Deep Induction for Inductive Families

Patricia Johann[1][0000−0002−8075−3904] and Edward Morehouse[2][0000−0002−9008−4660]

[1] Appalachian State University, Boone NC 28608, USA
johannp@appstate.edu
[2] edward.morehouse@gmail.com

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 [17] 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 List Int, List Bool, and 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 List 3 and List p, respectively. (Type- and) term-indexed types are supported as inductive families (IFs) [7] in dependently typed systems such as Agda [1,16], Epigram [13,14], and Idris [9].

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 algebraic data types (ADTs) and nested types; data types, like that of forests from [11], whose recursive occurrences appear below other type constructors; so-called *truly* nested types, like that of bushes from [2], whose recursive occurrences can appear below their own type constructors; and generalized algebraic data types (GADTs) [3,22], as found in Haskell and Agda. Of course, term-indexed types such as IFs can also be deep, 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 data types that allow only type indexing to those that also allow term indexing. In fact,

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 [10]. 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; as a result, 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 has been derived by instantiating our methodology. Section 5 contains an application of deep induction for IFs. 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 [10]. The methodology for deriving deep induction rules developed in this paper further extends that in [10] to IFs. The relationship between our results and those of [10,11] are discussed in detail throughout this paper.

To the best of our knowledge, other work on generating induction rules for IFs is either restricted to structural induction (see, e.g., [4,6,7,5]) or fails to adequately account for depth in term indices. For example, both [20] and [21] 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., vectors (length-indexed lists), is reduced to that 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
```

[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 results are not Agda-specific.

Since it uses a predicate P on entire lists, the data inside an element of type $\mathsf{List\,a}$ are essentially ignored by the standard structural induction rule for lists:

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

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:

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

Here, the lifting $\mathsf{List}^{\wedge}$ lifts its argument predicate Q on data of type a to a predicate on data of type $\mathsf{List\,a}$ by asserting that $\mathsf{List}^{\wedge}\,\mathsf{Q}$ holds of $\mathsf{xs} : \mathsf{List\,a}$ precisely when Q holds for every element of xs. It can be defined in Agda by:

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

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]:

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

Since the constructor pnode uses data of type $\mathsf{PTree}\,(\mathsf{a} \times \mathsf{a})$ to construct data of type $\mathsf{PTree\,a}$, the instances of PTree at various indices cannot be defined independently, and 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. The deep induction rule for perfect trees is thus:

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

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

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

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 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 deep data types. Among these are the truly nested type of bushes, and the data type of forests, which is deep but not truly nested:

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

The structural induction rule generated by Coq for forests is

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

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. The deep induction rule for forests from [11] is:

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

where

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

This rule is of much more use. The special case when Q is the constantly $\top$-valued predicate is equivalent to the expected induction rule as classically stated in Coq.

In [11], deep induction was also shown to be the key to defining *structural* induction rules for truly nested types like Bush. 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 detailed in [11], taking Q to be the constantly $\top$-valued predicate in the following deep induction rule for Bush gives the structural induction rule for Bush:

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

where

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

Deep induction has been recently extended to GADTs in [10]. A simple example of a GADT is the data type Seq of sequences:[2]

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

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 constructor inj. It can be fruitful to capture this kind of non-uniformity in the return types of GADTs' data constructors via their so-called *Henry Ford encodings* [3,8,12,18,19]. These encodings use the following equality type from Agda's standard library to, in essence, turn GADTs into nested types:

$$\mathsf{data}\;\_\equiv\_\,(\mathsf{x} : \mathsf{a}) : \mathsf{a} \to \mathsf{Set}\,\mathsf{where}\;\mathsf{refl} : \mathsf{x} \equiv \mathsf{x}$$

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:

$$\begin{aligned}&\mathsf{data}\,\mathsf{Seq}\,(\mathsf{a} : \mathsf{Set}) : \mathsf{Set}\;\mathsf{where}\\ &\quad \mathsf{inj} : \mathsf{a} \to \mathsf{Seq}\,\mathsf{a}\\ &\quad \mathsf{pair} \;\; : (\mathsf{b} \times \mathsf{c}) \equiv \mathsf{a} \to \mathsf{Seq}\,\mathsf{b} \to \mathsf{Seq}\,\mathsf{c} \to \mathsf{Seq}\,\mathsf{a}\end{aligned} \tag{4}$$

Henry Ford encodings for other GADTs are obtained similarly.

Deep induction rules for GADTs can now be defined using the lifting

$$\begin{aligned}&\equiv^{\wedge} : (\mathsf{a} \to \mathsf{Set}) \to (\mathsf{b} \to \mathsf{Set}) \to \mathsf{a} \equiv \mathsf{b} \to \mathsf{Set}\\ &\equiv^{\wedge}\;\; \mathsf{Q}\,\mathsf{Q}'\,\mathsf{refl} \;=\; (\mathsf{x} : \mathsf{a}) \to \mathsf{Q}\,\mathsf{x} \equiv \mathsf{Q}'\,\mathsf{x}\end{aligned}$$

for equality types, together with existentially quantified predicates[3] and the original methodology for nested types, to define their predicate liftings via their Henry Ford encodings. This approach gives the following lifting $\mathsf{Seq}^{\wedge}$ for Seq:

$$\begin{aligned}&\mathsf{Seq}^{\wedge} : (\mathsf{a} \to \mathsf{Set}) \to \mathsf{Seq}\,\mathsf{a} \to \mathsf{Set}\\ &\mathsf{Seq}^{\wedge}\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{inj}\,\mathsf{x}) = \mathsf{Q}_{\mathsf{a}}\,\mathsf{x}\\ &\mathsf{Seq}^{\wedge}\,\mathsf{Q}_{\mathsf{a}}\,(\mathsf{pair}\,\mathsf{p}\,\mathsf{s}_{\mathsf{b}}\,\mathsf{s}_{\mathsf{c}}) = \exists[\mathsf{Q}_{\mathsf{b}}]\exists[\mathsf{Q}_{\mathsf{c}}]\;\equiv^{\wedge}(\times^{\wedge}\,\mathsf{Q}_{\mathsf{b}}\,\mathsf{Q}_{\mathsf{c}})\,\mathsf{Q}_{\mathsf{a}}\,\mathsf{p} \times \mathsf{Seq}^{\wedge}\,\mathsf{Q}_{\mathsf{b}}\,\mathsf{s}_{\mathsf{b}} \times \mathsf{Seq}^{\wedge}\,\mathsf{Q}_{\mathsf{c}}\,\mathsf{s}_{\mathsf{c}}\end{aligned} \tag{5}$$

The lifting for Seq introduces new predicates on the new types introduced by its 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 Q_{b} and Q_{c} on the types b and c introduced by Seq's Henry Ford encoding, and then ensures that $\mathsf{Q}_{\mathsf{b}} \times \mathsf{Q}_{\mathsf{c}}$ and Q_{a} are equal. Otherwise it simply performs the usual two tasks of liftings, namely (i) ensuring

[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.

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

that any new primitive data 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. The deep induction rule for Seq is:

$$\begin{aligned}
&(\mathsf{P} : \{\mathsf{a} : \mathsf{Set}\} \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{Seq}\,\mathsf{a} \to \mathsf{Set}) \to \\
&\quad (\{\mathsf{a} : \mathsf{Set}\} \to (\mathsf{x} : \mathsf{a}) \to (\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to \mathsf{Q_a}\,\mathsf{x} \to \mathsf{P}\,\mathsf{Q_a}\,(\mathsf{inj}\,\mathsf{x})) \to \\
&\quad (\{\mathsf{a}\,\mathsf{b}\,\mathsf{c} : \mathsf{Set}\} \to (\mathsf{p} : (\mathsf{b} \times \mathsf{c}) \equiv \mathsf{a}) \to (\mathsf{s_b} : \mathsf{Seq}\,\mathsf{b}) \to (\mathsf{s_c} : \mathsf{Seq}\,\mathsf{c}) \to (\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to \\
&\qquad (\mathsf{Q_b} : \mathsf{b} \to \mathsf{Set}) \to (\mathsf{Q_c} : \mathsf{c} \to \mathsf{Set}) \to \mathsf{P}\,\mathsf{Q_b}\,\mathsf{s_b} \to \mathsf{P}\,\mathsf{Q_c}\,\mathsf{s_c} \to \mathsf{P}\,\mathsf{Q_a}\,(\mathsf{pair}\,\mathsf{p}\,\mathsf{s_b}\,\mathsf{s_c})) \to \\
&\quad (\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to (\mathsf{s} : \mathsf{Seq}\,\mathsf{a}) \to \mathsf{Seq}^{\wedge}\,\mathsf{Q_a}\,\mathsf{s} \to \mathsf{P}\,\mathsf{Q_a}\,\mathsf{s}
\end{aligned} \tag{6}$$

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 on the types of its term indices as well. 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.

We do exactly this in this paper. Moreover, we account for term indices in such a way that the deep induction rules for IFs we develop specialize to the rules of [10] 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 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 [7], and properly extend the deep induction rules for IFs in [21], 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 parameterizing predicates (on *both* the type and term indices) 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 (6) 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 this end, consider the proper IF Vec defining the data type of vectors over a type a taken (essentially) from Agda's standard library:

```
data Vec (a : Set) : Nat → Set where          data Nat : Set where
  [ ] : Vec a zero                              zero : Nat
  _::_ : a → Vec a n → Vec a (suc n)            suc : Nat → Nat
```
(7)

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 illustrated 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 that of also (iii) ensuring that the[4] 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 following predicate lifting and deep induction rule:

```
Vec^ : (a → Set) → (Nat → Set) → Vec a n → Set
Vec^ {n = zero} Qa Qn [ ] = Qn zero
Vec^ {n = suc m} Qa Qn (x :: xs) = Qa x × Vec^ Qa Qn xs × (Qn m → Qn (suc m))

(Qa : a → Set) → (Qn : Nat → Set) → (P : {n : Nat} → Vec a n → Set) →
  (Qn zero → P [ ]) →
  ({n : Nat} → (x : a) → (xs : Vec a n) → Qa x → P xs → Qn (suc n) → P (x :: xs)) →
  (xs : Vec a n) → Vec^ Qa Qn xs → P xs
```
(8)

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

[4] 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 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 [10,11] 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 presevation 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 [10,11] for IFs that are type-indexed only.

Since our methodology will handle an IF's type indices as they are handled in [10,11], 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 [10], namely:

$$\mathsf{data}\ \mathsf{G} : \mathsf{Set}^{\alpha} \to \mathsf{Set}\ \mathsf{where}\ \mathsf{c} : \forall\{\overline{\mathsf{B} : \mathsf{Set}}\} \to \mathsf{F}\,\mathsf{G}\,\overline{\mathsf{B}} \to \mathsf{G}(\overline{\mathsf{K}\,\overline{\mathsf{B}}}) \tag{9}$$

For brevity and clarity we indicate only one data constructor c in (9), even though a GADT can have any finite number of them, each with a type of the same form as c's. In (9), 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 exactly 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 from [10]:

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

As in [10], 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 [10] for details. In the fifth and sixth clauses, H 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 [10] guarantees that the techniques of that paper apply to the type indices both of the GADT underlying

an IF and of the GADT underlying the 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 [10,11], and thus at a uniform methodology for deriving deep induction rules for such IFs.

Given an IF D, a predicate on the type of its term indices, and predicates on each of the primitive types appearing in D, the lifting $\mathsf{D}^\wedge$ for D includes one clause for each of its data constructors. The clause for the data constructor c is constructed via the steps described below. We illustrate each step using the fpair constructor for the IF FSeq from Figure 1. 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 its type indices; see [10] for details. The following steps can then be taken directly for that converted IF, exactly as illustrated below. The liftings from [10] can be newly understood as liftings that accomplish all three of the tasks below for GADTs (the last trivially), so our methodology for IFs subsumes that for GADTs in [10] (and, hence, that for nested types in [11]) when no term indices are present.

The clause of $\mathsf{D}^\wedge$ for a data constructor c of an IF D is constructed as follows:

1. Check that all non-recursive data used by c to construct elements of D satisfy the liftings for their types of the given predicates on D's type indices. (But in our examples we omit checks for term indices here when they already arise in checks in Steps 2 and 3.) In the definition of FSeq, e.g., 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 $\equiv^\wedge (\times^\wedge \mathsf{Q_b}\, \mathsf{Q_c})\, \mathsf{Q_a}\, \mathsf{p}$.
2. Check that all recursive subdata of the element of D that c constructs satisfy the lifting being defined of the predicates on D's type indices and the type of its term indices. In the definition of FSeq, e.g., 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., 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 $\mathsf{Q_f\, i} \to \mathsf{Q_f\, j} \to \mathsf{Q_f}\, (\mathsf{i} +_\mathsf{f} \mathsf{j})$.

Altogether this gives the clause of $\mathsf{D}^\wedge$ for c. The clause of $\mathsf{FSeq}^\wedge$ for fpair, e.g., is:

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

Once we have its lifting, the deep induction rule for D 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 predicates on D's

```
data Fin : Nat → Set where
  fz : Fin (suc n)
  fs : Fin n → Fin (suc n)

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)

FSeq^ : (a → Set) → ({n : Nat} → Fin n → Set) → {i : Fin n} → FSeq a i → Set
FSeq^ Qa Qf (finj x i) = Qa x × Qf i
FSeq^ Qa Qf (fpair p {i} {j} sbi scj) = ∃ [Qb] ∃ [Qc] ≡^ (×^ Qb Qc) Qa p × FSeq^ Qb Qf sbi ×
                                        FSeq^ Qc Qf scj × (Qf i → Qf j → Qf (i +f j))

(P : {a : Set} → {n : Nat} → {i : Fin n} →
   (Qa : a → Set) → (Qf : {n : Nat} → Fin n → Set) → FSeq a i → Set) →          (Step 1)
 ({a : Set} → (x : a) → {n : Nat} → (i : Fin n) → (Qa : a → Set) →
   (Qf : {n : Nat} → Fin n → Set) → Qa x → Qf i → P Qa Qf (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) → (Qa : a → Set) → (Qb : b → Set) →
   (Qc : c → Set) → (Qf : {n : Nat} → Fin n → Set) →
   P Qb Qf sbi → P Qc Qf scj → Qf (i +f j) → P Qa Qf (fpair p sbi scj)) →      (Step 2)
 (Qa : a → Set) → (Qf : {n : Nat} → Fin n → Set) → {i : Fin n} → (s : FSeq a i) →
   FSeq^ Qa Qf s → P Qa Qf s                                                   (Step 3)
```

Fig. 1. Deep induction rule for Fin-indexed sequences.

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

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

that is parameterized by a predicate $\mathsf{Q_a}$ on the primitive type a and a predicate $\mathsf{Q_f}$ on the type Fin n of term indices appearing in FSeq's definition. Similarly, the first input to the deep induction rule for Vec is a predicate P of type $\{\mathsf{a} : \mathsf{Set}\} \to \{\mathsf{n} : \mathsf{Nat}\} \to (\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to (\mathsf{Q_n} : \mathsf{Nat} \to \mathsf{Set}) \to \mathsf{Vec}\,\mathsf{a}\,\mathsf{n} \to \mathsf{Set}$ that is parameterized by predicates $\mathsf{Q_a}$ on the primitive type a and $\mathsf{Q_n}$ on the type Nat of term indices appearing in Vec's definition. However, in this case 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 (8). This simplification can be applied to the deep induction rules for other IFs, such as FSeq in Figure 1, as well.

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 index appearing in the clause of $\mathsf{D}^\wedge$ 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.

The induction hypothesis for fpair in the deep induction rule for FSeq, e.g., is:

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

3. Conclude that, given an arbitary element of D and the ingredients needed to construct it, if the element satisfies D's lifting of P's parameterizing predicates then it satisfies P. For example, the conclusion for FSeq is:

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

Exactly this methodology has yielded all of the deep induction rules in this paper. The accompanying code file contains additional examples illustrating our methodology (but note that when the term index of an IF is given by a GADT, we can choose to obtain the predicate on it by lifting predicates on its type indices rather than giving it directly). 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). Due to space limitations, only the first and final two of these examples appear in the text of this paper, in (8) and Figures 1 and 2, respectively.

We call to attention some particular features of our methodology. Firstly, as in [10,11], there is no need to reflect predicates as data types. Secondly, 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 [10] but obtains (for type indices) there as well. Thirdly, the previous point is in stark contrast to the methods of [20,21], 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 [21] the deep induction rule for vectors is reduced to that for lists. Finally, 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

We now show how deep induction can be used to prove a non-trivial property of GADTs indexed by terms of an IF that is deep in its own type indices. Let _++_ be Agda's vector concatenation, and consider the type VSeq of vector-indexed sequences analogous to the type FSeq of finite-set-indexed sequences in Figure 1:

```
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 lifting and deep induction rule for VSeq are shown in Figure 2 in the appendix. We can use the latter to prove that, for every xs : Vec Nat n and every s : VSeq a xs for some a : 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 use the predicate eloe : Vec Nat n → Set defined by eloe xs = even (length xs) × all odd xs, where 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:

```
{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$^\wedge$ checks that every datum of in a tree satisfies a given predicate on the type of its elements, 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[5], and 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. Now, in order to use the deep induction rule for VSeq to prove prop, we need to construct a term of type VSeq$^\wedge$ KT eloe s from prop's argument of type Tree$^\wedge$(QvOnVec eloe) (leaves s). The function

```
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
```

does exactly this.

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

The code for the complete application appears in the appendix, and is also included in the code file accompanying this paper. How to use deep induction rules to prove properties of more general IFs should be apparent.

[5] 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 [10].

Appendix

– The IF 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 lifting for 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 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
VSeqInd P hinj hpair Qa Qv (vinj x xs) (Qax, Qvxs) = hinj x xs Qa Qv Qax Qvxs
VSeqInd P hinj hpair Qa Qv seq@(vpair p sbys sczs) lft@(Qb, Qc, e, ^Qsbys, ^Qsczs, hQv) =
  hpair p sbys sczs Qa Qb Qc Qv
    (VSeqInd P hinj hpair Qb Qv sbys ^Qsbys)
    (VSeqInd P hinj hpair Qb Qv sczs ^Qsczs)
    (QvOnIndex Qa Qv seq lft)
  where
    QvOnIndex : {a d : Set} → {n : Nat} → {xs : Vec d n} →
      (Qa : a → Set) → (Qv : {n : Nat} → Vec d n → Set) →
      (s : VSeq a xs) → VSeq^ Qa Qv s → Qv xs
    QvOnIndex Qa Qv (vinj x xs) (Qax, Qvxs) = Qvxs
    QvOnIndex Qa Qv (vpair x sbys scjs) (Qb, Qc, e, ^Qsbys, ^Qsczs, hQv) =
      hQv (QvOnIndex Qb Qv sbys ^Qsbys) (QvOnIndex Qc Qv scjs ^Qsczs)
```

Fig. 2. Deep induction rule for VSeq.

```
– 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

– 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)
```

Fig. 3. Code for Section 5.

```
– predicate transformer to extend an element 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 an element 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. 4. Code for Section 5 (continued).

References

1. The Agda Wiki, https://wiki.portal.chalmers.se/agda/pmwiki.php
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. Christensen, D.: Practical Reflection and Metaprogramming for Dependent Types. Ph.D. thesis, IT University of Copenhagen (2015)
5. Coq Development Team: The Coq proof assistant, version 8.19.2 (2024)
6. 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
7. Dybjer, P.: Inductive families. Formal Aspects of Computing **6**(4), 440–465 (1994). https://doi.org/10.1007/BF01211308
8. Hinze, R.: Fun with phantom types. In: The Fun of Programming, pp. 245–262. Palgrave Macmillan (2003)
9. Idris: A language for type-driven development, https://www.idris-lang.org/
10. 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
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. Peyton Jones, S.L. (ed.): Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003)
18. 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
19. 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
20. 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
21. Ullrich, M.: Generating induction principles for nested induction types in MetaCoq. Bachelor thesis, Saarland University (2020)
22. 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