Foundational Nonuniform (Co)datatypes for Higher-Order Logic

Jasmin Christian Blanchette*, Fabian Meier†, Andrei Popescu‡, and Dmitriy Traytel†
*Vrije Universiteit Amsterdam, The Netherlands, and Inria & LORIA, Nancy, France
†Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
‡School of Science and Technology, Middlesex University London, UK

Abstract—Nonuniform (or “nested” or “heterogeneous”) datatypes are recursively defined types in which the type arguments vary recursively. They arise in the implementation of finger trees and other efficient functional data structures. We show how to reduce a large class of nonuniform datatypes and codatatypes to uniform types in higher-order logic. We programmed this reduction in the Isabelle/HOL proof assistant, thereby enriching its specification language. Moreover, we derive (co)induction and (co)recursion principles based on a weak variant of parametricity.

I. INTRODUCTION

Inductive (or algebraic) datatypes—often simply called datatypes—are a central feature of typed functional programming languages and of most proof assistants. A simple example is the type of finite lists over a type parameter α, specified as follows (in a notation inspired by Standard ML):

\[ \text{α list} = \text{Nil} \mid \text{Cons α (α list)} \]

A datatype is uniform if the recursive occurrences of the datatype have the same arguments as the definition itself, as is the case for list; otherwise, the datatype is nonuniform. Nonuniform types are also called “nested” or “heterogeneous” in the literature. Powerlists are nonuniform:

\[ \text{α plist} = \text{Nil} \mid \text{Cons α ((α × α) plist)} \]

The type α plist is freely generated by the constructors Nil : α plist and Cons : α → (α × α) plist → α plist. When Cons is applied several times, the product type constructors (×) accumulate to create pairs, pairs of pairs, and so on. Thus, any powerlist of length 3 will have the form

\[ \text{Cons α (Cons (b_1, b_2) (Cons ((c_{11}, c_{12}), (c_{21}, c_{22})) Nil))} \]

Nonuniform datatypes arise in the implementation of efficient functional data structures, such as finger trees [25], and they underlie Okasaki’s bootstrapping and implicit recursive slowdown optimization techniques [36]. Yet many programming languages and proof assistants lack proper support for such types. For example, even though Standard ML allows nonuniform definitions, a typing restriction disables interesting recursive definitions. As for proof assistants, Agda, Coq, Lean, and Matita allow nonuniform definitions, but they are built into the logic (dependent type theory), with all the risks and limitations that this entails [13, Section 1].

For systems based on higher-order logic such as HOL4, HOL Light, and Isabelle/HOL, no dedicated support exists for nonuniform types, probably because they are widely believed to lie beyond the logic’s ML-style polymorphism. Building on the well-understood metatheory of uniform datatypes (Section II), we disprove this folk belief by showing how to define a large class of nonuniform datatypes by reduction to their uniform counterparts within higher-order logic (Section III).

Our constructions allow variations along several axes. They cater for mutual definitions:

\[ \text{α ptree} = \text{Node α (α pforest)} \]
\[ \text{α pforest} = \text{Nil} \mid \text{Cons (α ptree) ((α × α) pforest)} \]

They allow multiple recursive occurrences, with different type arguments:

\[ \text{α plist} = \text{Nil} \mid \text{Cons} (\alpha, \beta) \text{ α plist} \mid \text{Cons} (\alpha, \beta) \text{ α plist' } \]

They all allow multiple type arguments, which may all vary:

\[ (\alpha, \beta) \text{ tplist} = \text{Nil β} \mid \text{Cons} (\alpha, \beta, \text{ unit + β}) \text{ tplist} \]

Moreover, they allow the presence of datatypes, codatatypes, and other well-behaved type constructors both around the type arguments and around the recursive type occurrences:

\[ \text{α stree} = \text{Node α (((α fset) stree) fset}} \]

Here, fset is the type constructor associated with finite sets.

Furthermore, the constructions can be extended to coinductive (or coalgebraic) datatypes—codatatypes. Codatatypes are generally non-well-founded, allowing infinite values. They are often used to model the datatypes of languages with a nonstrict evaluation strategy, such as Haskell, and they can be very convenient for some proof tasks. The codatatype definition

\[ \text{α pstream} = \text{Cons α ((α × α) pstream)} \]

introduces “powerstreams,” with infinite values of the form Cons α (Cons (b_1, b_2) (Cons ((c_{11}, c_{12}), (c_{21}, c_{22})) Nil))

Unlike dependent type theory, higher-order logic requires all types to be nonempty (inhabited). To introduce a new type, we must both provide a construction in terms of existing types and prove its nonemptiness. For example, a datatype specification analogous to the pstream codatatype above should be rejected. In previous work [14], we showed how to decide the nonemptiness problem for uniform types—including mutually recursive specifications and arbitrary mixtures of datatypes.
and codatatypes—by viewing the definitions as a grammar, with the defined types as nonterminals. Here, we extend this result to nonuniform types (Section IV). This is achieved by encoding the nonuniformities in a generalized grammar, which can decide nonemptiness of the sets that arise in the construction of the nonuniform types.

Once a datatype has been introduced, users want to define functions that recurse on it and carry out proofs by induction involving these functions—and similarly for codatatypes. A uniform datatype definition generates an induction theorem and a recursor. Nonuniform datatypes pose a challenge, because neither the induction theorem nor the recursor can be expressed in higher-order logic, due to its limited polymorphism. For example, induction for list should look like this:

\[ \forall Q. Q \text{Nil} \land (\forall x. Q x \Rightarrow Q (\text{Cons} x x)) \Rightarrow \forall y. Q y \]

However, this formula is not typable in higher-order logic, because the second and third occurrences of the variable \( Q \) need different types: \( (\alpha \times \alpha) \text{plist} \to \text{bool} \) versus \( \alpha \text{plist} \to \text{bool} \). Our solution is to replace the theorem by a procedure parameterized by a polymorphic property \( \varphi_\alpha; \alpha \text{plist} \to \text{bool} \) (Section V). For \( \text{plist} \), the procedure transforms a proof goal of the form \( \varphi_\alpha \text{ys} \) into two subgoals \( \varphi_\alpha \text{Nil} \) and \( \forall x. x. \varphi_\alpha x x \Rightarrow \varphi_\alpha (\text{Cons} x x) \). A weak form of parametricity is needed to recursively transfer properties about \( \varphi_\alpha \) to properties about \( \varphi_{\alpha \times \alpha} \).

All the constructions and derivations are formalized in the Isabelle/HOL proof assistant and form the basis of high-level commands that let users define nonuniform types and (co)recursive functions on them and reason (co)inductively about them (Section VII). The commands are foundational: Unlike all previous implementations of nonuniform types in proof assistants, they require no new axioms or extensions of the logic. An example involving \( \lambda \)-terms in De Bruijn notation demonstrates the practical potential of our approach.

Our main contributions are the following: First, we designed a reduction of nonuniform datatypes to uniform datatypes within the relatively weak higher-order logic, including recursion and induction. Second, we adapted the constructions to support codatatypes as well, exploiting dualities. Third, we coded the reduction in a proof assistant based on higher-order logic, yielding a first implementation of nonuniform datatypes without dependent types. The formal proofs, the source code, and the examples are publicly available [11].

II. PRELIMINARIES

A. Higher-Order Logic

We consider classical higher-order logic (HOL) with Hilbert choice, the axiom of infinity, and rank-1 polymorphism. HOL is based on Church’s simple type theory [15]. It is the logic of Gordon’s original HOL system [19] and of its many successors and emulators, notably HOL4, HOL Light, and Isabelle/HOL.

Primitive types are built from type variables \( \alpha, \beta, \ldots \), a type \text{bool} of Booleans, and an infinite type \text{ind} using the function type constructor; thus, \( (\text{bool} \to \alpha) \to \text{ind} \) is a type. Primitive constants are equality \( = \colon \alpha \to \alpha \to \text{bool} \), the Hilbert choice operator, and 0 and Suc for \text{ind}. Terms are built from constants and variables by means of typed \( \lambda \)-abstraction and application.

A polymorphic type is a type \( T \) that contains type variables. If \( T \) is polymorphic with variables \( \vec{\alpha} = (\alpha_1, \ldots, \alpha_n) \), we write \( \vec{T} \) instead of \( T \). Formulas are closed terms of type \text{bool}.

The logical connectives and quantifiers on formulas are defined using the primitive constants—e.g., True as \( (\lambda x. \text{bool} x) = (\lambda x. \text{bool} x) \). Polymorphic formulas are thought of as universally quantified over their type variables. For example, \( \forall x : \alpha. x = x \) really means \( \forall a. a = a \). Nested type quantifications such as \( (\forall a. \ldots) \Rightarrow (\forall a. \ldots) \) are not expressible. We will express concepts in standard mathematical language whenever the expressiveness of HOL is not a concern.

The only primitive mechanism for defining new types in HOL is type definition: For any existing type \( \vec{T} \) and predicate \( P : \vec{T} \to \text{bool} \) such that \( \{ x : \vec{T} \mid P x \} \) is nonempty, we can introduce a type \( \vec{T} S \) isomorphic to \( \{ x : \vec{T} \mid P x \} \).

Upon meeting the definition \( \vec{T} S = \{ x : \vec{T} \mid P x \} \), the system requires the user to prove \( \exists x : \vec{T}. P x \) and then introduces the type \( \vec{T} S \), the projection \( \text{Rep}_\vec{T} : \vec{T} S \to \vec{T} \), and the embedding \( \text{Abs}_\vec{T} : \vec{T} S \to \vec{T} S \) such that \( \forall x. P (\text{Rep}_\vec{T} x) \), \( \forall x. \text{Abs}_\vec{T} (\text{Rep}_\vec{T} x) = x \), and \( \forall x. P x \Rightarrow \text{Rep}_\vec{T} (\text{Abs}_\vec{T} x) = x \). The nonemptiness check is necessary because all types in HOL must be nonempty. This is a well-known design decision connected to the presence of Hilbert choice in HOL [19], [37].

Thus, unlike dependent type theory, HOL does not have (co)datatypes as primitives. However, datatypes [5], [20], [21], [32], [41] and, more recently, codatatypes [41] are supported via derived specification mechanisms. Users can write fixpoint definitions an ML-style syntax, and the system defines the type using nonrecursive type definitions (ultimately appealing to \text{ind} and \to); defines the constructors and related operators; and proves characteristic properties, such as injectivity of constructors, induction theorems, and recur
We regard the elements of $\sigma F$ as containers filled with “content” from $\alpha_i$—the set$_F$ functions return the $\alpha_i$-content (known to be bounded by bd$_F$) and map$_F$ $\bar{f}$ replaces content via $\bar{f}$. For example, list is a unary BNF, where map$_{\text{list}}$ is the standard map function, set$_{\text{list}}$ collects all the list’s elements, and bd$_{\text{list}}$ is $\mathbb{R}_0$.

BNFs are closed under (least and greatest) fixpoint definitions [41] and the nonemptiness problem for BNFs is decidable [14]. These crucial properties enable a modular approach to mixing and nesting uniform (co)datatypes and deciding if a high-level specification yields valid, i.e., nonempty, HOL types. In addition, BNFs display predicator and relator predicates returning the first element of a list is a boolean $\text{Nil}$ and $\text{Cons}$ is a constructor. Separate constructors and destructors are easy to introduce as syntactic sugar [10, Section 4]. For powerlists, the $\text{Nil}$ constructor and relator structure [39]. The predicator $\text{pred}_F : (\alpha_1 \rightarrow \text{bool}) \rightarrow \cdots \rightarrow (\alpha_n \rightarrow \text{bool}) \rightarrow \sigma F \rightarrow \text{bool}$ and the relator $\text{rel}_F : (\alpha_1 \rightarrow \beta_1 \rightarrow \text{bool}) \rightarrow \cdots \rightarrow (\alpha_n \rightarrow \beta_n \rightarrow \text{bool}) \rightarrow \sigma F \rightarrow \beta F \rightarrow \text{bool}$, are defined from set$_F$ and map$_F$ as follows:

- $\text{pred}_F \bar{P} x \iff \forall i \in [n]. \forall a \in \text{set}_F x. P_i a$;
- $\text{rel}_F \bar{R} x \iff \exists z. \forall i \in [n]. \exists z \in (\alpha_i, a) | R_i (a b)) \land \text{map}_F \bar{f} z = x \land \text{map}_F \bar{z} = y$, where $\bar{f}$ and $\bar{z}$ are standard projection functions on the product type $\times$.

For list, set$_{\text{list}}$ $P x$ states that $P$ holds for all elements of the list $x$, and rel$_{\text{list}}$ $R x y$ states that $x$ and $y$ have the same length and are element-wise related by $R$.

Relators and predicates are useful to track parametricity [38], [42]. A polymorphic constant $c : \sigma F$ is parametric if, for all relations $R_i : \alpha_i \rightarrow \beta_i \rightarrow \text{bool}$ for each $i \in [n]$, we have $\text{rel}_F R c c$—i.e., every two instances of $c$ are related by the lifting of the relations associated with the component types. Parametricity applies not only to BNFs but also to any combination of BNFs using the function space. For polymorphic functions $f : \sigma F \rightarrow \sigma G$ between two BNFs, $f$ is parametric if and only if $f$ is a natural transformation [12, Appendix A].

### III. (Co)Datatype Definitions

Before describing the reduction of nonuniform (co)datatypes to uniform (co)datatypes in full generality, we start with a simple example that conveys the main idea. The reduction proceeds by defining a larger uniform datatype and carving out a subset that is isomorphic to the desired nonuniform type. To prove that the constructed type is the intended one, we establish the isomorphism between the defined nonuniform type and the right-hand side of its specification.

#### A. Introductory Example

Okasaki [36, Section 10.1.1] sketches how to mimic nonuniform datatypes using uniform datatypes. He approximates powerlists by the following definitions:

- $\text{datatype} \ a \text{ sh} = \text{Leaf } a \mid \text{Node } (a \times a) \text{ sh}$
- $\text{datatype} \ a \text{ raw} = \text{Nil}_0 \mid \text{Cons}_0 (a \text{ sh}) \ (a \text{ raw})$

The type $a \text{ raw}$ corresponds to lists of binary trees $a \text{ sh}$. It is larger than powerlists in two ways: (1) $a \text{ sh}$ allows non-full binary trees, which cannot arise in a powerlist; (2) $a \text{ raw}$ imposes no restriction on the depth of the binary trees, whereas a powerlist has elements successively of depth $0, 1, 2, \ldots$.

Okasaki considers these mismatches as one of two disadvantages of the above encoding. The other disadvantage is that the encoding requires users to insert $\text{Leaf}$ and $\text{Node}$ coercions to convert an element such as $(a, b), (c, d) : (a \times a) \times (a \times a)$ to Node $(\text{Leaf}_a, \text{Leaf}_b), (\text{Node}_c, \text{Leaf}_d) : a \text{ sh}$.

We overcome the first disadvantage by using a type definition. From the raw type, we select exactly those inhabitants that correspond to powerlists. To achieve this, we define two predicates, $\text{ok} : \alpha \times a \rightarrow \text{bool}$ and $\text{ok} : \alpha \rightarrow a \times \text{raw}$, as the least predicates satisfying the following rules:

- $\text{ok} n \text{ Nil}_0 \ (\text{Leaf } x)$
- $\text{ok} n \text{ Nil}_0 \ (\text{Nil } n) w = \Rightarrow \text{ok} n (\text{Cons}_0 (\text{Nil}_0 n) w)$

The predicate $\text{ok} n r$ checks whether $t$ is a full binary tree of depth $n$, and $\text{ok} n x$ checks that the first element of a list is a full tree of depth $n$, the second is of depth $n + 1$, and so on. The desired type starts at depth $0$: $a \text{ plist} = \{xs : a \times a | \text{ok } 0 \times x\}$.

The second disadvantage is addressed by hiding the internal construction of $a \text{ plist}$. We define the powerlist constructors $\text{Nil} : a \text{ plist} \mid \text{Cons} : a → (a × a) a \text{ plist} → a \text{ plist}$ in terms of $\text{Nil}_0$ and $\text{Cons}_0$. These definitions will require some additional machinery on the raw type.

#### B. Datatypes

We assume that the desired nonuniform datatype has a single constructor. Separate constructors are easy to introduce as syntactic sugar [10, Section 4]. For powerlists, the single constructor definition is $a \text{ plist} = \text{Ctor}_\text{plist} (\text{unit } a + a × a) \text{ plist}$. It corresponds to finding a least solution (up to isomorphism) to the type fixpoint equation $a \text{ plist} \simeq (a, (a F)plist) G$ with $a F = a × a$ and $a (a, a) G = \text{unit } + a × a × a$. We generalize this setting in multiple dimensions. First, we support a simultaneous definition of an arbitrary number $i$ of mutual nonuniform datatypes. For example, $\text{ptree}$ and $\text{pforest}$ from Section I are given by the system of fixpoint equations

- $a \text{ ptree } \simeq (a, (a F_1) \text{ ptree}, (a F_2) \text{ pforest}) G_1$
- $a \text{ pforest } \simeq (a, (a F_3) \text{ ptree}, (a F_4) \text{ pforest}) G_2$

where $\{a, (a, a) G_1 = a × a \times a, (a, (a, a) G_2 = \text{unit } + a × a \times a, a F_1 = a F_2 = a F_3 = a, a F_4 = a \times a \times a \}$. We assume that all $G$’s depend on the same type variables, even though the dependence may be spurious, as in the case of $G_1$ and $\beta$.

Second, a type may occur several times on the right-hand side of a definition. We support an arbitrary number $j$ of such occurrences. This feature is used in the $\text{plist}'$ type: $a \text{ plist}' \simeq (a, (a F_1) \text{ plist}', (a F_2) \text{ plist})' G$, where $\{a, (a, a) G = \text{unit } + a × a \times a, a F_1 = a F_2 = a F_3 = a, a F_4 = a \times a \times a \}$. We assume that all $G$’s depend on the same type variables, even though the dependence may be spurious, as in the case of $G_1$ and $\beta$.

Finally, the construction supports an arbitrary number $k$ of type parameters. The parameter changes may differ for different type parameters, such as in the $\text{plist}$ example: $(a, (a, (a, a) F_1, (a, a) F_2) \text{ plist}) G$, where $\{a, (a, a) G = \text{unit } + a × a, (a, a) F_1 = a × a, \text{ and } (a, a) F_2 \text{ unit } + a \}$.

As before for the $G$’s, all $F$’s may depend on all type parameters of the specified nonuniform type.

In the sequel, the indices $i, j$, and $k$ range over $[i], [j]$, and $[k]$, respectively. Moreover, we abbreviate indexed sequences using a horizontal bar; for example, $\sigma$ stands for $\alpha_1, \ldots, \alpha_k$. 
and $\overline{\alpha} F_j$ stands for $\overline{\alpha} F_{j1}, \ldots, \overline{\alpha} F_{jk}$. It should be clear from the context which index is omitted.

A definition of $i$ mutual nonuniform datatypes $T_i$ is a system of $i$ type fixpoint equations

$$\overline{\alpha} T_i \cong (\overline{\alpha}, (\overline{\alpha} F_1) \overline{\alpha} T_{\sigma(1)}, \ldots, (\overline{\alpha} F_j) \overline{\alpha} T_{\sigma(j)}) G_i$$

where $G_i$'s are $(k + j)$-ary BNFs, the $F_j$'s are $k$-ary BNFs, and $\sigma : [j] \to [i]$ is a monotone surjective function which expresses which of the $i$ mutual types belongs to which recursive occurrence. The construction generalizes Okasaki's idea and yields $k$-ary BNFs $T_i$ that are least solutions (up to isomorphism) to equation (1). A uniform datatype definition [10] is a special case with $j = i, \sigma(i) = i, \overline{\alpha} F_j = \overline{\alpha} T_i$.

We now define the shape types $\overline{\alpha} sh_k$ of overapproximate the recursive changes to the type arguments. There are $k$ shape types, corresponding to the $k$ type arguments, and they are mutually recursive datatypes:

$$\overline{\alpha} sh_k = \text{Leaf}_k \overline{\alpha} \cdot \text{Node}_{\overline{\alpha}}(\overline{\alpha} s'h F_k) \ldots \text{Node}_{\overline{\alpha}}(\overline{\alpha} s'h F_k)$$

For $\overline{\alpha} sh_k$, the $sh$ is $\overline{\alpha} k$. In general, each recursive occurrence may change the type arguments in a different way; this is reflected in the different Node constructors.

Next, we define $i$ uniform mutually recursive datatypes $raw_i$ that recur through the $G_i$'s in the same way as the $T_i$'s do, except that they keep the type arguments unchanged. The immediate $\overline{\alpha}$ arguments to $G_i$ are replaced by $\overline{\alpha} sh$:

$$\overline{\alpha} raw_i = \text{Raw}_i(\overline{\alpha} s'h \overline{\alpha} raw_{\sigma(1)}, \ldots, \overline{\alpha} raw_{\sigma(j)}) G_i$$

For every $i$, we specify subsets of the types $\overline{\alpha} raw_i$ that are isomorphic to the nonuniform types $T_i$, by defining predicates $ok_i$ that characterize the allowed shapes and their changes in the recursion. As in the powerlist example, the definition of $ok_i$ relies on auxiliary predicates $\overline{\alpha} \text{Pred}_i : \overline{\alpha} \text{list} \rightarrow \overline{\alpha} s'h \rightarrow \text{bool}$ on the shape types. The type of $\overline{\alpha} \text{Pred}_i$ shows an important difference to the $\overline{\alpha} \text{list}$ example: The first argument is not just a natural number denoting the depth of a full tree but has more structure. We call it the shadow of the shape and let $\Delta$ stand for $\overline{\alpha} \text{list}$. The additional structure is necessary because different Node constructors may occur in a single shape element. These occurrences in the full shape trees are layered: All Node constructors right above the Leaf constructors belong to a fixed occurrence $j$. The next layer of Nodes may belong to a different fixed occurrence $j'$. The shadow summarizes the occurrence indices. Consider $\text{Cons}_1(\alpha) = \alpha \cdot \alpha \cdot \alpha \cdot \alpha \cdot \alpha$ and $\text{Cons}_2(\alpha \times \alpha) = \alpha \cdot \alpha \cdot \alpha \cdot \alpha \cdot \alpha$ and $\text{Nil}(\alpha \times \alpha) = \alpha \cdot \alpha \cdot \alpha \cdot \alpha \cdot \alpha$.

This order of constructors forces $x$'s type to be $\alpha F = \alpha F_1 F_2$, with $\alpha F_1 = \alpha$ and $\alpha F_2 = \alpha \times \alpha$. Consequently, $x$ is embedded into a $sh$ as $\text{Node}_2(\text{map}_F \text{Node}_1(\text{map}_F_2(\text{map}_F \text{Leaf}(x))))$, whose shape is $[2, 2, 1]$. Formally, the predicates $\overline{\alpha} \text{Pred}_i$ are defined together as the least predicates satisfying the rules

$$\overline{\alpha} \text{Pred}_i(\overline{\alpha} s'h) \Rightarrow (\overline{\alpha} \text{Pred}_i(\overline{\alpha} s'h) \overline{\alpha} \text{Pred}_i(\overline{\alpha} s'h)) r$$

where $\overline{\alpha} \text{Pred}_i(\overline{\alpha} s'h)$ are defined similarly:

$$\overline{\alpha} \text{Pred}_i(\overline{\alpha} s'h) \Rightarrow (\overline{\alpha} \text{Pred}_i(\overline{\alpha} s'h) \overline{\alpha} \text{Pred}_i(\overline{\alpha} s'h)) r$$

Finally, the nonuniform type $T_i$ can be defined as the subset of $\overline{\alpha} raw_i$ that satisfies the $ok_i$ predicate for the empty shadow: $\overline{\alpha} T_i = \{ r : \overline{\alpha} raw_i \mid ok_i r \}$. Such a type definition introduces a new type and the embedding–projection pairs $\overline{\alpha} \text{Pred}_i : \overline{\alpha} T_i \rightarrow \overline{\alpha} raw_i$ and $\text{Abs}_i : \overline{\alpha} raw_i \rightarrow \overline{\alpha} T_i$. The emerging nonemptiness problem is discussed in Section IV.

We can prove by induction that $ok_i$ is invariant under the $\overline{\alpha} \text{map}_r$ function.

**Lemma 1:** $ok_i r \Rightarrow \overline{\alpha} \text{Pred}_i r$.

This property is sufficient to prove that $T_i$ is a BNF. By virtue of being a BNF, $T_i$ can appear around type arguments and recursive type occurrences in future uniform or nonuniform (co)datatype definitions.

### C. Constructors

If the type $T_i$ is the nonuniform type that we intended to construct, it should satisfy equation (1). We prove this isomorphism by defining a constructor $\text{Ctor}_i : \overline{\alpha} (\overline{\alpha} F_i) T_{\sigma(1)}, \ldots, (\overline{\alpha} F_i) T_{\sigma(j)}) G_i \rightarrow \overline{\alpha} T_i$ and a destructor $\text{dtor}_i : \overline{\alpha} T_i \rightarrow (\overline{\alpha} (\overline{\alpha} F_i) T_{\sigma(1)}, \ldots, (\overline{\alpha} F_i) T_{\sigma(j)}) G_i$ and by showing that they are inverses of each other.

Figure 1 gives diagrammatic definitions of $\text{Ctor}_i$ (by composing the functions on the outer arrows) and $\text{dtor}_i$ (by composing the functions on the inner arrows). All shape and raw types occurring in the diagram are annotated with their shadows. $\text{Abs}_i$ can be applied only to raw elements with shadow $\overline{\alpha}$. The unLeaf$_k$ and unRaw$_i$ functions are inverses of the corresponding constructors; they satisfy unLeaf$_k(\text{Leaf}_k a) = a$

and unRaw$_i(\text{Raw}_i(\overline{\alpha} s'h a)) = a$. The unLeaf$_k$ and unRaw$_i$ functions are inverses of the corresponding constructors; they satisfy unLeaf$_k(\text{Leaf}_k a) = a$

and unRaw$_i(\text{Raw}_i(\overline{\alpha} s'h a)) = a$. The unLeaf$_k$ and unRaw$_i$ functions are inverses of the corresponding constructors; they satisfy unLeaf$_k(\text{Leaf}_k a) = a$

and unRaw$_i(\text{Raw}_i(\overline{\alpha} s'h a)) = a$. The unLeaf$_k$ and unRaw$_i$ functions are inverses of the corresponding constructors; they satisfy unLeaf$_k(\text{Leaf}_k a) = a$.
and unRawi (Rawi r) = r. Note that unLeafk is underspecified and (like Absr) may be applied only to Leafk shape elements with shadow . Moreover, the definition must bridge the gap between the types ↪(↕Fj)rawi↑(j) of shadow and ↪(↕Fj)rawi↑(j) of shadow (j) (the rightmost arrows in Figure 1). This must happen recursively (even though the constructor Ctor itself is not recursive), by inlining the additional Fs into a new layer of the shape type (directly above the Leaf constructors) and therefore requires a generalization that transforms an arbitrary shadow u into u ↑ j (i.e., the list u with the element j appended to it). For each fixed j, inlining is implemented by means of mutual recursively primitive functions ↪j k : (↕Fj) shk → ↪ shk on the shape type:

\[
\begin{align*}
\mathbb{I}_k(\text{Leaf}_k f) &= \text{Node}_k (\text{map}_{F_k \downarrow} \text{Leaf} f) \\
\mathbb{I}_k(\text{Node}_k f) &= \text{Node}_k (\text{map}_{F_k \downarrow} f) \\
\uparrow_{j} u (\text{Raw}_i g) &= \text{Raw}_i (\text{map}_{G_i \downarrow} (\downarrow_{j} (1 \uparrow u)) \ldots (\downarrow_{j} (j \uparrow u)) g)
\end{align*}
\]

Inlining is injective. We define the (partial) inverse operations \(\downarrow j : \Delta \rightarrow \sigma \text{raw}_1 \rightarrow (\leftrightarrow F_j) \sigma \text{raw}_i\) and \(\mathbb{I}_k : \Delta \rightarrow \sigma \text{sh}_1 \rightarrow (\leftrightarrow F_j) \sigma \text{sh}_i\), which are useful when defining the destructors dtor. The additional shadow parameter in \(\mathbb{I}_r\) denotes how many more layers to destruct until we arrive at the last layer of Nodes (with only Leaf constructors below).

\[
\begin{align*}
\mathbb{I}_k &\downarrow \mathbb{I}_k (\text{Node}_k f) = \text{Leaf}_k (\text{map}_{F_k \uparrow} \text{unLeaf} f) \\
\mathbb{I}_k &\downarrow (f \uparrow u) (\text{Node}_k f) = \text{Node}_k (\text{map}_{F_k \uparrow} ((\mathbb{I}_k u) f)) \\
\downarrow_{j} u (\text{Raw}_i g) &= \text{Raw}_i (\text{map}_{G_i \uparrow} (\downarrow_{j} (1 \uparrow u)) \ldots (\downarrow_{j} (j \uparrow u)) g)
\end{align*}
\]

We establish the expected behavior of \(\uparrow_{j}\) and \(\downarrow_{j}\) with respect to shadows and prove that they are mutually inverse. The proofs proceed by induction on the raw type using very similar omitted auxiliary lemmas for \(\mathbb{I}_r\) and \(\mathbb{I}_k\).

**Lemma 2:**

1. \(\alpha_{k} \uparrow u \Rightarrow \alpha_{k} (u \uparrow j) (\uparrow_{j} r) = \uparrow_{j} u (\uparrow_{j} r) = r\);  
2. \(\alpha_{k} (u \uparrow j) (\uparrow_{j} r) = \alpha_{k} (u \uparrow j) \uparrow_{j} r = \downarrow_{j} (\uparrow_{j} r) = r\).

Since every pair of arrows in Figure 1 is mutually inverse (when applied to elements of the right shadow), we obtain our desired isomorphism property for Ctor and dtor.

**Theorem 3:** \(\text{dtor}_i (\text{Ctor}_i g) = g\) and \(\text{dtor}_i (\text{dtor}_i i) = i\).

Finally, we prove characteristic theorems for \(T_i\)’s BNF constants. We focus on the property that \(\text{map}_{G_i}\) commutes with the constructor Ctor. The theorems for the relator, the predicate, and the set functions are proved analogously.

**Theorem 4:** \(\text{map}_{G_i} (\mathbb{I}_r (\text{Ctor}_i g)) = \text{Ctor}_i (\text{map}_{G_i} (\mathbb{I}_r (\text{Ctor}_i g)))\) where \(\sigma R_i = \langle (\leftrightarrow F_j) \sigma T_{\alpha_i (1)} \ldots (\leftrightarrow F_j) \sigma T_{\alpha_i (j)} \rangle G_i\) and \(\text{map}_{G_i}\) is the map function associated to this composite BNF.

The proof of Theorem 4 relies on commutation properties of \(\text{map}_{\text{raw}}\) and \(\uparrow_{j}\) and of \(\text{map}_{\text{sh}_j}\) and \(\mathbb{I}_k\) that can be proved by induction. This is a persisive pattern when defining recursive functions on nonuniform datatypes.

**Lemma 5:** \(\text{map}_{\text{sh}_j} (\mathbb{I}_j) = \mathbb{I}_j (\text{map}_{\text{sh}_j} (\text{map}_{F_j} \mathbb{I}_j s)) \) and \(\text{map}_{\text{raw}_i} (\mathbb{I}_r (\mathbb{I}_r r)) = \mathbb{I}_r (\text{map}_{\text{raw}_i} (\text{map}_{F_i} \mathbb{I}_r r))\).

**D. Codatatypes**

The construction can be gracefully extended to support codatatypes, which are types whose elements may be infinitely deep. Codatatypes are the types \(T_i\) that are greatest solutions to equation (1).

This change in semantics needs to be reflected only at the raw level. Accordingly, the \(\text{raw}_i\) types are defined as mutually corecursive uniform types. The shape types remain unchanged, since even in an infinitely deep object all type arguments are finite (but unbounded) type expressions.

The subsequent changes are also minor. The predicates \(\alpha_{k}\) are defined as a mutual greatest (or coinductive) fixpoint of the same introduction rule as before for datatypes. The functions \(\uparrow_{j}\) and \(\downarrow_{j}\) are defined by primitive corecursion using the same equations as before.

All theorems from Subsections III-B and III-C hold as stated also for codatatypes. The proofs, however, are different: For example, whereas Lemma 2(1–2) was proved by induction on \(r\), for codatatypes the corresponding argument proceeds by coinduction on the now coinductive definitions of \(\alpha_{k}\). Similarly, the equational statements (e.g., Lemma 2(3–4) or the raw part of Lemma 5) are proved by coinduction on \(=\).

**IV. THE NONEMPTYNESS PROBLEM**

Types in HOL must be nonempty. As we are developing more sophisticated high-level datatype specification mechanisms, the problem of establishing nonemptiness of the introduced types becomes more difficult. For nonuniform mutual (co)datatypes \(T_i\), the question is whether \(T_i\) are indeed valid HOL types, i.e., are nonempty. We want an answer that is automatic (i.e., is given without asking the user to perform any proof) and complete (i.e., does not reject valid types).

In previous work, we offered a solution for mutual uniform (co)datatypes [14]. It is based on storing, for each BNF \(\sigma K\) with \(\sigma = \langle \alpha_1, \ldots, \alpha_n \rangle\), complete information on its conditional nonemptiness, i.e., on which combinations of nonemptiness assumptions for the argument types \(\alpha_i\) would be sufficient to guarantee nonemptiness of \(\sigma K\). For example, if \(n = 3\) and \(\sigma K\) is \(\alpha_1 \text{stream} + \alpha_2 \times \alpha_3\), then for \(\sigma K\) to be nonempty it suffices that either \(\alpha_1\), or both \(\alpha_2\) and \(\alpha_3\) be nonempty. We say that both \(\{\alpha_1\}\) and \(\{\alpha_2, \alpha_3\}\), or, simply, \(\{1\}\) and \(\{2,3\}\) are witnesses for the nonemptiness of \(\alpha_1 \text{stream} + \alpha_2 \times \alpha_3\).

The above discussion assumes that \(K\) operates on possibly empty collections of elements (which, technically, as a type constructor, it does not, since the HOL type variables are assumed to range over nonempty types). To model this, we employ the setK operators to capture the action of \(K\) on sets, as the homonymous constant \(K : \alpha_1 \text{set} \rightarrow \cdots \rightarrow \alpha_n \text{set} \rightarrow (\sigma K)\). Defined by \(K A_1 \ldots A_n = \{ x : \sigma K \mid \forall i \in [n], \text{set}_K x \subseteq A_i \}\). The constant \(K\) operates on sets in the same way as the type constructor \(K\) operates on types. Since sets can be empty, we can use them to express witnesshood.

Given \(I \subseteq [n]\), we call \(I\) a witness for \(K\) if, for all sets \(\overline{A_i}\), \(\forall i \in I, A_i \neq \emptyset\) implies \(K \overline{A} \neq \emptyset\). A set \(J \subseteq [n]\) set of witnesses for \(K\) is called perfect if for all witnesses \(I \subseteq [n]\) there exists a witness \(I \in J\) such that \(I \subseteq J\). Thus, a perfect set of witnesses
for $K$ is one where no witness is missed, in that any witness $J$ is equal to, or improved by, a witness $I \in \mathcal{I}$.

We fix a definition of $i$ mutual nonuniform datatypes $T_i$, as depicted in equation (1) from Section III-B. We assume that the involved BNFs (the $G_i$’s and the $F_{jk}$’s) are endowed with perfect sets of witnesses $\mathcal{J}(G_i)$ and $\mathcal{J}(F_{jk})$. We will show how to effectively construct perfect sets of witnesses for the $T_i$’s. This construction allows us to decide whether the $T_i$’s are nonempty, hence valid HOL types, by simply checking if their perfect sets are nonempty. In addition, it equips the $T_i$’s with the infrastructure needed to establish the nonemptiness of future (co)datatypes that may use them as parameters.

To find the witnesses for the $T_i$’s, we generalize the approach we developed for uniform datatypes. We define a context-free set-grammar (which is like a standard context-free grammar except that its productions act on finite sets instead of words) having the $T_i$’s as nonterminals and the argument types $\alpha_i$ as terminals. The productions of the set-grammar follow the direction of the destructors, $\mathcal{J}(T_i) \rightarrow (\mathcal{J}(\mathcal{F}_1), \mathcal{J}(\mathcal{F}_2), \ldots, \mathcal{J}(\mathcal{F}_i)) G_i$, with each $T_i$ deriving sets containing the nonterminals $T_j$ and the terminals $\alpha_k$ allowed by witnesses of $G_i$.

For the nonuniform case, when recursively applying productions $\tau: T_i \rightarrow (\tau_1, \tau_2, \ldots, \tau_l) G_i$ following the definitions, the $T_i$’s are applied to increasingly larger polynomial expressions involving the $F_{jk}$’s. To keep the grammar finite, we take a more abstract view, retaining from the $F_{jk}$-expressions only their witness-relevant information, obtained by suitably combining their perfect sets $\mathcal{J}(F_{jk})$. We define the set $\text{PolyWit}$, of polynomial witness sets (or polywits), inductively as follows:

- If $k \in [k]$, then $\{\{k\}\}$ ∈ PolyWit.
- If $(j,k) \in [j] \times [k]$ and $p_1, \ldots, p_k \in \text{PolyWit}$, then $\langle p_1, \ldots, p_k \rangle \cdot \mathcal{J}(F_{jk}) \in \text{PolyWit}$.

Polywits are sets of subsets of $[k]$. In the second clause above, we use the composition $(\langle p_1, \ldots, p_k \rangle \cdot \mathcal{J}(F_{jk}))$, which is defined as $\bigcup \mathcal{J}(F_{jk}) \cup \bigcup \mathcal{J}(F_{jk}) \cup \mathcal{J}(F_{jk}) \cap [J \in \bigcup \mathcal{J}(F_{jk}) \cap [J \in \bigcup \mathcal{J}(F_{jk}) \cup \mathcal{J}(F_{jk})]$. This composition captures the computation of witnesses for the composition of $F_{jk}$ with the BNFs corresponding to the polywits $p_1, \ldots, p_k$.

We fix a set of tokens, $\text{Tok} = \{t_i \mid i \in [i]\}$, to symbolically represent the $T_i$’s. We define the set-grammar $\text{Gr} = (\text{Term}, \text{NTerm}, \text{Prod})$ as follows. Its terminals are $[k]$, i.e., one number $k \in [k]$ for each type variable $\alpha_k$. The nonterminals are either polywits or have the form $(p_1, \ldots, p_k) \cdot t_i$, where each $p_k$ is a polywit and $i \in [i]$. There are two types of productions:

1) $p \rightarrow I$, where $p \in \text{PolyWit}$ and $I \in p$;
2) $\tau: t_i \rightarrow \tau_j$ where $i \in [i]$, $j \in \mathcal{J}(G_i)$ and $\tau_j = \{p \mid k \in J \cap [k] \cup \{\{\tau \cdot \mathcal{J}(F_{jk}) \cup \mathcal{J}(F_{jk})\} \cap [J] \cup \{\{\tau \cdot \mathcal{J}(F_{jk}) \cup \mathcal{J}(F_{jk})\} \cap [J] \mid k + j \in [J]\}.

The first type of production selects witnesses from polywits. The second type mirrors the recursion in the definition of the $T_i$’s by following the destructors and selecting the terminals and nonterminals according to the witnesses of $G_i$.

Let $\text{Lang}_i(G)$ be the language (i.e., the set of subsets of $[k]$) generated by $\text{Gr}$ starting from the nonterminal $\{\{a_1\}, \ldots, \{a_k\}\}$ (the token for $T_i$ applied to the trivial polywits for its argument types). Let $\text{Lang}_\omega_i(G)$ be the language cogenerated by $\text{Gr}$—allowing infinite chains of productions, and infinite derivation trees—again, starting from $\{\{a_1\}, \ldots, \{a_k\}\}$.

**Theorem 6:** If we interpret the definition as specifying mutual datatypes, then the definition is valid in HOL (i.e., the specified types are nonempty) if and only if $\text{Lang}_i(G) \neq \emptyset$ and $\text{Lang}_\omega_i(G)$ is a perfect set of witnesses for $T_i$. If we interpret the definition as specifying mutual codatatypes, then the definition is always valid in HOL and $\text{Lang}_\omega_i(G)$ is a perfect set of witnesses for $T_i$.

The theorem statement distinguishes the nonemptiness subproblem from the witness problem. This is because the $T_i$’s cannot be introduced as types without knowing their nonemptiness (i.e., the nonemptiness of their representing predicates $\alpha_k$). For codatatypes, nonemptiness always holds owing to the greatest fixpoint nature of the construction.

Since the raw $\omega_i$’s are BNFs, we have a perfect set of witnesses for them, but these will usually fail to satisfy $\alpha_k$ (and even if they do satisfy the predicate, they need not constitute a perfect set for $T_i$). To prove the theorem, we adapt the notion of witness from types to predicates and show that the languages (co)generated by $\text{Gr}$ offer perfect sets for $\alpha_k$ for any shadow $u$. We generalize from $\alpha_1$ to an arbitrary $u$ because the shadow increases along applications of the raw $\omega_i$ destructors. Our technical report [12, Appendix B] provides the details.

For any finite set-grammar $\text{Gr}$, the languages $\text{Lang}_i(G)$ and $\text{Lang}_\omega_i(G)$ are effectively computable by fixpoint iteration [14]. Moreover, iteration needs at most a number of steps equal to the number of nonterminals [14, Section 4.3]. For uniform datatypes, this number is precisely that of mutual types, $i$; for nonuniform datatypes, it is the larger number $i \times k \times |\text{PolyWit}|$, where $|\text{PolyWit}| = O(2^k)$. Fortunately, the worst-case double exponential in $k$ is unproblematic in practice for two reasons. First, the defined types tend to have few type variables. Second, the nonemptiness witnesses for the BNFs $G_i$ can prune a large part of the search space: If a $G_i$ has a constructor without recursive arguments, then $\{\}$ is a witness for $G_i$ and thus also for the (co)datatype $T_i$ regardless of $k$.

Consider the following contrived nonuniform codatatype of alternating streams over $\alpha_1$ and $\alpha_2$:

$$(\alpha_1, \alpha_2) \text{ alt} \equiv C \alpha_1 ((\alpha_2, \alpha_1) \text{ alt}) \mid D \alpha_2 ((\alpha_2, \alpha_1) \text{ alt})$$

We have $i = 1$, $j = 1$, $k = 2$, $\sigma$ is the unique function from $[1]$ to $[1]$, $(\alpha_1, \alpha_2) F_{11} = \alpha_2$, $(\alpha_1, \alpha_2) F_{12} = \alpha_1$, and $(\alpha_1, \alpha_2, \alpha_3) G_1 = \alpha_1 \times \alpha_2 + \alpha_2 \times \alpha_3$. Thus, $\{2\}$, $\{1\}$, and $\{1,3\}, \{2,3\}$ are perfect sets of witnesses for $F_{11}, F_{21},$ and $G_1$, respectively. Figure 2 shows two infinite derivation trees from the initial nonterminal $(\{\}, \{\}, \{\})$ in the grammar $\text{Gr}$ associated with this definition, where we write 1 and 2 for the polywits $\{\{1\}\}$ and $\{\{2\}\}$. The trees repeat the same pattern after reaching $(\{1,2\}, \{\})$. In the left tree, the top production is $(\{1,2\}) t_i \rightarrow (\{1\}, \{2\}, \{\}) t_i$; this is a valid production of type 2, based on $G_1$’s witness $\{1,3\}$. The tree’s other production of type 2, $(\{2,1\}) t_i \rightarrow (\{1\}, \{2,1\}, \{\}) t_i$, uses the other witness, $J = \{2,3\}$. The frontiers of the two trees are $\{1\}$ and $\{2\}$, respectively. The set $\{\{1\}, \{2\}\}$ is perfect for $\text{alt}$. 
Even though \( alt \) is always nonempty (since it is a codatatype), a perfect set of witnesses is necessary to decide the overall nonemptiness problem. If we used an imperfect set such as \{\{1,2\}\}, we would reject valid datatypes such as \( \alpha \text{fractal} = \text{Fractal} \{((\alpha, (\alpha, \alpha) \text{alt}) \text{fractal}) \text{alt}\} \), where we must know that \{\{1\}\} is a witness for \( alt \) to infer nonemptiness.

V. (CO)INDUCTION PRINCIPLES

In a proof assistant, high-level abstractions are pointless unless they are supported by a reasoning apparatus. Just like the types themselves, reasoning principles for nonuniform (co)datatypes can be derived in HOL. To avoid cluttering the ideas with technicalities, in this and the next section we discuss the restricted situation of a single (co)datatype \( \alpha T \) defined as fixpoint isomorphism \( \alpha T \simeq \langle \alpha, \alpha F T \rangle G \) as in Section III-B but with \( i = j = k = 1 \). We reuse the infrastructure from that section but omit most indices.

(Co)induction involves reasoning about the elements of a (co)datatype and those of their (co)recursive components. BNFs allow us to capture components abstractly, in terms of the “set” operators. For example, for any element \( r \) of the uniform (co)datatype \( \alpha \text{raw} \), its components are the elements \( r' \) of \( \text{set}_G (\text{unRaw} r) \)—because, in its fixpoint definition, \( \text{raw} \) appears recursively as the second argument of \( G \).

A. Induction

Induction for uniform datatypes can be smoothly expressed in HOL. For example, the induction principle for \( \alpha \text{raw} \) is the following HOL theorem, which we will refer to as \( \text{Ind}_{\text{raw}} \):

\[
\forall Q. \left( \forall r : \alpha \text{raw}. \left( \forall r' \in \text{set}_G (\text{unRaw} r). Q r' \right) \Rightarrow Q r \right) \Rightarrow
\forall r : \alpha \text{raw}. Q r
\]

It states that to show that a predicate \( Q \) holds for all \( \alpha \text{raw} \), it suffices to show that \( Q \) holds for any element \( r \) given that \( Q \) holds for the recursive components \( \text{set}_G (\text{unRaw} r) \) of \( r \).

As we remarked in Section I, a verbatim translation of \( \text{Ind}_{\text{raw}} \) for \( T \) would not be typable, since \( Q \) would be a variable used with two different types. But even if we change \( Q \) from a quantified variable to a polymorphic predicate \( Q : \alpha \text{raw} \rightarrow \text{bool} \) (and remove the outer \( \forall r \)), the formula would be unsound, due to the cross-type nature of the \( T \) components: Whereas \( t \) has type \( \alpha T \), its components \( t' \) have type \( \alpha F T \). For example, if \( Q \) is vacuously false on the type \( \alpha F T \), we could use such an induction theorem (with \( \alpha \) instantiated to \( \text{nat} \)) to wrongly infer that \( Q \) is true on \( \text{nat} T \).

On the other hand, for each polymorphic predicate \( P : \alpha T \rightarrow \text{bool} \), we can hope to prove the following inference rule in HOL, where for clarity we make explicit the universal quantification over the type variable \( \alpha \), occurring both in the assumption and the conclusion

\[
\forall \alpha. \forall t : \alpha T. \left( \forall t' \in \text{set}_G (\text{dtor}) \rightarrow P t' \right) \Rightarrow P t \quad \text{Ind}_{\text{P}}^\alpha
\]

Let us try to prove this rule sound. All we have at our disposal is the representation type \( \alpha \text{raw} \) and its induction principle. So we should try to reduce \( \text{Ind}_{\text{P}}^\alpha \) to \( \text{Ind}_{\text{raw}} \) along the embedding-projection pair \( \text{Rep} : \alpha T \rightarrow \alpha \text{raw} \) and \( \text{Abs} : \alpha \text{raw} \rightarrow \alpha T \), where the predicate \( \text{ok} [] \) describes the image of \( \text{Rep} \).

We start by defining \( Q \) to be \( P \circ \text{Abs} \) and try to prove \( \forall r. \text{ok} [] \rightarrow Q r \) using \( \text{Ind}_{\text{raw}} \), hoping to be able to connect the hypothesis of \( \text{Ind}_{\text{P}}^\alpha \) with that of \( \text{Ind}_{\text{raw}} \). We quickly encounter the following problem, depicted in Figure 3. Suppose \( r : \alpha \text{raw} \) corresponds to \( t : \alpha T \) (via the embedding–projection pair); then \( T \)-induction speaks about the \( T \) components \( t' : \alpha F T \) of \( t \), which do not correspond to the \( \alpha \text{raw} \) components \( r' : \alpha \text{raw} \) of \( r \), but rather to elements \( r'' : \alpha F T \) of the form \( \downarrow r' \). This mismatch is a consequence of our representation technique: To represent \( T \)'s destructor using \( \text{raw} \)'s destructor, we applied the “correction” \( \downarrow \). To cope with it, we appeal to the shape type \( \alpha \text{sh} \), which in the simplified setting amounts to \( \alpha + \alpha F + \alpha F^2 + \cdots \) and thus includes all the types inhabited by \( t \), its components, the components’ components, and so on.

So we weaken our goal and try to prove that \( P \) holds for types of the form \( \alpha \text{sh} T \). For \( Q \), this means switching from \( \alpha \text{raw} \) to \( \alpha \text{sh} \alpha \text{raw} \). As shown in Figure 4, now we can travel from the type \( \alpha \text{sh} \alpha \text{raw} \) back to the type \( \alpha \text{sh} \alpha \text{sh} \alpha \text{raw} \) by applying Node to level the nonuniformity \( F \) into the larger type \( \text{sh} \). For this to work, \( Q \) must reflect \( \text{map}_{\text{sh}} \text{Node} \), i.e., have \( Q (\text{map}_{\text{sh}} \text{Node} r'') \) imply \( Q r'' \).

Another issue is that if \( r'' = \text{map}_{\text{sh}} \text{Node} r'' \) is not in the image of \( \text{Rep} \), \( r'' \) has shadow \( [1] \) instead of the required \([\] \).

We must generalize the goal to arbitrary shadows, i.e., to

1Starting with this figure, we replace \( \text{map}_G \text{Abs} f g \) arrow annotations with arrows carrying two labels: \( f \) to the right in the arrow’s direction of travel and \( g \) to the left.
∀r α. ok r u r ⇒ Q’ r u r, for a suitable predicate Q’ : Δ → α sh raw → bool that extends Q in that Q’ [] = Q. To this end, we define δ : Δ → α sh raw → α sh raw, an operator that generalizes the trip from r’ to r” to r’’ described above to an arbitrary shadow u, and δ, the cumulative iteration of δ:

\[
\delta \circ u = \text{map}_{\text{raw}} \; \text{Node} \; (\delta \; u) = \delta \; \text{map}_{\text{sh}} \; (\delta \; u) \quad \text{and} \quad (u \uparrow 1) = \delta \; u \; (u \; \uparrow r).
\]

Intuitively, we can regard the elements of both β sh and β raw as trees with β leaves and whose nodes branch according to F. Then δ traverses elements of α sh raw until it reaches their innermost nodes (with only leaves, i.e., elements of α sh, as subtrees) and immerses them as top nodes in the inner shape layer. The additional shadow argument u is needed to identify when an innermost tree has been reached (since we count on well-behavedness of δ u r only if ok δ u r).

The sh counterparts of the above, [[δ]] : Δ → α sh sh → α sh sh and [[δ]], are defined similarly (using map_{sh} instead of map_{raw}). The key property of the “immerse” family of operators is that they commute with raw’s destructor in the following sense.

**Lemma 7:** The left subdiagram in Figure 5 is commutative.

Now, we can take Q’ u r to be Q (δ u r). Q’ can be proved by raw-induction on r, since it achieves the desired correspondence between the raw components and the T components (i.e., between the leftmost and rightmost edges of Figure 5). That the correspondence works is ensured by the diagram’s commutativity, as a composition of two commutative subdiagrams (the left subdiagram by the above lemma and the right one by the definition of dtor).

Thus, assuming the hypothesis of Ind^T, we have proved ∀α. ∀r : α sh raw, ∀v : Δ, ok r u ⇒ Q’ r u—in particular, ∀α. ∀r : α sh raw, ok r u ⇒ Q’ r u, which implies ∀α. ∀r : α sh T, P t. From this, we prove the more general fact ∀α. ∀r : α sh T, P t. It would suffice that P reflects Leaf, α T → α sh T. We can simply assume that P is injective-antitone-parametric (IAP), meaning that P (map_T f t) implies P t for all t : α T and all injective functions f : α → β (including Leaf and Node). In conclusion:

**Theorem 8:** If P is IAP, then Ind^T is derivable in HOL.

IAP is substantially weaker than (general) parametricity, which for P would mean P t ⇔ P (map_T f t) for all t and arbitrary functions f [12, Appendix A].

Due to the limitations of HOL, we were able to prove only a restricted form of induction. However, all formulas built from the usual terms used in functional programming and employing equality, the logical connectives, and universal quantifiers are IAP (if not fully parametric), and therefore fall within the scope of our theorem. The main outcasts are constants defined using Hilbert choice, existential quantifiers, and ad hoc overloaded constants.

### B. Coinduction

We designed the above infrastructure, consisting of the “immerse” operators, to work equally well for the codatatype as it does for the datatype. When α T is a codatatype, these operators are defined in the same way and can be used to derive the soundness of a nonuniform coinduction rule under similar assumptions to the induction case (from the corresponding uniform coinduction on the raw type):

\[
\forallα. \forallT_1, T_2 : α T. \quad P \; t_1 \; t_2 \Rightarrow \text{rel}_G (\; \Rightarrow \; \; \text{map}_T \; t_1 \; t_2 \; \; \text{map}_T \; g \; t_2)
\]

For this rule to be sound, P : α T → α T → bool must again interact well with injective functions, but this time in the opposite direction. We say that P is injective-monotone-parametric (IMP) if P t_1 t_2 implies P (map_T f t_1) (map_T g t_2) for all t_1, t_2 : α T and injective functions f, g : α → β.

**Theorem 9:** If P is IMP, then Coind^T is derivable in HOL. Unlike IAP, IMP disallows the usage of the universal quantifier in P, while it allows the existential quantifier. This is a quite desirable symmetry: Induction requires the universal quantifier to perform generalization over non-inductive parameters. For coinduction, the existential quantifier takes this role.

### VI. (CO)RECURSION PRINCIPLES

For nonuniform (co)datatypes to be practically useful, there must exist some infrastructure supporting (co)recursive function definitions. We start with datatypes and consider the following simple recursive function on powerlists:

**split** : (α × β) plist → α plist × β plist

**split Nil** = (Nil, Nil)

**split (Cons (a, b) xs)** = let (as, bs) = split (map_{plist} swap xs) in (Cons a as, Cons b bs)

Here, the pattern-matched variable xs has type (((α × β) × (α × β)) plist), and the auxiliary swap function is defined as swap ((a_1, b_1), (a_2, b_2)) = ((a_1, a_2), (b_1, b_2)). The function split uses polymorphic recursion: Its type on the right-hand side of the specification is different from the one on the left-hand side. More precisely, the recursive call is applied to an argument of type ( ((α × α) × (β × β)) plist). None of the existing HOL-based tools for defining recursive functions support polymorphic recursion—the gap we are about to fill.
The split function is not primitively recursive in the standard sense: The recursive call is applied to a modified pattern-matched argument \( \text{map\_pl} \), \( \text{swap} \), \( \text{xs} \). However, the modification takes place through the \( \text{map\_pl} \) function, which leaves the length of \( \text{xs} \) unchanged. Hence, such \textit{generalized primitively recursive} specifications are terminating.

A. Recursion

Primitively recursive specifications in HOL are reduced to nonrecursive definitions using a recursion combinator \([10]\). The equally expressive but less convenient \textit{primitively iterative specifications} can be reduced as well, using a simpler fold combinator. For a uniform datatype \( \alpha T = \text{Ctor} ((\alpha, \alpha T) G) \) (e.g., \( \alpha T = \alpha \text{ list} \) with \((\alpha, \beta) G = \text{unit} + \alpha \times \beta \)), the fold combinator has type \((\alpha, \beta) G \rightarrow \beta \alpha T \rightarrow \beta \). The recursion combinator’s blueprint, of \( \beta \) is very restrictive, because it forces the type argument \( \alpha \) to be the second argument enables recursive functions of a more refined type \( \alpha \). The explicit universal quantification over \( \alpha \) indicates that the blueprint needs to be truly polymorphic in \( \alpha \).

The primitive iteration schema provided by the above combinator is very restrictive, because it forces the type argument \( \beta \) of \( T \) to be fully polymorphic. Neither the split function nor a simple summation of a powerlist can be expressed using \( \text{map\_pl} \) function.

A function \( f = \text{fold}\ b \) for some fixed \( b : (\alpha, \beta) G \rightarrow \beta \), satisfies the characteristic recursive equation \( f (\text{Ctor} g) = b (\text{map\_G} \ id \ f \ g) \). We call \( b \) the \textit{blueprint} of \( f \). Note that \( b \) describes how to combine the \textit{results} of the recursive calls into a new result of type \( \beta \). The recursion combinator’s blueprint of type \((\alpha, \alpha T \times \beta) G \rightarrow \beta \), generalizes fold’s blueprint by providing access to the original \( \alpha T \) values, in addition to the results of the recursive calls. For simplicity, we focus on iteration.

For a nonuniform datatype \( \alpha T = (\alpha, \alpha F T) G \), the natural generalization of fold would be a combinator of type

\[
\forall Y. (\forall a. (\alpha, \alpha F Y) G \rightarrow \alpha Y) \rightarrow (\forall a. \alpha X F \rightarrow \alpha F X) \rightarrow (\forall a. \alpha Y F \rightarrow \alpha Y) \rightarrow \beta X T \rightarrow \beta Y
\]

where the universally quantified type constructor \( Y \) captures the positions where \( a \) have to be replaced by \( \alpha F \), since the recursive calls will be applied to a term of type \( \alpha F T \). The explicit universal quantification over \( \alpha \) indicates that the blueprint needs to be truly polymorphic in \( \alpha \).

The second argument enables recursive functions of a more refined type \( \beta X T \rightarrow \beta Y \) by providing a distributive law \( a : \forall a. \alpha X F \rightarrow \alpha F X \) which we call the \textit{argument} \( \text{swapper} \). Bird and Paterson require \( X \) and \( Y \) to be functors and the arguments \( b \) and \( a \) to be natural transformations. The function \( f = \text{fold}\ b \) is then a natural transformation as well and satisfies

\[
f (\text{Ctor} g) = b (\text{map\_G} \ id \ f \circ \text{map\_F} \ a) \ g
\]

It is straightforward to allow functors \( X \) and \( Y \) to be of arbitrary arity \( n \). The function split can then be defined by setting \( n = 2 \), \((\alpha, \beta) G = \text{unit} + \alpha \times \beta, \alpha F = \alpha \times \alpha, (\alpha, \beta) X = \alpha \times \beta, (\alpha, \beta) Y = \alpha \text{ plist} \times \beta \text{ plist}, a = \text{swap}, \) and

\[
b (\text{lnl} (\_)) = (\text{Nil}, \text{Nil})
\]

\[
b (\text{lnr} ((\alpha, \beta), y s)) = \text{let} (a s, b s) = y s
\]

in (\( \text{Cons} \ a s, \text{Cons} \ b s \))

where \( \text{lnl} \) and \( \text{lnr} \) are the left and right embeddings of \( + \). For simplicity, the rest of this section assumes \( n = 1 \).

We propose an even more flexible fold combinator that replaces the functor \( F \), which is fixed in the nonuniform datatype specification and appears in the recursive calls, with another arbitrary functor \( V \) of the same arity as \( F \) (here, 1):

\[
\forall X Y. (\forall a. (\alpha X, \alpha V Y) G \rightarrow \alpha Y) \rightarrow (\forall a. \alpha X F \rightarrow \alpha V X) \rightarrow (\forall a. \alpha Y F \rightarrow \alpha Y) \rightarrow \beta X T \rightarrow \beta Y
\]

This allows the recursive calls to return a type \( \alpha V Y \) instead of the fixed \( \alpha F Y \). The combinator satisfies the same characteristic equation \( (2) \) (with the more general types).

All these expressive combinators for nonuniform types have one problem in common: In HOL, type constructor quantification and type variable quantification can only occur at the outermost level. Thus, it is impossible to define the fold constants for nonuniform datatypes.

Instead, we follow a similar route as for induction. We devise a \textit{recursion procedure} that takes (here, unary) BNFs \( V, X, \) and \( Y \), a blueprint \( b : (\alpha, \alpha V Y) G \rightarrow \alpha Y \) and a \textit{swapper} \( a : \alpha X F \rightarrow \alpha V X \) as input and yields a function \( f : \alpha X T \rightarrow \alpha Y \) satisfying equation \( (2) \).

The procedure defines a recursive function using \( b \) and \( a \) on the \textit{raw} type and lifts it to the nonuniform type. To perform such a lifting for induction, the inductive property \( P \) must be a polymorphic IAP term. For recursion, we require both \( b \) and \( a \) to be polymorphic injective-parametric terms, i.e., parametric only for relations that are graphs of injective functions. This is a weaker assumption than Bird and Paterson’s naturality assumption (e.g., \( \text{map\_F} (\text{map\_X} f) = \text{map\_X} (\text{map\_V} f) \circ a \) for \( a \)). On (bounded) natural functors, injective-parametricity implies the weak naturality assumption that demands for the above equation to hold only for injective functions \( f \). Consequently, \( f \) will also only be a natural transformation for injective functions. By contrast, our construction is closed: If both \( b \) and \( a \) are fully parametric in some type parameters, \( f \) is fully parametric in the same type parameters.

The definition of \( f \) proceeds in four steps. First, we define a shape type \( \text{sh}_V \) for \( V \) analogously to \( sh \) for \( F \), including the constructors \( \text{Leaf}_V, \text{Node}_V \), their inverses \( \text{unLeaf}_V, \text{unNode}_V \), and the functions \( \text{ok}_X, \text{ok}_V \), and \( \text{ok}_Y \). Second, we lift \( a \) to shapes \( \overline{a} : \Lambda \rightarrow \alpha X sh \rightarrow \alpha sh V X \) by recursion on the shadow:

\[
\text{ok}_V = \text{map\_X} \text{Leaf}_V \circ \text{unLeaf} \\
\text{ok}_V (1 \circ u) = \text{map\_X} \text{Node}_V \circ a \circ \text{map\_F} (\overline{\text{u}}) \circ \text{unNode}
\]

Third, we define a \textit{raw} version of our function \( f_{\text{raw}} : \Lambda \rightarrow \alpha X T \rightarrow \alpha sh V Y \) by primitive recursion:

\[
f_{\text{raw}} (\text{Raw} g) = b (\text{map\_G} (\overline{\text{u}}) (\text{map\_V} \text{unNode}_V \circ f_{\text{raw}} (1 \circ u) g) \ g)
\]

The generalization to \( sh_V \) in the return type of \( f_{\text{raw}} \) is similar to what we did for induction. Finally, we define the function \( f \) as \( f = \text{map\_V} \text{unLeaf}_V \circ f_{\text{raw}} (\text{rep}) \).

Figure 6 justifies the above definitions by proving equation \( (2) \). Some of the arrows labeled by injective functions,}

\footnote{Strictly speaking, the boundedness assumption is not needed for \( X \) and \( Y \), which implies that \( a \) set is permitted to occur in those type expressions.}
such as \( \text{Leaf}_Y \) and \( \text{Node}_Y \) (possibly under further maps),
must be inverted for the diagram to make sense. Elements of
the two highlighted types have shadow \([1]\). All other elements
of types \( sh, sh_Y \), and \( \text{raw} \) in the diagram have shadow \([\]\).

Equation (2) is the outermost pentagon, which is filled by
commutative diagrams starting by unfolding the definitions of
\( f \) \( (\) twice, \( \text{Ctor} \) \( 2 \) \), and \( \text{map}_T \) \( 3 \) as well as the recursive
specification of \( f_{\text{raw}} \) \( 4 \). The quadrilateral \( 5 \) follows from
the naturality for injective functions (\( \text{Leaf}_V \)) of \( b \) and \( 6 \)
from the recursive specification of \( \Box_V \). The remaining commutative
pentagon \( 7 \) relates \( f_{\text{raw}} \) and \( \uparrow \) (similarly to Lemma 5 for
\( \text{map}_{\text{raw}} \)); the proof follows by induction. Therefore,
the property used in \( 7 \) for shadow \([\]\) must be generalized to
an arbitrary \( u \) and requires an auxiliary fact about \( a \) and \([\]\)
together with the facts that \( \Box \) and \( f_{\text{raw}} \) preserve \( \Box_u \) and
\( \text{ok}_u \). The proofs rely on the injective-parametricity of \( a \) and \( b \).

**Lemma 10:**
1) \( \Box_b u s \Rightarrow \text{pred}_{sh_Y} (\Box_b u) (\Box s) u s \);  
2) \( \text{ok}_u r \Rightarrow \text{pred}_{sh_Y} (\Box_b u) (f_{\text{raw}} u) r \);  
3) \( \Box_b u s \Rightarrow b (u \uparrow 1) (\Box s) = \text{map}_Y (\Box b) (\Box u (\text{map}_{\text{sh}} a s)) \);  
4) \( \text{ok}_u r \Rightarrow f_{\text{raw}} (u \uparrow 1) (\uparrow s) = \text{map}_Y (\Box r u (\text{map}_{\text{raw}} a s)) \).

Requiring \( Y \) to be a functor is very restrictive, because
it disallows many recursive functions with parameters. We
generalize the entire construction to \( a Y = a Y_1 \rightarrow a Y_2 \),
where \( Y_1 \) and \( Y_2 \) are natural functors. This allows first-order
arguments as well as higher-order arguments that do not refer
to \( a \) in their domain. This generalization is straightforward but
technically involved.

**B. Corecursion**

The recursion procedure is designed to work dually for
codatatypes. The corecursion procedure mainly reverses function
arrows: It takes the injective-parametric blueprint \( b : a Y \rightarrow b : (a X, a V Y) G \) and swapper \( a : a V X \rightarrow a X F \) as inputs and yields the function \( f : a Y \rightarrow a X F \), which satisfies
\( f_y = \text{Cons} (\text{map}_T \text{id} (\text{map}_T a o f) (b y)) \).

**VII. IMPLEMENTATION**

To add support for nonuniform types to Isabelle/HOL, we
followed the same general strategy as previously \([10]\):

1) We formalized in Isabelle/HOL an abstract datatype example
\( a T = \text{Ctor} ((a, a F T) G) \) as well as a codatatype.

2) We developed ML functions that generalize the abstract
examples to produce the derivations for a concrete set of
mutual types with an arbitrary number of type variables
and to derive the nonemptiness witnesses.

3) We developed ML functions that extend the results of
step 2 to multiple curried constructors—the high-level
view presented to users.

4) We developed the commands that process type and func-
tion definitions and that perform (co)induction.

For datatypes, step 1 starts by defining the type \( a T \), \( \text{Ctor} \),
and the BNF constants; then it derives theorems about them
and registers \( T \) as a BNF. This registration is performed by
an existing Isabelle command that lifts the BNF structure of
a type across an embedding–projection pair \([6]\). Induction is
formalized by deriving a lemma \( Q t \) in terms of a fixed but
unknown polymorphic predicate \( Q \) that is IAP and inductive
(i.e., \( \forall x \in \text{set}_{\Box} G \). \( Q x \) \( \Rightarrow \) \( Q \) \( (\text{Ctor} g) \) holds). Recursion is
formalized as a function \( f \) defined such that the recursive
equation \( f (\text{Ctor} g) = b (\text{map}_G \text{id} (f \circ \text{map}_T a) g) \) holds for
a fixed injective-parametric blueprint \( b \) and swapper \( a \).

The code for step 2 constructs the low-level types, terms,
and lemma statements presented in Sections III to VI and
proves the lemmas using specialized tactics—ML programs
that generalize the proofs from the formalization. In principle,
the tactics should always succeed, but it is necessary to
execute them to obtain the highest level of trustworthiness.
Assuming Isabelle’s inference kernel is correct, bugs in the
new commands might lead to run-time failures but never to
logical inconsistencies. For step 3, we were able to generalize
and reuse the infrastructure for uniform types that performs
the same lifting from low to high level \([10, \text{Sections 3–6}]\).

Step 4 takes the form of six main commands available to the
users and making definitions and reasoning about nonuniform
types almost as convenient as for uniform types.

The \texttt{nonuniform\_codatatype} commands can be used to
define nonuniform types. For example, the following definition
introduces a type of \( \lambda \)-terms over variables drawn from \( \alpha \), with
De Bruijn notation for bound variables \([8]\):

\[
\text{Var} \alpha \mid \text{App} (\alpha \text{tm}) (\alpha \text{tm}) \mid \text{Lam} ((\text{unit} + \alpha) \text{tm})
\]
Enter a $\lambda$-abstraction (Lam) creates a new variable, which is accommodated by the extended type $unit + \alpha$ consisting of the values $\lnl ()$ (the new variable) and $\lnr x \text{ for all } x : \alpha$. The command performs the type construction and computes a nonemptiness witness. Then it defines the constructors $\var$, $\app$, $\lam$ and corresponding destructors and derives characteristic theorems about the constructors, the destructors, and the BNF constants $\map{pm}$, $\pred{tm}$, $\rel{tm}$, and $\set{tm}$.

The nonuniform_prim(co)recursive commands allow users to define primitively (co)recursive functions, by specifying their (co)recursive equations.\footnote{The implementation of these two commands is incomplete at the time of this writing. We do not foresee any difficulties beyond those which we met for the other commands and expect to finish the implementation in the weeks following the submission deadline. The archive \cite{11} will be updated.} For example, the following definition introduces a function join that “flattens” a term whose variables are themselves terms:

\[
\begin{align*}
\text{nonuniform_primrecursive join} & : \alpha \tm \tm \rightarrow \alpha \tm \\
\text{join} & (\var x) = x \\
\text{join} & (\app s t) = \app (\text{join} s) (\text{join} t) \\
\text{join} & (\lam u) = \lam (\map{pm} \beta \text{ case } x \text{ of } \\
\lnl () & \Rightarrow \var (\lnl ()) \mid \lnr y \Rightarrow \map{pm} \lnr y (x)) \end{align*}
\]

The command extracts blueprints and swappers from the equations and emits parametricity proof obligations that must be discharged by the user. In the example, theswiper is the $\lambda$-expression of type $\alpha \tm \tm \rightarrow (\alpha \tm + \alpha) \tm$ that is passed to the outer $\map{pm}$. Once the proofs are complete, the command derives a low-level characteristic theorem about the defined function, from which the user-specified equations follow.

One of the most basic operations on $\lambda$-terms is substitution:

\[
\begin{align*}
\text{sub} & : (\alpha \rightarrow \beta \tm) \tm \rightarrow \alpha \tm \tm \rightarrow \beta \tm \\
\text{Due to the limitation that arguments to recursive functions must be BNFs, we cannot define higher-order functions like sub that depend on a type variable that changes in the recursive calls. But we can define subst as a composition: subst } \sigma & = \text{ join } \circ \map{pm} \sigma. \\
\text{The nonuniform_(co)induct commands can be used to prove a lemma by (co)induction. For example, the command nonuniform_induct } s & \text{ in subst_subst: } \\
\text{subst } \tau (\text{subst } \sigma s) & = \text{subst } (\text{subst } \tau \circ \sigma) s
\end{align*}
\]

emits proof obligations for parametricity and the three cases of the induction on $s$. Often, the parametricity proofs can be delegated to Isabelle’s Transfer tool \cite{27}. Once the obligations are discharged, the stated property is derived and stored under the specified name (subst_subst). For the technical reason given in Section V, the derivation can be performed only by an Isabelle command, not by a proof method as is done for uniform (co)datatypes \cite{10}. Unlike commands, proof methods can be invoked on arbitrary proof goals in the middle of a proof.

As a simple example involving nonuniform codatatypes, we prove the equivalence between two definitions of the constant powerstream. The required proofs are fully automatic after specifying the trivial bisimulation relation $\text{join r} \Leftrightarrow \exists x . \tm x$. $l = \text{const } x \land r = \map{pstream} \tau \tm x$ in the coinduction proof:

\[
\begin{align*}
\text{nonuniform_codatatype } \alpha \tm pstream = \text{ Cons } \alpha \beta x \text{ in the coinduction proof: } \\
\end{align*}
\]

VIII. DISCUSSION AND RELATED WORK

a) Inspiration: We generalized Okasaki’s construction \cite{36} to a large class of datatypes. Nordhoff et al. \cite{35} partially relied on this construction (defining the sh and raw types but without introducing a new nonuniform type) in their Isabelle/HOL formalization of 2-3 finger trees. We have not found any corresponding reduction of nonuniform to uniform codatatypes in the literature.

For recursion, we refined Bird and Paterson’s generalized fold combiners \cite{9} in several ways, including weakening the parametricity/naturality condition and enabling non-functor target domains. In turn, Bird and Paterson had improved on the standard sheaf-functor approach from category theory \cite{28}.

Our (co)induction principles take advantage of the BNF structure including set operators and relaturs. They constitute a lightweight alternative to fibration-based approaches \cite{18, 23} for the category of sets and functions.

b) Comparison with Other Proof Assistants: Our work shows that nonuniform (co)datatypes and the associated polymorphic (co)recursion \cite{22, 33} can be supported in the minimalistic rank-one polymorphic framework of HOL, and therefore made available in HOL-based proof assistants, which cover about half of the theorem proving community.

The dependent type theory (DTT) camp, represented by Agda, Coq, Matita, and Lean, has sophisticated type systems built into their mechanized logic, including native nonuniform datatypes. Several case studies in these proof assistants exploit nonuniformity \cite{4, 16, 26, 34, 40}.

Compared with the DTT systems, our support for nonuniform types in HOL has some limitations. Obviously, dependent families of (nonuniform) types cannot be expressed in HOL. In addition, Agda supports self-nested (co)datatypes—e.g., $\alpha$ bush = BNil $|$ BCons $\alpha$ (BCons bush bush). Moreover, our (co)induction principles have some restrictions concerning (a weak form of) parametricity. The reason is that we cannot perform well-founded induction across types in HOL. Although practical predicates about functional programs obey them, the restrictions are not necessary semantically. Our technical report \cite[Appendix D]{12} presents an axiomatic extension that eliminates them. However, adding axioms, regardless of how provably correct they may be, is generally frowned upon by users of HOL-based systems.

Our approach derives some advantages from its category-theoretical orientation. First, arbitrary parameter types, and not only (co)datatypes, are allowed in the specifications for nonuniform types, either inside or outside of the recursive occurrences in the specification. For example, the type $\text{stree}$ from Section I is possible because the type $\alpha \text{ fset}$ of finite sets is a BNF. This is excluded with DTT, which restricts datatypes to a predefined grammar. Second, since the foundational
approach compels us to maintain the functorial structure to justify fixpoint definitions, users enjoy map functions and relators, as well as some polytypic properties directly or within immediate reach. Our nonuniform recursion principle delivers parametric functions, i.e., natural transformations. Moreover, the fusion laws [9] [12, Appendix C], which help reasoning about functional programs, depend on functoriality and naturality, and they are immediate in our framework. By contrast, in DTT, little structure is available for nonuniform datatypes after definition. In particular, map functions and relators are missing and can be tricky to add for some types.

c) Other Work: The pioneering work of Bird and his collaborators on nonuniform datatypes [7], [8], [9] has been extended in several directions, including structures for efficient functional programming [24], [25], [29], datatypes with references [17], as well as work directly relevant for DTT proof assistants: reduction to W-types and container types [1], typed term rewriting frameworks for total programming [2], [3], [30], and induction in intensional DTT [31]. Our contribution was concerned with bootstrapping nonuniform datatypes in HOL on a sound and compositional basis. Time will tell if Isabelle/HOL users, or more generally the HOL community of users and researchers, will embrace nonuniform datatypes and their applications to the same extent as in advanced programming languages and type theories.

Acknowledgment: We thank David Basin for supporting this research, Peter Lammich for pointing us to his work on finger trees and for formalizing Okasaki’s construction for powerlists, Johannes Hölzl for taking the time to explain Lean’s nonuniform datatypes, and Mark Summerfield and the anonymous reviewers for suggesting textual improvements. Blanchette has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Popescu has received funding from UK’s textual improvements. Blanchette has received funding from the

REFERENCES