Deep Induction for Inductive-Inductive Types

Patricia Johann[0000-0002-8075-3904] and Ben Lenox[0009-0007-9037-029X]

Appalachian State University, Boone NC 28608, USA
johannp@appstate.edu lenoxbe@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 was originally defined for type-indexed types such as ADTs, nested types, and GADTs, but has recently been extended to the term-indexed types known as inductive families. Inductive-inductive types syntactically generalize the class of IFs whose indexing types are inductive by allowing their (still necessarily inductive) indexing types to be defined by mutual induction with the type being indexed. In this paper we show how to further extend deep induction to inductive-inductive types. More specifically, we extend to inductive-inductive types the entire methodology for deriving sound deep induction rules for inductive families, which itself extends that originally developed for nested types and subsequently extended to GADTs. We also include two non-trivial applications illustrating the usefulness of deep induction for inductive-inductive types.

Keywords: Deep induction · inductive-inductive types · 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., functional languages such as Haskell [22] and ML [17]. 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 $\mathsf{List}\,\mathsf{Int}$, $\mathsf{List}\,\mathsf{Bool}$, and $\mathsf{List}\,(\mathsf{List}\,\mathsf{t})$, respectively. More modern programming languages allow types to be indexed not just by types, but also by terms. The essential idea is that a list of type List, e.g., can be indexed by a term that represents its length or a proof that it satisfies some property. For instance, lists of length $\mathsf{3}$ and lists that a proof term p proves are sorted can be modeled by term-indexed types such as $\mathsf{List}\,\mathsf{3}$ and $\mathsf{List}\,\mathsf{p}$, respectively. In both of these examples, the indexed type is the type List of lists, whereas the indexing type, whose terms index lists, is the type of natural numbers for the former and the type of proofs for the latter. (Type- and) term-indexed types are supported as inductive families (IFs) [10] in dependently typed systems such as Agda [1,20], Epigram [15,16], and Idris [11].

Inductive-inductive types (IITs) [18,19] syntactically generalize the class of IFs whose indexing types are inductive. But whereas such an IF's indexing type must *already exist in its entirety* before its indexed type is defined, an IIT allows its indexing and indexed types to be defined simultaneously by *mutual induction.* IITs are thus simultaneously *less* general than arbitrary IFs because the indexing types of IITs are required to be inductive while those of arbitrary

IFs are not, but also *more* general than IFs whose indexing types are inductive because those indexing types can be defined by mutual induction with the types they index. Importantly, every IF whose indexing type is inductive can trivially be seen as an IIT. IITs have been used, e.g., to define the syntax of Martin-Löf Type Theory in itself [3,9] and to model Conway's surreal numbers [6,19].

Deep induction was introduced in [14] 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 type-indexed data types that can be deep include, trivially, ordinary algebraic data types (ADTs) and nested types; data types, like that of forests from [14] (also called rose trees in [12]), whose recursive occurrences appear below other type constructors; so-called *truly* nested types, like that of bushes from [2] (also called bootstrapped heaps in [21]), whose recursive occurrences can appear below their own type constructors; and generalized algebraic data types (GADTs) [4,25], such as are found in Haskell and Agda. Term-indexed types like IFs and IITs can also be deep, both in their type indices and in the types of their term indices. Since the structural induction rules currently generated by proof assistants for deep data types induct only over their top-level structures, leaving any data internal to that top-level structure untouched, 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, and thus opens the way for incorporating automatic generation of truly useful induction rules for deep data types — including deep IFs and IITs — into state-of-the-art proof assistants.

The recent paper [13] showed how deep induction can be extended from data types that allow type indexing only to those that also allow term indexing. In fact, the *entire methodology* for deriving sound deep induction rules that was developed for nested types in [14], and extended to GADTs in [12], was further extended to IFs in [13]. As observed there, an IF's indexed type can always be seen as an underlying (type-indexed) GADT obtainable by erasing all of the IF's term indices, together with an indexing type (which can in general also be an IF) whose terms index that underlying type. The key innovation required to extend deep induction to IFs in [13] was therefore to appropriately track satisfaction of predicates for a term-indexed type's term indices.

In this paper we show how deep induction can further be extended, now from IFs whose indexing types are inductive all the way to IITs. The mutual induction in a *proper* IIT (i.e., in an IIT that cannot be seen as an IF) entails that there is no separation of its indexing and indexed types — and thus no concept of an underlying data type for it — as there is for IFs. As we show below, the main task in extending deep induction to proper IITs is thus accommodating this inseparability of their indexing and indexed types. Along the way we also observe that more predicates can usefully be propagated over an IIT's depth than can be for IFs. As in [13], we again extend the *entire methodology* for deriving sound deep induction rules for the tower of "simpler" classes of data types — ADTs, nested types, and GADTs, and IFs whose indexing types are inductive — to also include IITs. The upshot is that deep induction for each of the simpler classes of data types in this tower can be seen as a special case of deep induction as developed here for IITs.

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 introduces IITs via three canonical examples from the literature. Section 3 reviews the current state of the art for deep induction for IFs. Proper IFs involve term-indexing, so they are not simply (type-indexed) GADTs. But their (inductive) indexing and indexed types are not mutually defined, so they are not proper IITs either. In Section 4 we present our general methodology for deriving deep induction rules for proper IITs. The deep induction rules our methodology delivers generalize those for IFs with inductive indexing types, which are the only kind of IFs we consider in the remainder of this paper. Each concrete instance of a deep induction rule appearing in this paper — including those given in Sections 5 and 6 for the IITs of contexts and types and of sorted lists, respectively — is derivable by instantiating the methodology in Section 4 to the data type of interest. Sections 5 and 7 contain applications of deep induction for the IITs of contexts and types and of sorted lists, respectively, which are defined in Section 2 below. Section 8 concludes and offers directions for future work. Our Agda implementation containing all of the deep induction rules appearing in this paper, proof terms that witness their soundness, and our applications of deep induction from Sections 5 and 7 is available at `https://cs.appstate.edu/johannp/`.

Related Work Deep induction was introduced for nested types in [14], extended to GADTs in [12], and further extended to IFs in [13]. The methodology for deriving deep induction rules developed in this paper further extends that in [13] from (inductively indexed) IFs to IITs. The relationship between our results and those of [12,13,14] are discussed in detail throughout this paper. To the best of our knowledge, other work on generating induction rules for IITs is either restricted to structural induction (see, e.g., [5,7,8,10,18,19]) or fails to adequately account for depth in term indices. For example, both [23] and [24] derive induction rules that are deep for nested types and some IFs 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 (i.e., length-indexed lists), is reduced to that for their underlying lists. Structural induction for IITs that are not IFs is treated in [18,19]. As far as we are aware, deep induction for such IITs has not previously been considered.

2 Inductive-inductive Types

To illustrate the difference between type-indexed types, the term-indexed types known as IFs, and IITs, first consider the following familiar ADT of lists:[1]

```
data List (a : Set) : Set where
  [] : List a                                    (1)
  _::_ : a → List a → List a
```

Here the type List is indexed over the type a of data elements that each list in List a contains. A variant of the type List a that carries more information than just

[1] We use Agda syntax for concreteness, but our results are not Agda-specific. We use Agda's facility for generalizing declared variables, so that, throughout the paper, implicit occurrences of a and b have type Set, and those of m and n have type Nat.

the type of the elements each of its lists contains is the following proper IF Vec defining the data type of vectors taken (essentially) from Agda's standard library:

```
data Nat : Set where          data Vec (a : Set) : Nat → Set where
  zero : Nat                    vnil : Vec a zero                          (2)
  suc : Nat → Nat               vcons : a → Vec a n → Vec a (suc n)
```

The data type Vec cannot be seen as an ADT. Not only is Vec indexed by the *type* a of data elements that each list in List a contains, but each of its elements is also indexed by the *term* n : Nat that is its length. For example, the vector vcons True (vcons False (vcons True vnil)) is an element of Vec Bool (suc (suc (suc zero))). Since the data type underlying Vec — i.e., the data type obtained from Vec by erasing its term indices — is the ADT List from (1), we can think of a vector of type Vec a n as a list of type List a that is indexed by its length.

The type Vec a n is a proper IF, but it is not a proper IIT: its indexing type Nat, while inductive, is not defined mutually with its indexed type. An example of a proper IIT is the following IIT of contexts and types from [19]. This IIT, which is term-indexed but not type-indexed, may well be the most important example of a proper IIT from the perspective of dependently typed programming. It has been used, e.g., in [3,9] to define the syntax of Martin-Löf Type Theory in itself. Its definition is:

```
mutual
  data Ctxt : Set where
    ε : Ctxt
    _·_ : (Γ : Ctxt) → Ty Γ → Ctxt                                (3)
  data Ty : Ctxt → Set where
    ι : {Γ : Ctxt} → Ty Γ
    Π : {Γ : Ctxt} → (A : Ty Γ) → Ty (Γ · A) → Ty Γ
```

The definition of Ctxt says we have an empty context ε, and if we have any context Γ and a type A valid in Γ, then we can extend Γ with a fresh variable x : A to get a new context Γ · A. The definition of Ty says that we have a base type ι that is valid in any context, and if A is a type valid in context Γ and B is a type valid in context Γ · A, then the dependent function type Π A B is also valid in context Γ. For the IIT of contexts and types, Ctxt is the indexing type and Ty, which takes elements of Ctxt as arguments, is the indexed type.

Another example of a proper IIT is the following type- and term-indexed IIT of (so-called) sorted lists:

```
mutual
  data SList (a : Set) {{orda : Ordered a}} : Set where
    snil : SList a
    scons : {x : a} → {xs : SList a} → x ≥L xs → SList a
                                                                      (4)
  data _≥L_ {a : Set} {{orda : Ordered a}} (x : a) : SList a → Set where
    triv : x ≥L snil
    extn : {y : a} → {ys : SList a} → (y≥ys : y ≥L ys) → x ≥ y →
                x ≥L ys → x ≥L (scons y≥ys)
```

This IIT is both a small variation on, and a generalization of, the IIT of sorted lists given in [18,19]. It is a variation because it uses $\geq$ rather than $\leq$ as in [18,19], and it is a generalization because it allows list elements to be not just natural numbers, but rather data from an arbitrary *ordered type* a (following the nomenclature of [18,19]), i.e., from any instance of the class

```
record Ordered (a : Set) : Set where
  field
    _≥_ : a → a → Set
```

of types supporting an arbitrary binary relation $\geq$. This binary relation is used in the clause for extn of the indexed type $\geq_L$ in (4). Given an ordered type a, the definition of SList says that the empty list is trivially sorted, and that if we have a proof x≥xs that $x \geq_L xs$ — i.e., that x : a is greater than or equal to every element of the sorted list xs of elements of type a — then we can add x to the front of xs to get a new sorted list scons x≥xs of elements of type a. Given an ordered type a, the definition of $\geq_L$ says that every element x : a is trivially larger than every element of the empty list, and that if y : a and ys : SList a are such that $x \geq y$ in a and, inductively, $y \geq_L ys$ with proof y≥ys, then $x \geq_L$ scons y≥ys. Note that the type of extn makes sense even if the ordering $\geq$ on type a is not transitive, although the type of extn can be simplified when $\geq$ is a partial or total order.

While it is very natural to think of an element of type SList a as a list of type List a that is indexed by the ordering $\geq_L$ defined mutually with it, a glance at the types of SList and $\geq_L$ reveals that it is actually the type $\geq_L$ that is indexed by terms of type SList. Properly identifying the indexing and indexed types of IITs will be critical when we develop our deep induction rules for them in Section 4.

Our final example of a proper IIT is that of dense order completions [18,19]:

```
mutual
  data (−)* (a : Set) {{orda : TOrdered a}} : Set where
    inj : a → a*
    mid : {x y : a*} → x <* y → a*
                                                                     (5)
  data _<*_ {a : Set} {{orda : TOrdered a}} : a* → a* → Set where
    inj< : {x y : a} → x < y → inj x <* inj y
    l<mid : {x y : a*} → (x<*y : x <* y) → x <* (mid x<*y)
    mid<r : {x y : a*} → (x<*y : x <* y) → (mid x<*y) <* y
```

Here, a is an instance of the class

```
record TOrdered (a : Set) : Set where
  field
    _<_ : a → a → Set
    <trans : {x y z : a} → x < y → y < z → x < z
```

of types that support a transitive ordering $<$. Given a type a that supports a transitive ordering, the definition of $(-)^*$ says that an element of a^* is either an injected element of a, or it is the midpoint between elements x and y already in a^* and such that x is less than y in the extension $<^*$ of $<$ to a^*. Given a type a that supports a transitive ordering, the definition of the extension $<^*$ of $<$ to a^* says that if $\mathsf{x} < \mathsf{y}$ in a then $\mathsf{inj}\,\mathsf{x} <^* \mathsf{inj}\,\mathsf{y}$ in a^*, and that if x and y are already elements of a^* and if $\mathsf{x{<}^*y}$ proves $\mathsf{x} <^* \mathsf{y}$, then $\mathsf{x} <^* (\mathsf{mid}\,\mathsf{x{<}^*y})$ and $(\mathsf{mid}\,\mathsf{x{<}^*y}) <^* \mathsf{y}$ in a^*.

The general form for IITs is given in [18,19]. There, a (finite) axiomatization of IITs is specified, together with a model that proves their elimination rules (i.e., their structural induction rules) sound. In this paper we restrict our attention to IITs whose indexing and indexed types are given by inductive GADTs of the form specified in [12]. Note that each of the three IITs above is so given.

3 The State-of-the-Art: Deep Induction for IFs

To see the difference between structural induction and deep induction, consider again the data type of lists from (1). The structural induction rule for lists is:

$$\begin{array}{l}(\mathsf{P} : \mathsf{List}\,\mathsf{a} \to \mathsf{Set}) \to \mathsf{P}\,[\,] \to \\ \;\;((\mathsf{x} : \mathsf{a}) \to (\mathsf{xs} : \mathsf{List}\,\mathsf{a}) \to \mathsf{P}\,\mathsf{xs} \to \mathsf{P}\,(\mathsf{x} :: \mathsf{xs})) \to \\ \;\;(\mathsf{xs} : \mathsf{List}\,\mathsf{a}) \to \mathsf{P}\,\mathsf{xs}\end{array} \tag{6}$$

Since it uses a predicate P on entire lists, this rule essentially ignores the data inside an element of type $\mathsf{List}\,\mathsf{a}$. 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:

$$\begin{array}{l}(\mathsf{Q} : \mathsf{a} \to \mathsf{Set}) \to (\mathsf{P} : \mathsf{List}\,\mathsf{a} \to \mathsf{Set}) \to \mathsf{P}\,[\,] \to \\ \;\;((\mathsf{x} : \mathsf{a}) \to (\mathsf{xs} : \mathsf{List}\,\mathsf{a}) \to \mathsf{Q}\,\mathsf{x} \to \mathsf{P}\,\mathsf{xs} \to \mathsf{P}\,(\mathsf{x} :: \mathsf{xs})) \to \\ \;\;(\mathsf{xs} : \mathsf{List}\,\mathsf{a}) \to \mathsf{List}^\wedge\,\mathsf{Q}\,\mathsf{xs} \to \mathsf{P}\,\mathsf{xs}\end{array} \tag{7}$$

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

$$\begin{array}{l}\mathsf{List}^\wedge : (\mathsf{a} \to \mathsf{Set}) \to \mathsf{List}\,\mathsf{a} \to \mathsf{Set} \\ \mathsf{List}^\wedge\,\mathsf{Q}\,[\,] \;=\; \top \\ \mathsf{List}^\wedge\,\mathsf{Q}\,(\mathsf{x} :: \mathsf{xs}) \;=\; \mathsf{Q}\,\mathsf{x} \times \mathsf{List}^\wedge\,\mathsf{Q}\,\mathsf{xs}\end{array} \tag{8}$$

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.

It is well known that the class of ADTs is subsumed by that of nested types, and that the class of nested types is in turn subsumed by that of GADTs. Moreover, just as structural induction can be extended to nested types and GADTs, so can deep induction [12,14]. However, as detailed in these works, *polymorphic* predicates are needed to derive either kind of induction rules for either nested types or GADTs. GADTs require, in addition, the use of Henry Ford encodings to handle the non-uniformity in their type indices [12]. These two ideas can be used together to derive deep induction rules for very general type-indexed types, such as the GADT LTerm of typed lambda terms from [12]. In addition to providing clean solutions to complex applications involving GADTs, deep induction has been used in [14] to solve the long-open problem of giving *structural* induction rules for truly nested types, such as the data type of bushes from [2]. It has also been shown in [12] to give induction rules for deep GADTs, like LTerm, that are more useful than the induction rules for them automatically derived by Rocq.

More directly relevant to the present paper is the extension of deep induction from GADTs to IFs recently reported in [13]. Since an IF allows term indices as well as type indices, its deep induction rule must take as input not only predicates on its type indices, but also a predicate on the type of its term indices.[2] Ultimately, all of these predicates must be appropriately propagated to all of the data of their domain types in all of the IF's data elements.

The predicate lifting for an IF performs exactly this propagation. As elaborated in [13], an IF's lifting must perform three tasks. It must (i) ensure that any new data used to construct a data element satisfy the predicates on their types that either parameterize the clauses of D's lifting or are obtained by lifting these predicates, (ii) ensure that all of the data element's recursive subdata also satisfy the liftings for their types of the predicates from (i), and (iii) ensure that the term index of every data element constructed using a data constructor of D's indexed type satisfies its predicate from (i) provided the term indices of that element's recursive subdata do. Performing each of these tasks for each data constructor results in the following predicate lifting for the IF Vec from (2):

$$\begin{array}{l}
\mathsf{Vec}^\wedge : (\mathsf{a} \to \mathsf{Set}) \to (\mathsf{Nat} \to \mathsf{Set}) \to \mathsf{Vec\,a\,n} \to \mathsf{Set} \\
\mathsf{Vec}^\wedge\,\{\mathsf{n} = \mathsf{zero}\}\,\mathsf{Q_a}\,\mathsf{Q_N}\,\mathsf{vnil} \;=\; \mathsf{Q_N}\,\mathsf{zero} \\
\mathsf{Vec}^\wedge\,\{\mathsf{n} = \mathsf{suc\,m}\}\,\mathsf{Q_a}\,\mathsf{Q_N}\,(\mathsf{vcons\,x\,xs}) \;=\; \mathsf{Q_a\,x} \times \mathsf{Vec}^\wedge\,\mathsf{Q_a}\,\mathsf{Q_N}\,\mathsf{xs} \times (\mathsf{Q_N m} \to \mathsf{Q_N}(\mathsf{suc\,m}))
\end{array} \tag{9}$$

The right-hand side of the clause for vnil comes from (iii). The components of the right-hand side of the clause for vcons come from (i), (ii), and (iii), respectively.

The deep induction rule for any IF can now be obtained as in [13] or, equivalently, by specializing the construction for IITs given in Section 4 to IFs, with the IF playing the role of an IIT's indexed type. For Vec, e.g., doing either yields the following deep induction rule:

$$\begin{array}{l}
(\mathsf{Q_a} : \mathsf{a} \to \mathsf{Set}) \to (\mathsf{Q_N} : \mathsf{Nat} \to \mathsf{Set}) \to (\mathsf{P} : \{\mathsf{n} : \mathsf{Nat}\} \to \mathsf{Vec\,a\,n} \to \mathsf{Set}) \to \\
\quad (\mathsf{Q_N\,zero} \to \mathsf{P\,vnil}) \to \\
\quad (\{\mathsf{n} : \mathsf{Nat}\} \to (\mathsf{x} : \mathsf{a}) \to (\mathsf{xs} : \mathsf{Vec\,a\,n}) \to \mathsf{Q_N}\,(\mathsf{suc\,n}) \to \mathsf{Q_a\,x} \to \mathsf{P\,xs} \to \mathsf{P}\,(\mathsf{vcons\,x\,xs})) \to \\
\quad (\mathsf{xs} : \mathsf{Vec\,a\,n}) \to \mathsf{Vec}^\wedge\,\mathsf{Q_a}\,\mathsf{Q_N}\,\mathsf{xs} \to \mathsf{P\,xs}
\end{array} \tag{10}$$

With respect to the construction on page 9, the first line comes from Steps 1 and 2[3]; the second and third are the induction hypotheses for vnil and vcons, respectively, from Step 3; and the fourth is from Step 4. In the induction hypothesis for vnil, the first type is from Step 3b (because zero is the term index of vnil) no types come from Steps 3a or 3c, and the conclusion is from Step 3d. In the induction hypothesis for vcons, the first three types are from Step 3a, the fourth and fifth are from Step 3b, the sixth is from Step 3c, and the conclusion is from Step 3d.

[2] For ease of exposition we assume throughout that IFs and IITs have exactly one term index. Generalizing to more than one term index is straightforward, if tedious.

[3] Since Vec is only trivially mutually defined with Nat, the counterpart to P for Nat is not needed to define P or the deep induction rule that proves it. It is thus omitted. Other simplifications are also made to arrive at the exact the rule (10) from [13].

Just as the deep induction rule for any GADT specializes to its structural induction rule when the predicates on its type indices are constantly $\top$-valued, so the deep induction rule for any IF developed in [13] specializes to its structural induction rule from [10] when, in addition, the predicate on the type of its term indices is constantly $\top$-valued. Moreover, the deep induction rules — and thus the structural induction rules — for IFs conservatively extend the corresponding rules for GADTs, which in turn conservatively extend those for nested types, which themselves conservatively extend those for ADTs.

4 Deep Induction Rules for IITs

We now show how to extend deep induction from IFs to IITs. Unlike IFs, whose indexing type must already exist in its entirety before the type it indexes is defined, IITs allow mutual definition of their indexing and indexed types. Modifying the construction from [13] to account for the mutuality of the definitions of an IIT's indexing type and its indexed type is subtle and requires care. This is in part because of the mutuality itself, but also because the mutuality entails that a larger class of predicates can be lifted to, and thus can parameterize deep induction rules for, IITs than IFs. The identification and appropriate handling of these subtleties is the central contribution of this paper.

We construct our deep induction rules for IITs in such a way that they specialize to the rules of [13] for those IITs that can be seen as IFs (and thus specialize to the rules of [12] for those IFs that can be seen as GADTs, etc.). Such specialization is a minimal success criterion for the rules we construct for IITs because it ensures that our methodology for producing them is a conservative extension of all those that have come before. A second success criterion is that the deep induction rules we construct for IITs specialize to the structural induction rules of [18,19] for them. This is shown, as usual, by taking all predicates parameterizing the clauses of an IIT's deep induction rule for its indexing and indexed types to be constantly $\top$-valued. It is critical because the structural induction rule for any data type should always be a special case of its deep induction rule.

The lifting for an IIT D has a clause for D's indexing type and a clause for D's indexed type. These are defined by mutual induction. Each clause of D's lifting is parameterized over a predicate P on D's indexed type, which is itself parameterized over predicates on its type indices and an element of its indexing type. The predicate P can either be obtained by lifting predicates on D's type indices and the type of its term indices, or not. (Those that are not are called *primitive predicates.*) In the latter case, these predicates can themselves either be primitive or obtained by lifting primitive predicates on their own type indices (if any). For the IF Vec, e.g., a predicate $\mathsf{P} : \{\mathsf{n} : \mathsf{Nat}\} \to \mathsf{Vec\,a\,n} \to \mathsf{Set}$ can be given as primitive or obtained by lifting primitive predicates $\mathsf{Q_a}$ on a and $\mathsf{Q_N}$ on Nat. When an IITs indexed type is not uniform (analogous to when an IF's underlying data type is a proper GADT), P *must* be a primitive predicate and the predicates on D's type indices cannot be factored out of P. But for all IITs in the literature they can be, so we construct our liftings and deep induction rules below for uniform IITs. Given the predicates to be lifted, the clause of D's lifting for an element of its indexing or indexed type constructed using data constructor c is obtained by:

(i) ensuring that *the data element being constructed and* any non-recursive data used to construct that data element satisfy the predicates on their types that either parameterize the clauses of D's lifting or are obtained by lifting them. *In this step, we lift only to types other than* D*'s indexed or indexing types.*

(ii) ensuring that all of the data element's (*mutually*) recursive subdata also satisfy the liftings for their types of the predicates from (i). (If D's indexing type is not — or, rather, only trivially — defined mutually with its indexed type, then, as an optimization, the checks on the term indices of the data element's recursive subdata can be omitted by the simplifications elaborated below.)

(iii) ensuring that the term index of every data element constructed using a data constructor of D's indexed type satisfies its predicate from (i) provided the term indices of that element's recursive subdata do.

In (i) and (ii), the italicized text is new relative to the same steps in [13]. The italicized text in (ii) is not needed in [13] because there is no mutually recursive subdata. The italicized text in (i) is not needed in [13] because lifting a primitive predicate on the indexed type of an IF always requires that the primitive predicate holds for all elements of the indexed type (i.e., is trivial), and is thus of no computational interest. For example, the lifting to Vec for the primitive predicate $\mathsf{P} : \{\mathsf{n} : \mathsf{Nat}\} \to \mathsf{Vec\,a\,n} \to \mathsf{Set}$ on vectors is:

$$\begin{aligned}
&\mathsf{Vec}^\wedge : (\{\mathsf{n} : \mathsf{Nat}\} \to \mathsf{Vec\,a\,n} \to \mathsf{Set}) \to \mathsf{Vec\,a\,n} \to \mathsf{Set}\\
&\mathsf{Vec}^\wedge\,\{\mathsf{n} = \mathsf{zero}\}\,\mathsf{P}\,\mathsf{vnil}\; = \;\mathsf{P}\,\mathsf{vnil}\\
&\mathsf{Vec}^\wedge\,\{\mathsf{n} = \mathsf{suc\,m}\}\,\mathsf{P}\,(\mathsf{vcons\,x\,xs})\; = \mathsf{P}\,(\mathsf{vcons\,x\,xs}) \times \mathsf{Vec}^\wedge\,\mathsf{P}\,\mathsf{xs}
\end{aligned} \tag{11}$$

In the clause of $\mathsf{Vec}^\wedge$ for vnil the right-hand side comes from (i). Indeed, (ii) does not contribute any types because vnil has no recursive subdata, and (iii) does not contribute any types because there are no predicates on the indexing type Nat of Vec alone. In the right-hand side of the clause for vcons, the first type comes from (i) and the second comes from (ii). As above, (iii) does not contribute any types. The resulting lifting for Vec is indeed trivial since the first types in each of its right-hand sides ensure that P must actually hold for every vector. For this reason attention is restricted in [13] to predicates on Vec and other IFs that are obtained by lifting predicates on their type indices and the types of their term indices. We will, however, see in Section 5 below that lifting a primitive predicate on the indexed type of an IIT can result in a predicate that is both non-trivial and useful.

Like the lifting for an IIT D, the deep induction rule for D has both a clause for its indexing type and a clause for its indexed type, and these are defined by mutual induction. Each such clause must be parameterized over the same predicate(s) that parameterize(s) the clauses of D's lifting. And, as is usual for deep induction [12,13,14], these predicates must be carefully propagated to all of the data of their domain types. Thus, given the lifting for a (uniform) IIT D, we construct D's deep induction rule as follows:

1. The first argument to each clause of D's deep induction rule is a predicate on D's indexed type (or if the application requires that D's lifting is parameterized by predicates on D's type indices and on the type of its term index, then those predicates are given as arguments instead).

2. The next arguments to each clause are predicates on D's *indexing and* indexed types to be shown to hold for all elements of those types. The predicate on each must be parameterized over the recursive data needed to construct its argument type. *Each such datum must satisfy the Step 2 predicate for its type.* In particular, i) the predicate on D's indexed type must be parameterized over the term index of the element being constructed, and ii) this term index must satisfy the predicate for its type introduced in this step.
3. The next arguments to each clause are induction hypotheses, one for each data constructor of D's *indexing and* indexed types. If c is a data constructor for D's indexing or indexed type, then 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 terms checking that *the term constructed using* c*,* its term index (if any), and each ingredient from Step 3a *that is not the term index of a recursive subterm from Step 3a of the term constructed using* c satisfies the predicate for its type from Step 1.
   (c) Take as final arguments terms checking that for each (*mutually*) recursive ingredient from Step 3a *that is* not *the term index of another such ingredient*, both it *and its term index (if any)* satisfy (appropriate instances of) the predicates for their types from Step 2. *In the check that any element of* D*'s indexing or indexed type satisfies its predicate from Step 2, the induction hypothesis' arguments checking that that element's recursive subdata satisfy their predicates from Step 2 must be used.*
   (d) Have as its conclusion that the term constructed using c satisfies the predicate for its type from Step 2. *If* c *constructs an element of* D*'s indexed type then, in the check that this element satisfies its predicate from Step 2, it is the induction hypothesis' argument checking that this element's term index satisfies the predicate for its type from part ii) of Step 2 that must be used. If no such argument exists, then it is the induction hypothesis for this element's index term, together with the (other) arguments to the induction hypothesis for this term, that must be used.*
4. The conclusion of the clause of D's deep induction rule for its indexing or indexed type is that, given an arbitary element of this type and the ingredients needed to construct its type, if the element satisfies the clause of D's lifting for the Step 1 predicates parameterizing its clause of the deep induction rule, then it satisfies the predicate for its type from Step 2. *To assert that an element of* D*'s indexed type satisfies its predicate from Step 2, it is the clause of the deep induction rule for the indexing type, together with the proof projected out from the element's lifting asserting that the element's term index satisfies* its *lifting, that must be used.*

The italicized text above is new relative to [13]. Critically, the projection functions referenced in Step 4 always exist: the inductive-inductive nature of D's definition ensures that it is always possible to project out from a proof that an element of D's indexed type satisfies its predicate (or the lifting to its type of the predicates) parameterizing the clauses of the D's deep induction rule both that

i) each of the ingredients needed to construct that element satisfies its predicate (or the lifting to its type of the predicates) parameterizing the clauses of D's lifting, and that ii) the same holds for the indexing term of that element.

We show how this construction gives deep induction rules for the IIT of contexts and types in Section 5 and for the IIT of sorted lists in Section 6.

5 Deep Induction for the IIT of Contexts and Types

To illustrate the derivation of liftings and deep induction rules for IITs, we begin by deriving both for the IIT of contexts and types from (3). The above construction gives the following lifting:

$$
\begin{aligned}
&\mathsf{mutual}\\
&\quad \mathsf{Ctxt}^{\wedge} : ((\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}) \to \mathsf{Ctxt} \to \mathsf{Set}\\
&\quad \mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,\epsilon = \top\\
&\quad \mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,(\Gamma \cdot \mathsf{A}) = \mathsf{Ty}^{\wedge}\,\mathsf{Q}\,\Gamma\,\mathsf{A} \times \mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,\Gamma\\
&\quad \mathsf{Ty}^{\wedge} : ((\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}) \to (\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}\\
&\quad \mathsf{Ty}^{\wedge}\,\mathsf{Q}\,\Gamma\,\iota = \mathsf{Q}\,\Gamma\,\iota \;\times\; \mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,\Gamma\\
&\quad \mathsf{Ty}^{\wedge}\,\mathsf{Q}\,\Gamma\,(\Pi\,\mathsf{A}\,\mathsf{B}) = \mathsf{Q}\,\Gamma\,(\Pi\,\mathsf{A}\,\mathsf{B}) \times \mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,\Gamma \;\times\; \mathsf{Ty}^{\wedge}\,\mathsf{Q}\,\Gamma\,\mathsf{A}\;\times\\
&\qquad\qquad\qquad \mathsf{Ty}^{\wedge}\,\mathsf{Q}\,(\Gamma \cdot \mathsf{A})\,\mathsf{B} \;\times ((\mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,\Gamma \times \mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,(\Gamma \cdot \mathsf{A})) \to \mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,\Gamma)
\end{aligned}
\tag{12}
$$

The right-hand side of the clause of $\mathsf{Ctxt}^{\wedge}$ for ϵ comes from the facts that there are no parameterizing predicates on Ctxt alone, that ϵ requires no new or recursive subdata to construct it, and that the type Ctxt is not term-indexed. In the right-hand side of the clause of $\mathsf{Ctxt}^{\wedge}$ for $\_\cdot\_$, both types come from (ii). As in the previous clause, no types come from (iii) since the type Ctxt is not term-indexed. In the right-hand side of the clause of $\mathsf{Ty}^{\wedge}$ for ι, the first type comes from (i) and the second comes from both (ii) and (iii), but only appears once. In the right-hand side of the clause of $\mathsf{Ty}^{\wedge}$ for Π, the first type comes from (i), the next three types come from (ii), and the arrow type comes from (iii).

The observation at the end of the penultimate paragraph of the previous section ensures that the following projection functions exist for the IIT of contexts and types. The function $\mathsf{CTprojPrim}$ asserts that if we can lift a predicate Q to type A in context Γ, then we know that Q holds of Γ and A. The function $\mathsf{CTprojIndex}$ asserts that if we can lift a predicate Q to type A in context Γ, then we can lift Q to Γ. For this IIT there are no further proofs to extract. We have:

$$
\begin{aligned}
&\mathsf{CTprojPrim} : \{\mathsf{Q} : (\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}\} \to \{\Gamma : \mathsf{Ctxt}\} \to (\mathsf{A} : \mathsf{Ty}\,\Gamma) \to\\
&\qquad\qquad \mathsf{Ty}^{\wedge}\,\mathsf{Q}\,\Gamma\,\mathsf{A} \to \mathsf{Q}\,\Gamma\,\mathsf{A}\\
&\mathsf{CTprojPrim}\,\iota\,(\mathsf{Q}\iota, {}^{\wedge}\mathsf{Q}\Gamma) = \mathsf{Q}\iota\\
&\mathsf{CTprojPrim}\,(\Pi\,\mathsf{A}\,\mathsf{B})\,(\mathsf{Q}\Pi\mathsf{AB}, {}^{\wedge}\mathsf{QB}) = \mathsf{Q}\Pi\mathsf{AB}\\
&\mathsf{CTprojIndex} : \{\mathsf{Q} : (\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}\} \to \{\Gamma : \mathsf{Ctxt}\} \to (\mathsf{A} : \mathsf{Ty}\,\Gamma) \to\\
&\qquad\qquad \mathsf{Ty}^{\wedge}\,\mathsf{Q}\,\Gamma\,\mathsf{A} \to \mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,\Gamma\\
&\mathsf{CTprojIndex}\,\iota\,(\mathsf{Q}\iota, {}^{\wedge}\mathsf{Q}\Gamma) = {}^{\wedge}\mathsf{Q}\Gamma\\
&\mathsf{CTprojIndex}\,(\Pi\,\mathsf{A}\,\mathsf{B})\,(\mathsf{Q}\Pi\mathsf{AB}, {}^{\wedge}\mathsf{QB}) = \mathsf{CTprojIndex}\,\mathsf{A}\,(\mathsf{CTprojIndex}\,\mathsf{B}\,{}^{\wedge}\mathsf{QB})
\end{aligned}
\tag{13}
$$

We can use these functions to eliminate redundancies in the lifting in (12). Specifically, in the second clause of $\mathsf{Ctxt}^{\wedge}$ the last term can be omitted since it is derivable from $\mathsf{Ty}^{\wedge}\,\mathsf{Q}\,\Gamma\,\mathsf{A}$ using $\mathsf{CTprojIndex}$. Similarly, in the clause of $\mathsf{Ty}^{\wedge}$ for Π the arrow type can be omitted because its conclusion is already present as the second type, and the second and third types can be omitted because they are derivable from the fourth using $\mathsf{CTprojIndex}$. These simplifications give the following variant of the above lifting, which we use in the deep induction rule in Figure 1 below:

$$\begin{array}{l}
\mathsf{mutual}\\
\quad \mathsf{Ctxt}^{\wedge} : ((\mathsf{\Gamma} : \mathsf{Ctxt}) \to \mathsf{Ty}\,\mathsf{\Gamma} \to \mathsf{Set}) \to \mathsf{Ctxt} \to \mathsf{Set}\\
\quad \mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,\epsilon = \top\\
\quad \mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,(\mathsf{\Gamma} \cdot \mathsf{A}) = \mathsf{Ty}^{\wedge}\,\mathsf{Q}\,\mathsf{\Gamma}\,\mathsf{A}\\[4pt]
\quad \mathsf{Ty}^{\wedge} : ((\mathsf{\Gamma} : \mathsf{Ctxt}) \to \mathsf{Ty}\,\mathsf{\Gamma} \to \mathsf{Set}) \to (\mathsf{\Gamma} : \mathsf{Ctxt}) \to \mathsf{Ty}\,\mathsf{\Gamma} \to \mathsf{Set}\\
\quad \mathsf{Ty}^{\wedge}\,\mathsf{Q}\,\mathsf{\Gamma}\,\iota = \mathsf{Q}\,\mathsf{\Gamma}\,\iota\ \times\ \mathsf{Ctxt}^{\wedge}\,\mathsf{Q}\,\mathsf{\Gamma}\\
\quad \mathsf{Ty}^{\wedge}\,\mathsf{Q}\,\mathsf{\Gamma}\,(\mathsf{\Pi}\,\mathsf{A}\,\mathsf{B}) = \mathsf{Q}\,\mathsf{\Gamma}\,(\mathsf{\Pi}\,\mathsf{A}\,\mathsf{B}) \times \mathsf{Ty}^{\wedge}\,\mathsf{Q}\,(\mathsf{\Gamma} \cdot \mathsf{A})\,\mathsf{B}
\end{array} \tag{14}$$

While $\mathsf{CTprojPrim}$ is not used in the simplifications here, *is* used to construct the inhabitants witnessing the soundness of the deep induction rule in Figure 1. Also, other simplifications of (12) are possible, some of which *do* use $\mathsf{CTprojPrim}$.

The deep induction rule for the IIT of contexts and types in Figure 1 is now obtained as described starting on page 9. The predicate Q parameterizing both clauses comes from Step 1. The predicates Pc and Pt come from Step 2. In the induction hypothesis for $\mathsf{h\Pi}$, e.g., the first three types come from Step 3a, the next three come from Step 3b, the next four come from Step 3c, and the conclusion comes from Step 3d. The induction hypotheses for the other data constructors are obtained similarly. Proof witnesses inhabiting the types in Figure 1 — and, thus, showing that the deep induction rule for the IIT of contexts and types is sound — are given in the code file that accompanies this paper.

A small application illustrates the usefulness of deep induction for the IIT of contexts and types. It uses the deep induction rule from Figure 1 to prove that a type that is hereditarily left-simple is first-order. The predicates $\mathsf{leftSimple}$ and $\mathsf{firstOrder}$ are defined as follows:

$$\begin{array}{l}
\mathsf{data\ nonFunction}\ (\mathsf{\Gamma} : \mathsf{Ctxt}) : \mathsf{Ty}\,\mathsf{\Gamma} \to \mathsf{Set\ where}\\
\quad \mathsf{nf}\iota : \mathsf{nonFunction}\,\mathsf{\Gamma}\,\iota\\[4pt]
\mathsf{leftSimple} : (\mathsf{\Gamma} : \mathsf{Ctxt}) \to \mathsf{Ty}\,\mathsf{\Gamma} \to \mathsf{Set}\\
\mathsf{leftSimple}\,\mathsf{\Gamma}\,\iota = \mathsf{nonFunction}\,\mathsf{\Gamma}\,\iota\\
\mathsf{leftSimple}\,\mathsf{\Gamma}\,(\mathsf{\Pi}\,\mathsf{A}\,\mathsf{B}) = \mathsf{nonFunction}\,\mathsf{\Gamma}\,\mathsf{A}\\[4pt]
\mathsf{firstOrder} : (\mathsf{\Gamma} : \mathsf{Ctxt}) \to \mathsf{Ty}\,\mathsf{\Gamma} \to \mathsf{Set}\\
\mathsf{firstOrder}\,\mathsf{\Gamma}\,\iota = \mathsf{nonFunction}\,\mathsf{\Gamma}\,\iota\\
\mathsf{firstOrder}\,\mathsf{\Gamma}\,(\mathsf{\Pi}\,\mathsf{A}\,\mathsf{B}) = \mathsf{nonFunction}\,\mathsf{\Gamma}\,\mathsf{A} \times \mathsf{firstOrder}\,(\mathsf{\Gamma} \cdot \mathsf{A})\,\mathsf{B}
\end{array} \tag{15}$$

Here, the definition of the type $\mathsf{nonFunction}$ captures the fact that only ι is not a function type, the predicate $\mathsf{leftSimple}$ asserts that a type is *left-simple* if it doesn't have any $\mathsf{\Pi}$s nested on the left of a $\mathsf{\Pi}$, and the predicate $\mathsf{firstOrder}$ asserts that a type is *first-order* if it doesn't have any $\mathsf{\Pi}$s nested on the left of a $\mathsf{\Pi}$ and is recursively first-order on the right. We can now precisely state and prove our desired result — namely, that for any type formed in any context, if that type is hereditarily left-simple then it is first-order — as in Figure 2.

Note that this application of deep induction uses a non-trivial predicate $\mathsf{leftSimple}$ on the indexed set Ty. But it also uses a trivial predicate on the index-

```
mutual
 CtxtDInd : (Q : (Γ : Ctxt) → Ty Γ → Set) →
   (Pc : Ctxt → Set) →
   (Pt : (Γ : Ctxt) → Ty Γ → Pc Γ → Set) →
   (hϵ : Pc ϵ) →
   (h· : (Γ : Ctxt) → (A : Ty Γ) → Q Γ A → (PcΓ : Pc Γ) → Pt Γ A PcΓ → Pc (Γ · A)) →
   (hι : (Γ : Ctxt) → Q Γ ι → (PcΓ : Pc Γ) → Pt Γ ι PcΓ) →
   (hΠ : (Γ : Ctxt) → (A : Ty Γ) → (B : Ty (Γ · A)) → Q Γ A → Q (Γ · A) B → Q Γ (Π A B) →
     (PcΓ : Pc Γ) → Pt Γ A PcΓ → (PcΓA : Pc (Γ · A)) → Pt (Γ · A) B PcΓA → Pt Γ (Π A B) PcΓ) →
   (Γ : Ctxt) → Ctxt^ Q Γ → Pc Γ

 TyDInd : (Q : (Γ : Ctxt) → Ty Γ → Set) →
   (Pc : Ctxt → Set) →
   (Pt : (Γ : Ctxt) → Ty Γ → Pc Γ → Set) →
   (hϵ : Pc ϵ) →
   (h· : (Γ : Ctxt) → (A : Ty Γ) → Q Γ A → (PcΓ : Pc Γ) → Pt Γ A PcΓ → Pc (Γ · A)) →
   (hι : (Γ : Ctxt) → Q Γ ι → (PcΓ : Pc Γ) → Pt Γ ι PcΓ) →
   (hΠ : (Γ : Ctxt) → (A : Ty Γ) → (B : Ty (Γ · A)) → Q Γ A → Q (Γ · A) B → Q Γ (Π A B) →
     (PcΓ : Pc Γ) → Pt Γ A PcΓ → (PcΓA : Pc (Γ · A)) → Pt (Γ · A) B PcΓA → Pt Γ (Π A B) PcΓ) →
   (Γ : Ctxt) → (A : Ty Γ) → (^QA : Ty^ Q Γ A) →
     Pt Γ A (CtxtDInd Q Pc Pt hϵ h· hι hΠ Γ (CTprojIndex A ^QA))
```

Fig. 1. Deep induction rule for the IIT of contexts and types

ing type Ctxt so that a type of interest can be formed in *any* context. A second, larger application of deep induction for IITs that uses non-trivial predicates on *both* the IIT's indexing type *and* on its indexed type is given in Section 7.

6 Deep Induction for the IIT of Sorted Lists

The application in Section 5 uses a primitive predicate on the indexed type of the IIT of contexts and types. We now consider the IIT of sorted lists from (4), whose indexing type and indexed type both have the same index type a. Both clauses of both the lifting and the deep induction rule for the IIT of sorted lists are thus parameterized over a predicate on a. We have:

```
mutual
  SList^ : {{orda : Ordered a}} → (a → Set) → SList a → Set
  SList^ Q snil = ⊤
  SList^ Q (scons {x} {xs} x≥xs) =  Q x  × SList^ Q xs × ≥L^ Q x≥xs

  ≥L^ : {{orda : Ordered a}} → (a → Set) → {x : a} → {xs : SList a} → x ≥L xs → Set
  ≥L^ Q {x} triv = Q x  ×  SList^ Q snil
  ≥L^ Q (extn {x} {y} {ys} y≥ys _ x≥ys) =  Q x  ×  Q y × SList^ Q ys ×  ≥L^ Q y≥ys ×
                                           ≥L^ Q x≥ys  × (SList^Q ys → SList^Q (scons y≥ys))
                                                                                        (16)
```

With respect to the construction on page 9, the right-hand side of the clause of SList^ for snil comes from the facts that there is no predicate that can be

```
hls⇒fo : (Γ : Ctxt) → (A : Ty Γ) → Ty^ leftSimple Γ A → firstOrder Γ A
hls⇒fo Γ A hyp = TyDInd Q Pc Pt hϵ h· hι hΠ Γ A hyp where

  Q : (Γ : Ctxt) → Ty Γ → Set
  Q = leftSimple

  Pc : Ctxt → Set
  Pc Γ = ⊤

  Pt : (Γ : Ctxt) → Ty Γ → Pc Γ → Set
  Pt Γ A PcΓ = firstOrder Γ A

  hϵ : Pc ϵ
  hϵ = tt

  h· : (Γ : Ctxt) → (A : Ty Γ) → Q Γ A → (PcΓ : Pc Γ) → Pt Γ A PcΓ → Pc (Γ · A)
  h· Γ A QA PcΓ PtA = PcΓ

  hι : (Γ : Ctxt) → Q Γ ι → (PcΓ : Pc Γ) → Pt Γ ιPcΓ
  hι Γ Qι PcΓ = Qι

  hΠ : {Γ : Ctxt} → (A : Ty Γ) → (B : Ty (Γ · A)) → Q Γ A → Q (Γ · A) B →
    Q Γ (Π A B) → (PcΓ : Pc Γ) → Pt Γ A PcΓ → (PcΓA : Pc (Γ · A)) →
    Pt (Γ · A) B PcΓA → Pt Γ (Π A B) PcΓ
  hΠ A B QA QB QΠAB PcΓ PtA PcΓA PtB = (QΠAB, PtB)
```

Fig. 2. Application of deep induction rule for the IIT of contexts and types

applied to snil alone, no new data or recursive subdata is needed to construct snil, and the type SList has no term indices. In the right-hand side of the clause of $\mathsf{SList}^{\wedge}$ for scons, the first type come from (i), the second and third types come from (ii), and no types come from (iii) since the type SList is not term-indexed. In the right-hand side of the clause of $\geq_{\mathsf{L}}^{\wedge}$ for triv, the first type comes from (i) and the second type comes from both (ii) and (iii), but only appears once. In the right-hand side of the clause of $\geq_{\mathsf{L}}^{\wedge}$ for extn, the first two types come from (i), the next three types come from (ii), and the arrow type comes from (iii).

We can use the following projection functions to simplify $\mathsf{SList}^{\wedge}$ and $\geq_{\mathsf{L}}^{\wedge}$:

```
SprojPrim : {{orda : Ordered a}} → {Q : a → Set} → {x : a} →
                {xs : SList a} → (x≥xs : x ≥L xs) →≥L^ Q x≥xs → Q x
SprojPrim triv Qx = Qx
SprojPrim (extn _ _ x≥ys) (_, Qx≥ys) = SprojPrim x≥ys Qx≥ys
                                                                        (17)
SprojIndex : {{orda : Ordered a}} → {Q : a → Set} → {x : a} →
                {xs : SList a} → (x≥xs : x ≥L xs) →≥L^ Q x≥xs → SList^Q xs
SprojIndex triv _ = tt
SprojIndex (extn _ _ _) (Qy≥ys, _) = Qy≥ys
```

The function SprojPrim asserts that if an element x≥xs satisfies its lifting, then the primitive datum x used to construct it must satisfy Q. To define it, we capitalize on the facts that $\geq_{\mathsf{L}}$ is inductive, and that the parameter x does not change in the recursive calls in its definition, to project out Qx only once the data constructor triv is reached. The function SprojIndex ensures that if an element x≥xs satisfies its lifting, then its indexing term xs also satisfies its lifting. The functions SprojPrim and SprojIndex are the analogues for the IIT of sorted lists of the functions CTprojPrim and CTprojIndex for the IIT of contexts and types. Using SprojPrim and SprojIndex, the first and second types in the clause of $\mathsf{SList}^\wedge$ for scons are derivable from the third and thus can be omitted. The second type in the clause of $\geq_{\mathsf{L}}^{\wedge}$ for triv simplifies to $\top$ and so can be omitted. The first three types in the clause of $\geq_{\mathsf{L}}^{\wedge}$ for extn can similarly be omitted, and the arrow type there can be omitted because its conclusion $\mathsf{SList}^\wedge\,\mathsf{Q}\,(\mathsf{scons}\,\mathsf{y{\geq}ys})$ simplifies to exactly the fourth type. From these observations we obtain the following simplified version of the lifting for the IIT of sorted lists, which we use in its deep induction rule below:

$$
\begin{array}{l}
\mathsf{mutual}\\
\quad \mathsf{SList}^\wedge : \{\{\mathsf{orda} : \mathsf{Ordered\,a}\}\} \to (\mathsf{a} \to \mathsf{Set}) \to \mathsf{SList\,a} \to \mathsf{Set}\\
\quad \mathsf{SList}^\wedge\ \mathsf{Q\,snil} = \top\\
\quad \mathsf{SList}^\wedge\ \mathsf{Q}\,(\mathsf{scons\,x{\geq}xs}) = \geq_{\mathsf{L}}^{\wedge}\ \mathsf{Q\,x{\geq}xs}\\
\quad \geq_{\mathsf{L}}^{\wedge} : \{\{\mathsf{orda} : \mathsf{Ordered\,a}\}\} \to (\mathsf{a} \to \mathsf{Set}) \to \{\mathsf{x} : \mathsf{a}\} \to \{\mathsf{xs} : \mathsf{SList\,a}\} \to \mathsf{x} \geq_{\mathsf{L}} \mathsf{xs} \to \mathsf{Set}\\
\quad \geq_{\mathsf{L}}^{\wedge}\ \mathsf{Q}\,\{\mathsf{x}\}\,\mathsf{triv} = \mathsf{Q\,x}\\
\quad \geq_{\mathsf{L}}^{\wedge}\ \mathsf{Q}\,(\mathsf{extn\,y{\geq}ys}\ \_\ \mathsf{x{\geq}ys}) = \geq_{\mathsf{L}}^{\wedge}\ \mathsf{Q\,y{\geq}ys} \times \geq_{\mathsf{L}}^{\wedge}\ \mathsf{Q\,x{\geq}ys}
\end{array}
\tag{18}
$$

As for the IIT of contexts and types, other simplifications are possible here.

Once we have the lifting for the IIT of sorted lists in hand we can derive its deep induction rule. This is given in Figure 3. It has two clauses, one for the indexing type SList and one for the indexed type $\geq_{\mathsf{L}}$. Both of these are parameterized over a predicate Q from Step 1, as well as predicates Pl and P≥ to be proved for SList and $\geq_{\mathsf{L}}$, respectively, from Step 2. Both are also parameterized over the induction hypotheses hnil, hcons, htriv, and hextn for all of their constructors. These induction hypotheses are obtained as described in Section 4. For example, in the induction hypothesis hextn for extn, the first six arguments come from Step 3a, the next two arguments come from Step 3b, the next four come from Step 3c, and the conclusion of the induction hypothesis comes from Step 3d. Note that Step 3c gives two (potentially different) proof terms Plys1 and Plys2 of type Pl ys; this is explained in the next paragraph. Moreover, in the type of P≥yys the term Plys1 witnessing that ys satisfies Pl is used, as specified in Step 3c, and similarly for Plys2 and P≥xys. Also as specified in Step 3d, the conclusion that the term constructed by extn satisfies P≥ uses the induction hypothesis hcons and the arguments to hextn in its assertion that the term index scons y≥ys of the term constructed using extn satisfies Pl. The induction hypotheses for the other data constructors are derived similarly. Finally, note that the conclusion of $\geq_{\mathsf{L}}$DInd uses SListDInd and SprojIndex, exactly as specified in Step 4 of our construction.

A particular subtlety of deep induction is this: proving that a predicate holds for a term of the indexed type that has more than one recursive subterm indexed by the same indexing term can involve more than one proof witness that the predicate to be proved for the indexing type holds for that index. For example, the proof witnesses Plys1 and Plys2 for Pl ys in the type of hextn come from the arguments y≥ys and x≥ys to hextn, respectively, both of which are indexed by

```
mutual
 SListDInd : {{orda : Ordered a}} → (Q : a → Set) →
   (Pl : SList a → Set) →
   (P≥: {x : a} → {xs : SList a} → x ≥L xs → Pl xs → Set) →
   (hnil : Pl snil) →
   (hcons : {x : a} → {xs : SList a} → (x≥xs : x ≥L xs) → Q x →
                 (Plxs : Pl xs) → P≥ x≥xs Plxs → Pl (scons x≥xs)) →
   (htriv : {x : a} → Q x → P≥ {x} triv hnil) →
   (hextn : {x y : a} → {ys : SList a} → (y≥ys : y ≥L ys) → (x≥y : x ≥ y) → (x≥ys : x ≥L ys) →
                 Q x → (Qy : Q y) → (Plys1 : Pl ys) → (P≥yys : P≥ y≥ys Plys1) → (Plys2 : Pl ys) →
                 (P≥xys : P≥ x≥ys Plys2) → P≥ (extn y≥ys x≥y x≥ys) (hcons y≥ys Qy Plys1 P≥yys)) →
   (xs : SList a) → SList^ Q xs → Pl xs

 ≥L DInd : {{orda : Ordered a}} → (Q : a → Set) →
   (Pl : SList a → Set) →
   (P≥: {x : a} → {xs : SList a} → x ≥L xs → Pl xs → Set) →
   (hnil : Pl snil) →
   (hcons : {x : a} → {xs : SList a} → (x≥xs : x ≥L xs) → Q x →
                 (Plxs : Pl xs) → P≥ x≥xs Plxs → Pl (scons x≥xs)) →
   (htriv : {x : a} → Q x → P≥ {x} triv hnil) →
   (hextn : {x y : a} → {ys : SList a} → (y≥ys : y ≥L ys) → (x≥y : x ≥ y) → (x≥ys : x ≥L ys) →
                 Q x → (Qy : Q y) → (Plys1 : Pl ys) → (P≥yys : P≥ y≥ys Plys1) → (Plys2 : Pl ys) →
                 (P≥xys : P≥ x≥ys Plys2) → P≥ (extn y≥ys x≥y x≥ys) (hcons y≥ys Qy Plys1 P≥yys)) →
   {x : a} → {xs : SList a} → (x≥xs : x ≥L xs) → (Qx≥xs : ≥L^ Q x xs x≥xs) →
    P≥ x≥xs (SListDInd Q Pl P≥ hnil hcons htriv hextn xs (SprojIndex x≥xs Qx≥xs))
```

Fig. 3. Deep induction rule for the IIT of sorted lists

the same element ys of SList a. But different proofs of Pl ys may have been used in constructing the proofs P≥yys : P≥ y≥ys Plys1 and its counterpart for x≥ys, so Plys1 and Plys2 need not be identical. This issue also arises for deep induction for IFs. But it never arises for structural induction since there is always exactly one way to prove a predicate holds for a given term using structural induction.

As for the deep induction rule for the IIT of contexts and types, proof witnesses inhabiting the types in Figure 3 — and, thus, showing that the deep induction rule for sorted lists is sound — are given in the code file that accompanies this paper. The deep induction rule for the IIT of dense order completions is also given there. Deep induction rules for IITs whose indexing and indexed types are non-uniform are far more complicated than than those for the uniform IITs considered here, but these can be obtained by combining the methodology presented here for IITs with the techniques developed in [12,13,14].

When an indexed type's indexing set is not inductive we do not necessarily have a technique at the ready for proving a predicate holds for all of its elements. But when it is, (structural or deep) induction becomes available. If the IIT's indexed type also has the same inductive structure as its indexing type, then its lifting effectively embeds the deep induction rule for its indexing type. This can make it possible to further simplify the IIT's lifting and deep induction rule.

7 Application of Deep Induction for the IIT of Sorted Lists

As another illustration of the usefulness of deep induction for IITs, in this section we develop a second, larger application of deep induction that makes use of non-trivial predicates on both an IIT's indexing and indexed types. Specifically, we show how the deep induction rule for the IIT of sorted lists from Figure 3 can be used to prove that, if $>$ is the built-in ordering on Nat, then mapping a $>$- and evenness-preserving function over any list of even natural numbers in strictly decreasing order results in another such list. We describe this application in this section, and include our complete Agda solution in the code file that accompanies this paper.

To prove the aforementioned theorem for sorted lists we require the following ordering and predicates. As usual, the type Nat of natural numbers is that from (2), the relevant ordering is the built-in ordering

```
instance
  NatOrdered : OrderedNat
  _≥_{{NatOrdered}} = Nat._≥_
```

on Nat, and the predicate even on Nat is given by:

```
even : Nat → Set
even zero = ⊤
even (suc zero) = ⊥
even (suc (suc n)) = even n
```

We also need to check that the sorted lists we are mapping over contain only even natural numbers that appear in strictly decreasing order. The following mutually defined functions do this. Although the ordering $>$ on Nat below is transitive, we define $>_L$ as we do in order to accommodate non-transitive orderings as well.

```
mutual
  >SList : SList Nat → Set
  >SList snil = ⊤
  >SList (scons x≥xs) = >L x≥xs

  >L: {x : Nat} → {xs : SList Nat} → x ≥L xs → Set
  >L triv = ⊤
  >L {x} (extn{y} y≥ys _ x≥ys) = x > y × >L x≥ys × >L y≥ys
```

in the following functions defined using the lifting for sorted lists from (18):

```
^Even&DecrSList : SList Nat → Set
^Even&DecrSList xs = SList^ even xs × >SList xs

^Even&Decr≥L: {x : Nat} → {xs : SList Nat} → x ≥L xs → Set
^Even&Decr≥L x≥xs = ≥L^ even x≥xs × >L x≥xs
```

Of course, if we happen to know that an element of SList contains only even natural numbers and that these numbers appear in strictly decreasing order, then we can infer individually both that its elements are all even and that its elements appear in strictly decreasing order:

```
^Even&DecrSList⇒^Even : (xs : SList Nat) → ^Even&DecrSList xs → SList^ even xs
^Even&DecrSList⇒^Even xs (evenxs, _) = evenxs

^Even&DecrSList⇒Decr : (xs : SList Nat) → ^Even&DecrSList xs →>SList xs
^Even&DecrSList⇒Decr xs (_, decrxs) = decrxs
```

Similarly, if we happen to know that ^Even&Decr$\geq_{\mathsf{L}}$ holds for x≥xs then we can both lift even to x≥xs and infer that $>_{\mathsf{L}}$ x≥xs holds:

```
^Even&Decr≥L⇒^Even : {x : Nat} → {xs : SList Nat} → (x≥xs : x ≥L xs) →
                   ^Even&Decr≥L x≥xs →≥L^ even x≥xs
^Even&Decr≥L⇒^Even x≥xs (evenx≥xs, _) = evenx≥xs

^Even&Decr≥L⇒Decr : {x : Nat} → {xs : SList Nat} → (x≥xs : x ≥L xs) →
                   ^Even&Decr≥L x≥xs → >L x≥xs
^Even&Decr≥L⇒Decr x≥xs (_, x>xs) = x>xs
```

These facts will be useful in proving the main result of this section in Figure 4.

The functions to be mapped over sorted lists preserve both the ordering > on, and the evenness of, natural numbers. The following predicates identify them:

```
evenPres : (Nat → Nat) → Set
evenPres f = (x : Nat) → even x → even (f x)

>Pres : (Nat → Nat) → Set
>Pres f = (x y : Nat) → x > y → f x > f y
```

To apply such functions to the elements of the IIT of sorted lists we need map functions mapSList and map$\geq_{\mathsf{L}}$ for SList and $\geq_{\mathsf{L}}$, respectively. But in order for the applications of mapSList and map$\geq_{\mathsf{L}}$ to a function f : a → b to have the expected types SList a → SList b and $\geq_{\mathsf{L}}$ {a} → $\geq_{\mathsf{L}}$ {b}, respectively, f must preserve ≥. We therefore map only functions f that are monotone over sorted lists:

```
monotone : {{orda : Ordered a}} → {{ordb : Ordered b}} → (f : a → b) → Set
monotone {a} f = (a1 a2 : a) → a1 ≥ a2 → f a1 ≥ f a2
```

The map function for the IIT of sorted lists is then given by:

```
mutual
  mapSList : {{orda : Ordered a}} → {{ordb : Ordered b}} →
                (f : a → b) → monotone f → SList a → SList b
  mapSList _ _ snil = snil
  mapSList f monof (scons x≥xs) = scons (map≥L f monof x≥xs)

  map≥L: {{orda : Ordered a}} → {{ordb : Ordered b}} →
                (f : a → b) → (monof : monotone f) → {x : a} → {xs : SList a} →
                      x ≥L xs → f x ≥L mapSList f monof xs
  map≥L _ _ triv = triv
  map≥L f monof {x} (extn {y} y≥ys x≥y x≥ys) =
    extn (map≥L f monof y≥ys) (monof x y x≥y) (map≥L f monof x≥ys)
```

To state our theorem we need the following auxiliary results establishing that, for Nat, boolean equality == is interchangeable with Agda's built-in definitional equality ≡:

```
===⇒≡: (m == n) ≡ true → m ≡ n
===⇒≡ {zero} {zero} p = refl
===⇒≡ {suc a} {suc b} p = cong suc (===⇒≡ p)

¬===⇒≢: (m == n) ≡ false → m ≢ n
¬===⇒≢ {zero} {suc b} p = 0≢1+n
¬===⇒≢ {suc a} {zero} p = 1+n≢0
¬===⇒≢ {suc a} {suc b} p suca≡sucb = ¬===⇒≢ p (suc−injective suca≡sucb)
```

Here, cong asserts that its function argument preserves propositional equality, suc−injective asserts that the data constructor suc for Nat is injective, 0≢1+n asserts that the data constructor zero for Nat is not propositionally equal to a successor natural number, and 1+n≢0 asserts that no successor natural number is propositionally equal to zero. These functions are all defined in Agda's standard library. We also need the fact that a function that preserves > on Nat also preserves ≥ on Nat. This can be coded in Agda as:

```
>Pres⇒Mono : (f : Nat → Nat) →>Pres f → monotone f
>Pres⇒Mono f >Presf a1 a2 a1≥a2 with a1 == a2 in p
... | false = <⇒≤ (>Presf a1 a2 (≤∧≢⇒< a1≥a2 (≢−sym (¬===⇒≢ p))))
... | true rewrite ===⇒≡ {a1} {a2} p = ≤−reflexive refl
```

Here, <⇒≤ asserts that if m < n for m, n : Nat then m ≤ n, ≤∧≢⇒< asserts that ≤ and ≢ imply <, ≢−sym asserts that not being definitionally equal is a symmetric relation, and ≤−reflexive asserts that ≤ is a reflexive relation. The keyword rewrite allows for the substitution of one expression in a given type for another based on a given equality proof. More specifically, if g is a term and q is a proof that l ≡ r, then rewrite q = g convinces Agda to accept a goal term g whose type is obtained from the original goal type by replacing all occurrences of l by r. Thus, in the last clause of >Pres⇒Mono, in which the term ===⇒≡ {a1} {a2} p of type a1 ≡ a2 plays the role of q, the goal type f a1 ≥ f a2 of >Pres⇒Mono f >Presf a1 a2 a1≥a2 is treated as though it were rewritten by replacing each occurrence of a1 by a2. The end result is that Agda is convinced to accept the goal term ≤−reflexive refl, whose type is f a2 ≤ f a2, rather than requiring a goal term whose type is f a1 ≤ f a2.

Our desired theorem can now be stated and proved as in Figure 4.

8 Conclusion and Directions for Future Work

This paper gives the first-ever deep induction rules for proper IITs and demonstrates their soundness. The mutual definition of an IIT's indexing and indexed types requires considerably more finesse in defining liftings and deep induction rules for them than doing so for IFs, in part because they can be parameterized over a larger class of predicates than IFs can without giving trivial results.

```
mapPres>&Even : (f : Nat → Nat) → (>Presf : >Pres f) → evenPres f → (xs : SListNat) →
  SList^ even xs → >SList xs → ^Even&DecrSList (mapSList f (>Pres⇒Mono f >Presf) xs)
mapPres>&Even f >Presf evenPresf xs evenxs =
  SListDInd even (λ ys → >SList ys → ^Even&DecrSList (mapSListf ys))
                 (λ y≥ys _ → >L y≥ys → ^Even&Decr≥L (map≥Lf y≥ys))
                 hnil hcons htriv hextn xs evenxs where

    monof : monotone f
    monof = >Pres⇒Mono f >Presf

    mapSListf : SList Nat → SList Nat
    mapSListf = mapSList f monof

    map≥Lf : {x : Nat} → {xs : SList Nat} → x ≥L xs → f x ≥L mapSListf xs
    map≥Lf = map≥L f monof

    hnil :>SList snil → ^Even&DecrSList snil
    hnil tt = tt, tt

    hcons : {x : Nat} → {xs : SList Nat} → (x≥xs : x ≥L xs) → even x →
      (>SList xs → ^Even&DecrSList (mapSListf xs)) →
      (>L x≥xs → ^Even&Decr≥L (map≥Lf x≥xs)) →
      >SList (scons x≥xs) → ^Even&DecrSList (scons (map≥Lf x≥xs))
    hcons x≥xs evenx PIxs P≥xxs x>xs = P≥xxs x>xs

    htriv : {x : Nat} → even x → >L (triv {x = x}) → ^Even&Decr≥L {f x} triv
    htriv {x} evenx tt = evenPresf x evenx , tt

    hextn : {x y : Nat} → {ys : SList Nat} → (y≥ys : y ≥L ys) → (x≥y : x Nat.≥ y) →
            (x≥ys : x ≥L ys) → even x → even y →
            (>SList ys → ^Even&DecrSList (mapSListf ys)) →
            (>L y≥ys → ^Even&Decr≥L (map≥Lf y≥ys)) →
            (>SList ys → ^Even&DecrSList (mapSListf ys)) →
            (≥L x≥ys → ^Even&Decr≥L (map≥L f x≥ys)) →
            >L (extn y≥ys x≥y x≥ys) →
            ^Even&Decr≥L (extn (map≥Lf y≥ys) (monof x y x≥y) (map≥Lf x≥ys))
    hextn {x} {y} y≥ys x≥y x≥ys evenx eveny PIys1 P≥yys PIys2 P≥xys (x>y , x>ys, y>ys) =
            (^Even&Decr≥L⇒^Even (map≥Lf y≥ys) (P≥yys y>ys) ,
             ^Even&Decr≥L⇒^Even (map≥Lf x≥ys) (P≥xys x>ys)) ,
            >Presf x y x>y ,
            ^Even&Decr≥L⇒Decr (map≥Lf x≥ys) (P≥xys x>ys) ,
            ^Even&Decr≥L⇒Decr (map≥Lf y≥ys) (P≥yys y>ys)
```

Fig. 4. Application of deep induction for the IIT of sorted lists

But this paper actually delivers far more than deep induction rules for some specific IITs. It actually gives a *general methodology* for deriving sound deep induction rules for IITs that can be instantiated to particular ones of interest. This methodology can serve as a basis for conservatively extending proof assistants' generation of structural induction rules for IITs to the generation of deep induction rules for them, which we have demonstrated are significantly more useful.

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. Chapman, J.: Type theory should eat itself. In: Proceedings of the International Workshop on Logical Frameworks and Meta-languages. pp. 21–36 (2009). https://doi.org/10.1016/j.entcs.2008.12.114
4. Cheney, J., Hinze, R.: First-class phantom types. Tech. Rep. CUCIS TR2003-1901, Cornell University (2003)
5. Christiansen, D.: Practical Reflection and Metaprogramming for Dependent Types. Ph.D. thesis, IT University of Copenhagen (2015)
6. Conway, J.: On Numbers and Games. AK Peters (2001)
7. Coq Development Team: The Coq proof assistant, version 8.19.2 (2024)
8. 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
9. Danielsson, N.A.: A formalisation of a dependently typed language as an inductive-recursive family. In: TYPES 2006. pp. 93–109 (2007). https://doi.org/10.1007/978-3-540-74464-1_7
10. Dybjer, P.: Inductive families. Formal Aspects of Computing **6**(4), 440–465 (1994). https://doi.org/10.1007/BF01211308
11. Idris: A language for type-driven development, https://www.idris-lang.org/
12. 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
13. Johann, P., Morehouse, E.: Deep induction for inductive families. In: Logic, Language, Information, and Computation. pp. 21–37 (2025). https://doi.org/10.1007/978-3-031-99536-1_2
14. 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
15. McBride, C.: Epigram: Practical programming with dependent types. In: Advanced Functional Programming. pp. 130–170 (2005). https://doi.org/10.1007/11546382_3
16. McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming **14**(1), 69–111 (2004). https://doi.org/10.1017/S0956796803004829
17. 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
18. Nordvall Forsberg, F.: Inductive-inductive definitions. PhD thesis, Swansea University (2013)
19. Nordvall Forsberg, F., Setzer, A.: Inductive-inductive definitions. In: Computer Science Logic. pp. 454–468 (2010). https://doi.org/10.1007/978-3-642-15205-4_35
20. Norell, U.: Dependently typed programming in Agda (2008), Lecture Notes, Advanced Functional Programming Summer School
21. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1999). https://doi.org/10.1017/CBO9780511530104
22. Peyton Jones, S.L. (ed.): Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003)
23. 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

24. Ullrich, M.: Generating induction principles for nested induction types in MetaCoq. Bachelor thesis, Saarland University (2020)
25. 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