) =
                 hQa (Qb, λ_ → refl, ^Qblb)
```

The function $\mathsf{QprojIndex}$ can now be used to eliminate redundancies in the definition of $\mathsf{LTerm}^\wedge$ in (16). Specifically, in the clause for abs of the definition of $\mathsf{LTerm}^\wedge$, the type $\mathsf{LType}^\wedge\,\mathsf{Q_c}\,\mathsf{lc} \to \mathsf{LType}^\wedge\,\mathsf{Q_a}\,\mathsf{la}$ can be omitted since its hypothesis $\mathsf{LType}^\wedge\,\mathsf{Q_c}\,\mathsf{lc}$ is derivable from $\mathsf{LTerm}^\wedge\,\mathsf{Q_c}\,\mathsf{t}$ using $\mathsf{QprojIndex}$ and, together with the first and second terms, these imply the conclusion $\mathsf{LType}^\wedge\,\mathsf{Q_a}\,\mathsf{la}$ of the condition by the definition of $\mathsf{LType}^\wedge$. In the clause for app, the term $\mathsf{LType}^\wedge\,(\to^\wedge\ \mathsf{Q_b}\,\mathsf{Q_a})\,(\mathsf{arr\,refl\,lb\,la}) \to \mathsf{LType}^\wedge\,\mathsf{Q_b}\,\mathsf{lb} \to \mathsf{LType}^\wedge\,\mathsf{Q_a}\,\mathsf{la}$ can be omitted since $\mathsf{LType}^\wedge\,\mathsf{Q_a}\,\mathsf{la}$ follows from $\mathsf{LType}^\wedge\,(\to^\wedge\ \mathsf{Q_b}\,\mathsf{Q_a})\,(\mathsf{arr\,refl\,lb\,la})$ and $\mathsf{LType}^\wedge\,\mathsf{Q_b}\,\mathsf{lb}$ by the definition of $\mathsf{LType}^\wedge$, and these can be obtained from $\mathsf{LTerm}^\wedge\,(\to^\wedge\ \mathsf{Q_b}\,\mathsf{Q_a})\,\mathsf{f}$ and $\mathsf{LTerm}^\wedge\,\mathsf{Q_b}\,\mathsf{x}$, respectively, using $\mathsf{QprojIndex}$. Finally, in the clause for list, the type $\mathsf{LType}^\wedge\,(\mathsf{List}^\wedge\,\mathsf{Q_b})\,(\mathsf{list\,refl\,lb}) \to \mathsf{LType}^\wedge\,\mathsf{Q_a}\,\mathsf{la}$ can be omitted since the definitions of LType, LTerm, and $\mathsf{LType}^\wedge$, and the term of type $\mathsf{LType}^\wedge\,\mathsf{Q_b}\,\mathsf{lb}$ of this clause of $\mathsf{LTerm}^\wedge$ for list, ensure that the conclusion always holds. These simplifications give the variant of $\mathsf{LTerm}^\wedge$ in Figure 3, which does indeed coincide with the lifting given in [9] as promised. We use this variant in the deep induction rule for LTerm there.

To derive the deep induction rule for a predicate P parameterized on a primitive predicate $\mathsf{Q_a}$ and the Henry Ford encodings of LType and the IF LTerm, we first note that $\mathsf{Q_a}$ cannot be factored out of the definition of P since it is used at different instances in the recursive calls of LType and LTerm. In the deep induction rule for LTerm the predicate P thus comes from Step 1 of the methodology for constructing deep induction rules given on page 12. The induction hypotheses for each of LTerm's data constructors are constructed as in Step 2 there (with repetitions omitted). For example, in the induction hypothesis habs for abs the first ten arguments come from Step 2a, the next three come from Step 2b, the next three come from Step 2c, the final argument comes from Step 2d, and the conclusion comes from Step 2e. The induction hypotheses for the other data constructors are obtained similarly. Finally, the conclusion of the deep induction rule is obtained as in Step 3 of the overall methodology. After simplification by $\mathsf{QprojIndex}$ (similar to that carried out for $\mathsf{LTerm}^\wedge$ above), the deep induction rule in Figure 3 is easily seen (essentially) to coincide with that for LTerms from [9].

We now turn our attention to the deep induction rule for LTerm that uses a data type predicate $\mathsf{Q} : \{\mathsf{a} : \mathsf{Set}\} \to \mathsf{LType\,a} \to \mathsf{Set}$ that is given directly rather than obtained by lifting a predicate on a. Neither this rule nor its associated lifting was mentioned, let alone coded, for [10]. We give both in Figure 4 here. To derive them, we first observe that LTerm does not actually need to be Henry-Ford-encoded when a data type predicate Q on LType is used (although it certainly *can* be). In Figure 4 we therefore lift Q to the original definition of LTerm from (15). Further, we need give no lifting of Q to LType since Q is already a predicate on that type, so we use the original definition of LType in the code in Figure 4. The version $\mathsf{LTerm}^{\wedge\prime}$ of the lifting for the original definition of LTerm in (15) obtained from the methodology on page 11 for constructing liftings is

$$\begin{array}{l}
\mathsf{LTerm}^{\wedge\prime} : (\{\mathsf{a} : \mathsf{Set}\} \to \mathsf{LType\,a} \to \mathsf{Set}) \to \{\mathsf{la} : \mathsf{LType\,a}\} \to \mathsf{LTerm\,la} \to \mathsf{Set}\\
\mathsf{LTerm}^{\wedge\prime}\,\mathsf{Q}\,(\mathsf{var\,la\,s}) = \mathsf{Q\,la}\\
\mathsf{LTerm}^{\wedge\prime}\,\mathsf{Q}\,(\mathsf{abs\,lb\,lc\,s\,t}) = \mathsf{Q\,lb} \times \mathsf{LTerm}^{\wedge\prime}\,\mathsf{Q\,t} \times (\mathsf{Q\,lc} \to \mathsf{Q}\,(\mathsf{arr\,lb\,lc}))\\
\mathsf{LTerm}^{\wedge\prime}\,\mathsf{Q}\,(\mathsf{app\,lb\,la\,f\,x}) = \mathsf{LTerm}^{\wedge\prime}\,\mathsf{Q\,f} \times \mathsf{LTerm}^{\wedge\prime}\,\mathsf{Q\,x} \times\\
\qquad\qquad (\mathsf{Q}\,(\mathsf{arr\,lb\,la}) \to \mathsf{Q\,lb} \to \mathsf{Q\,la})\\
\mathsf{LTerm}^{\wedge\prime}\,\mathsf{Q}\,(\mathsf{list\,lb\,t}) = \mathsf{Q\,lb} \times \mathsf{List}^\wedge\,(\mathsf{LTerm}^{\wedge\prime}\,\mathsf{Q})\,\mathsf{t} \times\\
\qquad\qquad (\mathsf{List}^\wedge\,\mathsf{Q}\,(\mathsf{map\,projType\,t}) \to \mathsf{Q}\,(\mathsf{list\,lb}))
\end{array} \tag{17}$$

Here, the clause of $\mathsf{LTerm}^{\wedge\prime}$ for var the term $\mathsf{Q\,la}$ on its right-hand side comes from Step 3 of the methodology for constructing the clauses of the lifting given on page 11, and the other steps contribute no terms. In the clause for abs of $\mathsf{LTerm}^{\wedge\prime}$, the first term comes from Step 1, the second term comes from Step 2, and the third comes from Step 3. In the clause for app of $\mathsf{LTerm}^{\wedge\prime}$, the first two terms come from Step 2, and the final term comes from Step 3. Finally, in the clause for list of $\mathsf{LTerm}^{\wedge\prime}$, the first term comes from Step 1 (to account for the case when t is the empty list), the second term comes from Step 2, and the final term comes from Step 3. There, $\mathsf{map} :: (\mathsf{a} \to \mathsf{b}) \to \mathsf{List\,a} \to \mathsf{List\,b}$ is the standard map function for lists, and $\mathsf{projType} : \{\mathsf{la} : \mathsf{LType\,a}\} \to \mathsf{LTerm\,la} \to \mathsf{LType\,a}$ is given by $\mathsf{projType}\,\{\mathsf{la} = \mathsf{la}\}\,\mathsf{t} = \mathsf{la}$. However, like $\mathsf{LTerm}^\wedge$ above, $\mathsf{LTerm}^{\wedge\prime}$ can be simplified. This time we use the following analogue of $\mathsf{QprojIndex}$ for data type predicates to arrive at the simplified version of $\mathsf{LTerm}^{\wedge\prime}$ in Figure 4:

```
mutual
  QprojProduct : (Q : a : Set → LType′ a → Set) → {la : LType′ a} →
    (t : List (LTerm′ la)) → List^ (LTerm^′ Q) t → List^ Q (map projType t)
  QprojProduct Q [ ] ^Qt = tt
  QprojProductQ (x :: xs) (Qx, ^Qxs) = (QprojIndex′ Q x Qx, QprojProduct Q xs ^Qxs)

  QprojIndex′ : (Q : {a : Set} → LType a → Set) → {la : LType a} →
            (t : LTerm la) → LTerm^′ Q t → Q la
  QprojIndex′ Q (var la s) Qla = Qla
  QprojIndex′ Q (abs lb lc s t) (Qlb, ^Qt, hQ) = hQ Qlb (QprojIndex′ Q t ^Qt)
  QprojIndex′ Q (app lb la f x) (^Qf, ^Qx, hQ) =
     hQ (QprojIndex′ Q f ^Qf)(QprojIndex′ Q x ^Qx)
  QprojIndex′ Q (list lb t) (Qlb, ^Qt, hQ) = hQ (QprojProduct Q t ^Qt)
```

But note that, even when we can project out the hypotheses of a clause's predicate satisfaction requirement, we cannot remove its conclusion from the lifting unless we somehow know that Q respects the calculus' type formers.

To derive the corresponding deep induction rule for a predicate P parameterized on a data type predicate Q and the original definitions of LType and LTerm, we first observe that Q can be factored out of the predicate P to be proved for all LTerms. In the deep induction rule for LTerm, the predicates Q and P both therefore come from Step 1 of the overall methodology for constructing deep induction rules given on page 12. The induction hypothesis for each of LTerm's data constructors is then constructed as in Step 2 there. For example, in the induction hypothesis habs for abs the first six arguments come from Step 2a, the next one comes from Step 2c, the next one comes from Step 2d, and the conclusion comes from Step 2e. The induction hypotheses for the other data constructors are obtained similarly. Finally, the conclusion of the deep induction rule is obtained as in Step 3 of the overall methodology. Note that the deep induction rule for LTerm in Figure 4 differs fundamentally from that in Figure 3 (and thus from that in [9]). This is because the term-index predicate satisfaction requirements in $\mathsf{LTerm}^{\wedge\prime}$ are not redundant, and therefore cannot be eliminated, when lifting a data type predicate on LType to LTerms.

We can now use the deep induction rules from Figure 3 and Figure 4 to compute the types of lambda terms. This application was discussed in [10], and the types of lambda terms were computed using the rule from Figure 3 in the code file accompanying that paper. However, for space reasons, this computation did not appear in [10] itself. In this paper, we include in Figure 5 the code for this computation that uses the deep induction rule in Figure 3. In addition, we include in Figure 6 the code for this computation that instead uses the deep induction rule in Figure 4. As noted in [10], the type of a given lambda term is always immediately accessible as its index, so we could simply project its type out instead of computing it. (This would be similar to projecting out a vector's natural number index rather than computing the vector's length recursively.)

```
-- the Henry Ford encoding of LType:
data LType (a : Set) : Set where
  bool : Bool ≡ a → LType a
  arr : (b → c) ≡ a → LType b → LType c → LType a
  list : List b ≡ a → LType b → LType a

-- the Henry Ford encoding of LTerm:
data LTerm (la : LType a) : Set where
  var : String → LTerm la
  abs : (p : (b → c) ≡ a) → (lb : LType b) → (lc : LType c) → arr p lb lc ≡ la → String → LTerm lc →
        LTerm la
  app : (lb : LType b) → LTerm (arr refl lb la) → LTerm lb → LTerm la
  list : (p : List b ≡ a) → (lb : LType b) → list p lb ≡ la → List (LTerm lb) → LTerm la

-- the predicate lifting of a primitive predicate to the Henry Ford encoding of LType:
LType^ : (a → Set) → LType a → Set
LType^ Q_a (bool p) = ∃[Q_bool] ≡^ Q_bool Q_a p
LType^ Q_a (arr p lb lc) = ∃[Q_b] ∃[Q_c] ≡^ (→^ Q_b Q_c) Q_a p × LType^ Q_b lb × LType^ Q_c lc
LType^ Q_a (list p lb) = ∃[Q_b] ≡^ (List^ Q_b) Q_a p × LType^ Q_b lb

-- the predicate lifting of a primitive predicate to the Henry Ford encoding of LTerm (simplified):
LTerm^ : (a → Set) → {la : LType a} → LTerm la → Set
LTerm^ Q_a {la} (var s) = LType^ Q_a la
LTerm^ Q_a (abs p lb lc q s t) = ∃ [Q_b] ∃ [Q_c] ≡^ (→^ Q_b Q_c) Q_a p × LType^ Q_b lb × LTerm^ Q_c t
LTerm^ Q_a (app lb f x) = ∃ [Q_b] LTerm^ (→^ Q_b Q_a) f × LTerm^ Q_b x
LTerm^ Q_a (list p lb q t) = ∃ [Q_b] ≡^ (List^ Q_b) Q_a p × LType^ Q_b lb × List^ (LTerm^ Q_b) t

-- the deep induction rule for a primitive predicate and the Henry Ford encoding of LTerm:
LTermInd :
  (P : {a : Set} → (a → Set) → {la : LType a} → LTerm la → Set) →
     (hvar : {a : Set} → {la : LType a} → (s : String) → (Q_a : a → Set) → LType^ Q_a la →
                  P Q_a {la} (var s)) →
     (habs : {a b c : Set} → (p : (b → c) ≡ a) → {la : LType a} → (lb : LType b) → (lc : LType c) →
                  (q : arr p lb lc ≡ la) → (s : String) → (t : LTerm lc) → (Q_a : a → Set) →
                  (Q_b : b → Set) → (Q_c : c → Set) → ≡^ (→^ Q_b Q_c) Q_a p →
                  LType^ Q_b lb → P Q_c t → LType^ Q_a la → P Q_a (abs p lb lc q s t)) →
     (happ : {a b : Set} → {la : LType a} → (lb : LType b) → (f : LTerm (arr refl lb la)) →
                  (x : LTerm lb) → (Q_a : a → Set) → (Q_b : b → Set) →
                  P (→^ Q_b Q_a) f → P Q_b x → LType^ Q_a la → P Q_a (app lb f x)) →
     (hlist : {a b : Set} → (p : List b ≡ a) → {la : LType a} → (lb : LType b) → (q : list p lb ≡ la) →
                  (t : List (LTerm lb)) → (Q_a : a → Set) → (Q_b : b → Set) →≡^ (List^ Q_b) Q_a p →
                  LType^ Q_b lb → List^ (P Q_b) t → LType^ Q_a la → P Q_a (list p lb q t)) →
     {a : Set} → (Q_a : a → Set) → {la : LType a} → (t : LTerm la) → LTerm^ Q_a t → P Q_a t
```

Fig. 3. Deep induction rule for a primitive predicate and the Henry-Ford-encoded IF LTerm.

```
-- the predicate lifting of a data type predicate to LTerm' (simplified):
LTerm^' : ({a : Set} → LType a → Set) → {la : LType a} → LTerm la → Set
LTerm^' Q (var la s) = Q la
LTerm^' Q (abs lb lc s t) = Q lb × LTerm^' Q t  × Q (arr lb lc)
LTerm^' Q (app lb la f x) = LTerm^' Q f × LTerm^' Q x  × Q la
LTerm^' Q (list lb t) = Q lb × List^ (LTerm^' Q) t × Q (list lb)

-- the deep induction rule for a data type predicate for LTerm':
LTermInd' :
  (Q : {a : Set} → LType a → Set) →
  (P : {a : Set} → {la : LType a} → LTerm la → Set) →
      (hvar : {a : Set} → (la : LType a) → (s : String) → Q la → P (var la s)) →
      (habs : {b c : Set} → (lb : LType b) → {lc : LType c} → (s : String) → (t : LTerm lc) →
            P t → Q (arr lb lc) → P (abs lb lc s t)) →
      (happ : {a b : Set} → {lb : LType b} → {la : LType a} → (f : LTerm (arr lb la)) →
            (x : LTerm lb) → P f → P x → Q la → P (app lb la f x)) →
      (hlist : {b : Set} → (lb : LType b) → (t : List (LTerm lb)) → List^ P t → Q (list lb) →
            P (list lb t)) →
      {a : Set} → (la : LType a) → (t : LTerm la) → LTerm^' Q t → P t
```

Fig. 4. Deep induction rule for a data type predicate and the original IF LTerm.

However, we will instead use the fact that each lambda term contains its type index to check that the type indices computed for them by the deep induction rules in Figures 3 and 4 are actually correct.

To use the deep induction rule LTermInd in Figure 3 we use the predicate P parameterized by a primitive predicate $\mathsf{Q_a}$ given by

$$\begin{aligned}&\mathsf{P : \{a : Set\} \to (a \to Set) \to \{la : LType\,a\} \to LTerm\,la \to Set}\\&\mathsf{P\,\{a\}\,Q_a\,t = Maybe\,(LType\,a)}\end{aligned}\tag{18}$$

in LTerm's deep induction rule. Here, Maybe a is the standard type

```
data Maybe (a : Set) : Set where
  nothing : Maybe a
  just : a → Maybe a
```

from Agda's standard library, for which just x represents a successful computation that returns the value x of type a, and nothing represents a failed computation with return type a. The induction hypothesis for each of LTerm's data constructors attempts to infer the LType of a term constructed using that data constructor. For example, the induction hypothesis habs for abs is given by

$$\begin{aligned}&\mathsf{habs\,p\,lb\,lc\,q\,s\,t\,Q_a\,Q_b\,Q_c\,e\,{}^\wedge Qblb\,(just\,\_)\,{}^\wedge Qala = just\,(arr\,p\,lb\,lc)}\\&\mathsf{habs\,p\,lb\,lc\,q\,s\,t\,Q_a\,Q_b\,Q_c\,e\,{}^\wedge Qblb\,nothing\,{}^\wedge Qala\ = nothing}\end{aligned}$$

and the induction hypothesis hlist for list is given by

$$\begin{aligned}
&\mathsf{hlist\ p\ lb\ q\ [\,]\ Q_a\ Q_b\ e\ {}^\wedge Qblb\ {}^\wedge Pt\ {}^\wedge Qala} && = \mathsf{nothing}\\
&\mathsf{hlist\ p\ lb\ q\ (x :: xs)\ Q_a\ Q_b\ e\ {}^\wedge Qblb\ (just\ lb', {}^\wedge Pxs)\ {}^\wedge Qala} && = \mathsf{just\ (list\ p\ lb')}\\
&\mathsf{hlist\ p\ lb\ q\ (x :: xs)\ Q_a\ Q_b\ e\ {}^\wedge Qblb\ (nothing, {}^\wedge Pxs)\ {}^\wedge Qala} && = \mathsf{nothing}
\end{aligned}$$

Note that the empty list is a particular source of failure for type inference since it contains no data on which to recurse.

We can now compute the LType of a given LTerm using

$$\begin{aligned}
&\mathsf{inferType} : \{\mathsf{la} : \mathsf{LType\ a}\} \to \mathsf{LTerm\ la} \to \mathsf{Maybe\ (LType\ a)}\\
&\mathsf{inferType\ t} = \mathsf{LTermInd\ P\ hvar\ habs\ happ\ hlist\ KT\ t\ (LTerm^\wedge KT\ t)}
\end{aligned}$$

Here, P is the predicate in (18) and, because we are computing the LType of an LTerm rather than projecting it, $\mathsf{Q_a}$ is instantiated to the constantly $\top$-valued predicate KT on a. In addition, $\mathsf{LTerm^\wedge KT}$ is the lifting the constantly $\top$-valued predicate on LType to LTerm (which is itself obtained by lifting KT to LTypes).[7] Finally, hvar, habs, happ, and hlist are the aforementioned induction hypotheses corresponding to LTerm's data constructors var, abs, app, and list, respectively. This formalization verifies, for example, that the LType representing the type of the LTerm representing the lambda term $(\lambda x.x)y$ is exactly the LType representing the type of the LTerm representing the variable y; that the LType representing the type of the LTerm representing a list all of whose elements have the (common) type represented by a is $\mathsf{list\ a}$; and that the empty list is untypeable (because there are no elements to infer the type of, as noted above). The full instance of the deep induction rule from Figure 3 for this application appears in Figure 5 and in the code accompanying this paper.

Now, to do this same type inference using the deep induction rule $\mathsf{LTermInd'}$ in Figure 4 we instantiate its predicates Q and P to

$$\begin{aligned}
&\mathsf{Q} : \{\mathsf{a} : \mathsf{Set}\} \to \mathsf{LType\ a} \to \mathsf{Set}\\
&\mathsf{Q} = \mathsf{KT}
\end{aligned}$$

and

$$\begin{aligned}
&\mathsf{P} : \{\mathsf{a} : \mathsf{Set}\} \to \{\mathsf{la} : \mathsf{LType\ a}\} \to \mathsf{LTerm\ la} \to \mathsf{Set}\\
&\mathsf{P}\ \{\mathsf{a}\}\ \mathsf{t} = \mathsf{Maybe\ (LType\ a)}
\end{aligned} \qquad (19)$$

The induction hypothesis for each of LTerm's data constructors attempts to infer the LType of a term constructed using that data constructor. For example, the induction hypothesis habs for abs is given by

$$\begin{aligned}
&\mathsf{habs\ lb\ \{lc\}\ s\ t\ (just\ \_)\ Qarr} && = \mathsf{just\ (arr\ lb\ lc)}\\
&\mathsf{habs\ lb\ s\ t\ nothing\ Qarr} && = \mathsf{nothing}
\end{aligned}$$

[7] To lift the constantly $\top$-valued predicate on LTypes to LTerms we postulate function extensionality and identify the singleton types $\top$, $\top \times \top$, and $\mathsf{a} \to \top$ for any type a. These propositions are provable in type theories such as homotopy type theory, but must be added as postulates in vanilla Agda.

```
-- type inference for LTerm via the algorithm in [9]:

inferType : {la : LType a} → LTerm la → Maybe (LType a)
inferType t = LTermInd P hvar habs happ hlist KT t (LTerm^KT t) where
  --
  P : {a : Set} → (a → Set) → {la : LType a} → LTerm la → Set
  P {a} Qa t = Maybe (LType a)
  --
  hvar : {a : Set} → {la : LType a} → (s : String) → (Qa : a → Set) →
                          LType^ Qa la → P Qa {la} (var s)
  hvar {la = la} s Qa ^Qala = just la
  --
  habs : {a b c : Set} → (p : (b → c) ≡ a) → {la : LType a} → (lb : LType b) →
           (lc : LType c) → (q : arr p lb lc ≡ la) → (s : String) → (t : LTerm lc) →
           (Qa : a → Set) → (Qb : b → Set) → (Qc : c → Set) → ≡^ (→^ Qb Qc) Qa p →
           LType^ Qb lb → P Qc t → LType^ Qa la → P Qa (abs p lb lc q s t)
  habs p lb lc q s t Qa Qb Qc e ^Qblb (just _) ^Qala = just (arr p lb lc)
  habs p lb lc q s t Qa Qb Qc e ^Qblb nothing ^Qala = nothing
  --
  happ : {a b : Set} → {la : LType a} → (lb : LType b) → (f : LTerm (arr refl lb la)) →
           (x : LTerm lb) → (Qa : a → Set) → (Qb : b → Set) →
           P (→^ Qb Qa) f → P Qb x → LType^ Qa la → P Qa (app lb f x)
  happ {la = la} lb f x Qa Qb (just _) (just _) ^Qala = just la
  happ lb f x Qa Qb (just _) nothing ^Qala = nothing
  happ lb f x Qa Qb nothing Px ^Qala = nothing
  --
  hlist : {a b : Set} → (p : List b ≡ a) → {la : LType a} → (lb : LType b) →
           (q : list p lb ≡ la) → (t : List (LTerm lb)) → (Qa : a → Set) → (Qb : b → Set) →
           ≡^ (List^ Qb) Qa p → LType^ Qb lb → List^ (P Qb) t → LType^ Qa la →
           P Qa (list p lb q t)
  hlist p lb q [ ] Qa Qb e ^Qblb ^Pt ^Qala = nothing
  hlist p lb q (x :: xs) Qa Qb e ^Qblb (just lb′, ^Pxs) ^Qala = just (list p lb′)
  hlist p lb q (x :: xs) Qa Qb e ^Qblb (nothing, ^Pxs) ^Qala = nothing
```

Fig. 5. Type inference for lambda terms using the deep induction rule from Figure 3.

and the induction hypothesis hlist for list is given by

```
hlist lb [ ] ^Pt Qlb                  = nothing
hlist lb (x :: xs) (nothing, Pxs) Qlb = nothing
hlist lb (x :: xs) (just lb′, Pxs) Qlb  = just (list lb′)
```

Here, again, the empty list is a source of failure for type inference.

We can now compute the LType of a given LTerm using

```
inferType′ : {la : LType a} → LTerm la → Maybe (LType a)
inferType′ {la = la} t = LTermInd′ Q P hvar habs happ hlist la t (LTerm^KT′ t)
```

Here, Q and P are the predicates in (19). In addition, LTerm^KT′ is the lifting of the constantly $\top$-valued predicate Q to LTerm, and hvar, habs, happ, and hlist are the aforementioned induction hypotheses corresponding to LTerm's data constructors var, abs, app, and list, respectively. Like the formalization in Figure 5, the formalization in Figure 6 verifies, for example, that the LType representing the type of the LTerm representing the lambda term $(\lambda x.x)y$ is exactly the LType representing the type of the LTerm representing the variable y; that the LType representing the type of the LTerm representing a list all of whose elements have the (common) type represented by a is list a; and that the empty list is untypeable (because there are no elements to infer the type of, as noted above). The full instance of the deep induction rule from Figure 4 for this application appears in Figure 6 and in the code accompanying this paper.

```
-- type inference for LTerm' via the algorithm in [9]:

inferType′ : {la : LType′ a} → LTerm′ la → Maybe (LType′ a)
inferType′ {la = la} t = LTermInd′ Q P hvar habs happ hlist la t (LTerm^KT′ t) where
 --
 Q : {a : Set} → LType′ a → Set
 Q = KT′
 --
 P : {a : Set} → {la : LType′ a} → LTerm′ la → Set
 P {a} t = Maybe (LType′ a)
 --
 hvar : {a : Set} → (la : LType′ a) → (s : String) → Q la → P (var la s)
 hvar la s Qla = just la
 --
 habs : {b c : Set} → (lb : LType′ b) → {lc : LType′ c} → (s : String) → (t : LTerm′ lc) →
          P t → Q (arr lb lc) → P (abs lb lc s t)
 habs lb {lc = lc} s t (just _) Qarr = just (arr lb lc)
 habs lb s t nothing Qarr = nothing
 --
 happ : {a b : Set} → {lb : LType′ b} → {la : LType′ a} → (f : LTerm′ (arr lb la)) →
          (x : LTerm′ lb) → P f → P x → Q la → P (applb la f x)
 happ {la = la} f x (just _) (just _) Qla = just la
 happ f x (just _) nothing Qla = nothing
 happ f x nothing Px Qla = nothing
 --
 hlist : {b : Set} → (lb : LType′ b) → (t : List (LTerm′ lb)) → List^ P t → Q (list lb) → P (list lb t)
 hlist lb [ ] ^Pt Qlb = nothing
 hlist lb (x :: xs) (nothing, Pxs) Qlb = nothing
 hlist lb (x :: xs) (just lb′, Pxs) Qlb = just (list lb′)
```

Fig. 6. Type inference for lambda terms using the deep induction rule from Figure 4.

7 Conclusion and Future Directions

This paper gives a general methodology for deriving sound deep induction rules for proper (i.e., non-trivially term-indexed) IFs. Both our methodology and the sound deep induction rules it delivers generalize not just deep induction for type-indexed data types such as ADTs, nested types, and GADTs, but also structural induction for IFs. Every specific deep induction rule derived in this paper is observed to be a concrete instance of the methodology presented here. The code accompanying this paper includes Agda implementations of all of the deep induction rules appearing here, together with proof terms that witness their soundness. It also implements the case studies from Sections 5 and 6. Our development opens the way to incorporating automatic generation of deep induction rules for IFs into proof assistants. We anticipate an abundance of practical applications of such rules should they become routinely available.

Appendix

```
-- the evenness predicates on Nats:
data even : Nat → Set where
  zeven : even zero
  sseven : even n → even (suc (suc n))

-- the sum of even Nats is even:
sumEvens : even m → even n → even (m + n)
sumEvens zeven neven = neven
sumEvens (sseven meven) neven = sseven (sumEvens meven neven)

-- the oddness predicate on Nats:
odd : Nat → Set
oddn = ¬(even n)

-- the trivial predicate on a type:
KT : a → Set
KT = const ⊤

-- identifies the singleton types ⊤ × ⊤ and ⊤
postulate
  preunit : (⊤ × ⊤) ≡ ⊤

-- the data type of leaf-labeled binary trees
data Tree (a : Set) : Set where
  leaf : a → Tree a
  node : Tree a → Tree a → Tree a

-- predicate lifting for leaf-labeled binary trees:
Tree^ : (a → Set) → Tree a → Set
Tree^ Q (leaf x) = Q x
Tree^ Q (node xs ys) = Tree^ Q xs × Tree^ Q ys

-- function that collects the label-index pairs from the vinj constuctors of a VSeq:
leaves : {xs : Vec d n} → VSeq a xs → Tree (Σ Set id × Σ Nat (Vec d))
leaves {n = n} {a = a} (vinj x xs) = leaf ((a, x), (n, xs))
leaves (vpair p sbys sczs) = node (leaves sbys) (leaves sczs)

-- apply a Vec predicate to the Vec inside such a label-index pair:
QvOnVec : ({n : Nat} → Vec d n → Set) → Σ Set id × Σ Nat (Vec d) → Set
QvOnVec Qv (_, (_, xs)) = Qv xs
```

Fig. 7. Code for VSeq application from Section 5.

```
-- construct a VSeq lifting from the hypotheses of our desired proposition,
-- which is about a certain predicate holding for all vinj leaf constructors of a given VSeq term.
mkVSeq^ : (Qv : {n : Nat} → Vec d n → Set) →
  ({m n : Nat} → (ys : Vec d m) → (zs : Vec d n) → Qv ys → Qv zs → Qv (ys ++ zs)) →
  {xs : Vec d n} → (s : VSeq a xs) → Tree^ (QvOnVec Qv) (leaves s) → VSeq^ KT Qv s
mkVSeq^ Qv pres (vinj x xs) Qvxs = (tt, Qvxs)
mkVSeq^ Qv pres (vpair refl {ys} {zs} sbys sczs) (^Qsbys, ^Qsczs) =
(KT, KT, const preunit, mkVSeq^ Qv pres sbys ^Qsbys, mkVSeq^ Qv pres sczs ^Qsczs, pres ys zs)

-- predicate transformer to extend a primitive predicate to all elements of a Vec:
all : (a → Set) → Vec a n → Set
all Q [ ] = ⊤
all Q (x :: xs) = Q x × all Q xs

-- if all elements of two Vecs satisfy a primitive predicate
-- then so do all elements of their concatenation:
allConcat : (Q : a → Set) → (xs : Vec a m) → (ys : Vec a n) → all Q xs → all Q ys → all Q (xs ++ ys)
allConcat Q [ ] ys hxs hys = hys
allConcat Q (x :: xs) ys (Qx, Qxs) Qys = (Qx, allConcat Q xs ys Qxs Qys)

-- an example of a predicate on vectors of Nats:
-- having even length and only odd entries.
eloe : Vec Nat n → Set
eloe xs = even (length xs) × all odd xs

-- this predicate is preserved under vector concatenation:
eloePres : (ys : Vec Nat m) → (zs : Vec Nat n) → eloe ys → eloe zs → eloe (ys ++ zs)
eloePres ys zs (meven, ysodd) (neven, zsodd) = (sumEvens meven neven, allConcat odd ys zs ysodd zsodd)

-- finally, we can use deep induction to prove a propositition about VSeqs:
-- if all of the vinj subterms of an VSeq have indices that are even-length Vecs with odd Nat entries
-- then the whole VSeq term does as well.
prop : {xs : Vec Nat n} → (s : VSeq a xs) → Tree^ (QvOnVec eloe) (leaves s) → eloe xs
prop s hyp = VSeqInd P hinj hpair KT eloe s (mkVSeq^ eloe eloePres s hyp)
  where
  P : {xs : Vec d n} → (Qa : a → Set) → (Qv : {n : Nat} → Vec d n → Set) → VSeq a xs → Set
  P {xs = xs} Qa Qv s = Qv xs
  --
  hinj : (x : a) → {n : Nat} → (xs : Vec d n) → (Qa : a → Set) → (Qv : {n : Nat} → Vec d n → Set) →
    Qa x → Qv xs → P Qa Qv (vinj x xs)
  hinj x xs Qa Qv Qax Qvxs = Qvxs
  --
  hpair : (p : (b × c) ≡ a) → {m n : Nat} → {ys : Vec d m} → {zs : Vec d n} →
    (sbys : VSeq b ys) → (sczs : VSeq c zs) →
    (Qa : a → Set) → (Qb : b → Set) → (Qc : c → Set) → (Qv : {n : Nat} → Vec d n → Set) →
    P Qb Qv sbys → P Qc Qv sczs →
    Qv (ys ++ zs) → P Qa Qv (vpair p sbys sczs)
  hpair p sbys sczs Qa Qb Qc Qv Psbys Psczs Qvyszs = Qvyszs
```

Fig. 8. Code for VSeq application from Section 5 (continued).

References

1. The Agda Wiki, https://wiki.portal.chalmers.se/agda
2. Bird, R., Meertens, L.: Nested datatypes. In: Mathematics of Program Construction. pp. 52–67 (1998). https://doi.org/10.1007/BFb0054285
3. Cheney, J., Hinze, R.: First-class phantom types. Tech. Rep. CUCIS TR2003-1901, Cornell University (2003)
4. Christiansen, D.: Practical Reflection and Metaprogramming for Dependent Types. Ph.D. thesis, IT University of Copenhagen (2015)
5. Coquand, T., Paulin-Mohring, C.: Inductively defined types. In: COLOG-88. pp. 50–66 (1990). https://doi.org/10.1007/3-540-52335-9_47
6. Dybjer, P.: Inductive families. Formal Aspects of Computing **6**(4), 440–465 (1994). https://doi.org/10.1007/BF01211308
7. Hinze, R.: Fun with phantom types. In: The Fun of Programming, pp. 245–262. Palgrave Macmillan (2003)
8. Idris: A language for type-driven development, https://www.idris-lang.org/
9. Johann, P., Ghiorzi, E.: (Deep) induction rules for GADTs. In: Certified Programs and Proofs. pp. 324–337 (2022). https://doi.org/10.1145/3497775.3503680
10. Johann, P., Morehouse, E.: Deep induction rules for inductive families. In: Workshop on Logic, Language, Information and Computation. pp. 21–37 (2025). https://doi.org/10.1007/978-3-031-99536-1_2
11. Johann, P., Polonsky, A.: Deep induction: Induction rules for (truly) nested types. In: Foundations of Software Science and Computation Structures. pp. 339–358 (2020). https://doi.org/10.1007/978-3-030-45231-5_18
12. McBride, C.: Dependently Typed Programs and their Proofs. Ph.D. thesis, University of Edinburgh (1999)
13. McBride, C.: Epigram: Practical programming with dependent types. In: Advanced Functional Programming. pp. 130–170 (2005). https://doi.org/10.1007/11546382_3
14. McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming **14**(1), 69–111 (2004). https://doi.org/10.1017/S0956796803004829
15. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML, Revised Edition. MIT Press (1997). https://doi.org/10.7551/mitpress/2319.001.0001
16. Norell, U.: Dependently typed programming in Agda (2008), Lecture Notes, Advanced Functional Programming Summer School
17. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1999). https://doi.org/10.1017/CBO9780511530104
18. Peyton Jones, S.L. (ed.): Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003)
19. The Rocq prover, https://rocq-prover.org
20. Schrijvers, T., Peyton Jones, S.L., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: International Conference on Functional Programming. pp. 341–352 (2009). https://doi.org/10.1145/1596550.1596599
21. Sheard, T., Pasalic, E.: Meta-programming with built-in type equality. In: Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-languages. pp. 49–65 (2008). https://doi.org/10.1016/j.entcs.2007.11.012
22. Tassi, E.: Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. In: 10th International Conference on Interactive Theorem Proving. pp. 1–18 (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.29

23. Ullrich, M.: Generating induction principles for nested induction types in MetaCoq. Bachelor thesis, Saarland University (2020)
24. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Principles of Programming Languages. pp. 224–235 (2003). https://doi.org/10.1145/604131.604150