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):

$$\alpha\ list = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\alpha\ list)$$

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:

$$\alpha\ plist = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ ((\alpha \times \alpha)\ plist)$$

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

$$\mathsf{Cons}\ a\ (\mathsf{Cons}\ (b_1, b_2)\ (\mathsf{Cons}\ ((c_{11}, c_{12}), (c_{21}, c_{22}))\ \mathsf{Nil}))$$

Nonuniform datatypes arise in the implementation of efficient functional data structures, such as finger trees [24], and they underlie Okasaki's bootstrapping and implicit recursive slowdown optimization techniques [35]. 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 [12, Section 1].

978-1-5090-3018-7/17/$31.00 ©2017 IEEE

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 folklore 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:

$$\begin{aligned}\alpha\ ptree &= \mathsf{Node}\ \alpha\ (\alpha\ pforest)\\ \alpha\ pforest &= \mathsf{Nil} \mid \mathsf{Cons}\ (\alpha\ ptree)\ ((\alpha \times \alpha)\ pforest)\end{aligned}$$

They allow multiple recursive occurrences, with different type arguments:

$$\alpha\ plist' = \mathsf{Nil} \mid \mathsf{Cons}_1\ \alpha\ (\alpha\ plist') \mid \mathsf{Cons}_2\ \alpha\ ((\alpha \times \alpha)\ plist')$$

They allow multiple type arguments, which may all vary:

$$(\alpha, \beta)\ tplist = \mathsf{Nil}\ \beta \mid \mathsf{Cons}\ \alpha\ ((\alpha \times \alpha,\ unit + \beta)\ tplist)$$

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:

$$\alpha\ stree = \mathsf{Node}\ \alpha\ (((\alpha\ fset)\ stree)\ fset)$$

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

$$\alpha\ pstream \overset{\infty}{=} \mathsf{Cons}\ \alpha\ ((\alpha \times \alpha)\ pstream)$$

introduces "powerstreams," with infinite values of the form $\mathsf{Cons}\ a\ (\mathsf{Cons}\ (b_1, b_2)\ (\mathsf{Cons}\ ((c_{11}, c_{12}), (c_{21}, c_{22}))\ \ldots))$.

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 [13], 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 *plist* should look like this:

$$\forall Q.\ Q\ \mathsf{Nil} \wedge \big(\forall x\, xs.\ Q\ xs \Longrightarrow Q\ (\mathsf{Cons}\ x\ xs)\big) \Longrightarrow \forall ys.\ Q\ ys$$

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

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 λ-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 [14]. It is the logic of Gordon's original HOL system [18] 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 *bool* of Booleans, and an infinite type *ind* using the function type constructor; thus, $(bool \to \alpha) \to ind$ is a type. Primitive *constants* are equality $= : \alpha \to \alpha \to bool$, the Hilbert choice operator, and 0 and Suc for *ind*. Terms are built from constants and variables by means of typed λ-abstraction and application.

A *polymorphic type* is a type T that contains type variables. If T is polymorphic with variables $\overline{\alpha} = (\alpha_1, \ldots, \alpha_n)$, we write $\overline{\alpha}\ T$ instead of T. *Formulas* are closed terms of type *bool*. The logical connectives and quantifiers on formulas are defined using the primitive constants—e.g., True as $(\lambda x : bool.\ x) = (\lambda x : bool.\ x)$. Polymorphic formulas are thought of as universally quantified over their type variables. For example, $\forall x : \alpha.\ x = x$ really means $\forall \alpha.\ \forall x : \alpha.\ x = x$. Nested type quantifications such as $(\forall \alpha.\ \ldots) \Longrightarrow (\forall \alpha.\ \ldots)$ 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 $\overline{\alpha}\ T$ and predicate $P : \overline{\alpha}\ T \to bool$ such that $\{x : \overline{\alpha}\ T \mid P\ x\}$ is nonempty, we can introduce a type $\overline{\alpha}\ S$ isomorphic to $\{x : \overline{\alpha}\ T \mid P\ x\}$. Upon meeting the definition $\overline{\alpha}\ S = \{x : \overline{\alpha}\ T \mid P\ x\}$, the system requires the user to prove $\exists x : \overline{\alpha}\ T.\ P\ x$ and then introduces the type $\overline{\alpha}\ S$, the *projection* $\mathsf{Rep}_S : \overline{\alpha}\ S \to \overline{\alpha}\ T$, and the *embedding* $\mathsf{Abs}_S : \overline{\alpha}\ T \to \overline{\alpha}\ S$ such that $\forall x.\ P\ (\mathsf{Rep}_S\ x)$, $\forall x.\ \mathsf{Abs}_S\ (\mathsf{Rep}_S\ x) = x$, and $\forall x.\ P\ x \Longrightarrow \mathsf{Rep}_S\ (\mathsf{Abs}_S\ x) = x$. 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 [18], [36].

Thus, unlike dependent type theory, HOL does not have (co)datatypes as primitives. However, datatypes [5], [19], [20], [31], [40] and, more recently, codatatypes [40] 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 *ind* and $\to$); defines the constructors and related operators; and proves characteristic properties, such as injectivity of constructors, induction theorems, and recursor theorems.

B. Bounded Natural Functors

We take uniform (co)datatypes for granted, thus assuming the availability of types such as $\alpha\ list$. Often it is useful to think in terms of type constructors. For example, *list* is a type constructor in one variable, sum $(+)$ and product types $(\times)$ are binary type constructors. Most type constructors are not only operators on types but have a richer structure, that of *bounded natural functors* (*BNF*s) [40].

We write $[n]$ for $\{1, \ldots, n\}$ and $\alpha\ set$ for the powertype of α, consisting of sets of α elements; it is isomorphic to $\alpha \to bool$. An n-ary BNF is a tuple $F = (F, \mathsf{map}_F, (\mathsf{set}^i_F)_{i\in[n]}, \mathsf{bd}_F)$, where

- F is an n-ary type constructor;
- $\mathsf{map}_F : (\alpha_1 \to \beta_1) \to \cdots \to (\alpha_n \to \beta_n) \to \overline{\alpha}\ F \to \overline{\beta}\ F$;
- $\mathsf{set}^i_F : \overline{\alpha}\ F \to \alpha_i\ set$ for $i \in [n]$;
- bd_F is an infinite cardinal number

satisfying the following properties:

- (F, map_F) is an n-ary weak-pullback-preserving functor;
- each set^i_F is a natural transformation between the functor (F, map_F) and the powerset functor (set, image);
- $\forall i \in [n].\ \forall a \in \mathsf{set}^i_F\ x.\ f_i\ a = g_i\ a \Longrightarrow \mathsf{map}_F\ \overline{f}\ x = \mathsf{map}_F\ \overline{g}\ x$;
- $\forall i \in [n].\ \forall x : \overline{\alpha}\ F.\ \big|\mathsf{set}^i_F\ x\big| \leq \mathsf{bd}_F$.

We regard the elements of $\overline{\alpha}\, F$ as containers filled with "content" from α_i—the set^i_F functions return the α_i-content (known to be bounded by bd_F) and $\mathsf{map}_F\ \overline{f}$ replaces content via $\overline{f}$. For example, *list* is a unary BNF, where map_{list} is the standard map function, set_{list} collects all the lists's elements, and bd_{list} is $\aleph_0$.

BNFs are closed under uniform (least and greatest) fixpoint definitions [40] and the nonemptiness problem for BNFs constructed by basic functors, fixpoints and composition is decidable [13]. 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 structure [38]. The *predicator* $\mathsf{pred}_F : (\alpha_1 \to bool) \to \cdots \to (\alpha_n \to bool) \to \overline{\alpha}\ F \to bool$ and the *relator* $\mathsf{rel}_F : (\alpha_1 \to \beta_1 \to bool) \to \cdots \to (\alpha_n \to \beta_n \to bool) \to \overline{\alpha}\ F \to \overline{\beta}\ F \to bool$, are defined from set_F and map_F as follows:

- $\mathsf{pred}_F\ \overline{P}\ x \Leftrightarrow \forall i \in [n].\ \forall a \in \mathsf{set}^i_F\ x.\ P_i\ a$;
- $\mathsf{rel}_F\ \overline{R}\ x\ y \Leftrightarrow \exists z.\ (\forall i \in [n].\ \mathsf{set}^i_F\ z \subseteq \{(a,b) \mid R_i\ a\ b\}) \ \wedge\ \mathsf{map}_F\ \overline{\mathsf{fst}}\ z = x\ \wedge\ \mathsf{map}_F\ \overline{\mathsf{snd}}\ z = y$, where fst and snd are standard projection functions on the product type $\times$.

For *list*, $\mathsf{pred}_{list}\ P\ xs$ states that P holds for all elements of the list xs, and $\mathsf{rel}_{list}\ R\ xs\ ys$ states that xs and ys have the same length and are element-wise related by R.

Relators and predicators are useful to track parametricity [37], [41]. A polymorphic constant $\mathsf{c} : \overline{\alpha}\ F$ is *parametric* if, for all relations $R_i : \alpha_i \to \beta_i \to bool$ for each $i \in [n]$, we have $\mathsf{rel}_F\ \overline{R}\ \mathsf{c}\ \mathsf{c}$—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 $\mathsf{f} : \overline{\alpha}\ F \to \overline{\alpha}\ G$ between two BNFs, f is parametric if and only if f is a natural transformation (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 [35, Section 10.1.1] sketches how to mimic nonuniform datatypes using uniform datatypes. He approximates powerlists by the following definitions:

$\mathtt{datatype}\ \alpha\ sh = \mathsf{Leaf}\ \alpha \mid \mathsf{Node}\ (\alpha\ sh \times \alpha\ sh)$
$\mathtt{datatype}\ \alpha\ raw = \mathsf{Nil}_0 \mid \mathsf{Cons}_0\ (\alpha\ sh)\ (\alpha\ raw)$

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

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

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, $\boxed{\mathsf{ok}} : nat \to \alpha\ sh \to bool$ and $\mathsf{ok} : nat \to \alpha\ raw \to bool$, as the least predicates satisfying the following rules:

$$\begin{array}{ll} \boxed{\mathsf{ok}}\ 0\ (\mathsf{Leaf}\ x) & \boxed{\mathsf{ok}}\ n\ l \wedge \boxed{\mathsf{ok}}\ n\ r \Longrightarrow \boxed{\mathsf{ok}}\ (n+1)\ (\mathsf{Node}\ (l,r)) \\ \mathsf{ok}\ n\ \mathsf{Nil}_0 & \boxed{\mathsf{ok}}\ n\ x \wedge \mathsf{ok}\ (n+1)\ xs \Longrightarrow \mathsf{ok}\ n\ (\mathsf{Cons}_0\ x\ xs) \end{array}$$

The predicate $\boxed{\mathsf{ok}}\ n\ t$ checks whether t is a full binary tree of depth n, and $\mathsf{ok}\ n\ xs$ 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: $\alpha\ plist = \{xs : \alpha\ raw \mid \mathsf{ok}\ 0\ xs\}$.

The second disadvantage is addressed by hiding the internal construction of $\alpha\ plist$. We define the powerlist constructors $\mathsf{Nil} : \alpha\ plist$ and $\mathsf{Cons} : \alpha \to (\alpha\times\alpha)\ plist \to \alpha\ plist$ in terms of Nil_0 and 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 $\alpha\ plist = \mathsf{Ctor}_{plist}\ (unit + \alpha \times (\alpha\times\alpha)\ plist)$. It corresponds to finding a least solution (up to isomorphism) to the type fixpoint equation $\alpha\ plist \simeq (\alpha, (\alpha\ F)\ plist)\ G$ with $\alpha\ F = \alpha\times\alpha$ and $(\alpha,\beta)\ G = unit + \alpha\times\beta$.

We generalize this setting in multiple dimensions. First, we support a simultaneous definition of an arbitrary number $\mathbf{i}$ of *mutual* nonuniform datatypes. For example, *ptree* and *pforest* from Section I are given by the system of fixpoint equations

$$\begin{aligned} \alpha\ ptree &\simeq (\alpha, (\alpha\ F_1)\ ptree, (\alpha\ F_2)\ pforest)\ G_1 \\ \alpha\ pforest &\simeq (\alpha, (\alpha\ F_3)\ ptree, (\alpha\ F_4)\ pforest)\ G_2 \end{aligned}$$

where $(\alpha,\beta,\gamma)\ G_1 = \alpha\times\gamma$, $(\alpha,\beta,\gamma)\ G_2 = unit + \beta\times\gamma$, $\alpha\ F_1 = \alpha\ F_2 = \alpha\ F_3 = \alpha$, and $\alpha\ F_4 = \alpha\times\alpha$. 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 β.

Second, a type may occur several times on the right-hand side of a definition. We support an arbitrary number $\mathbf{j}$ of such *occurrences*. This feature is used in the $plist'$ type: $\alpha\ plist' \simeq (\alpha, (\alpha\ F_1)\ plist', (\alpha\ F_2)\ plist')\ G$, where $(\alpha,\beta,\gamma)\ G = unit + \alpha\times\beta + \alpha\times\gamma$, $\alpha\ F_1 = \alpha$, and $\alpha\ F_2 = \alpha\times\alpha$.

Finally, the construction supports an arbitrary number $\mathbf{k}$ of *type parameters*. The parameter changes may differ for different type parameters, such as in the *tplist* example: $(\alpha,\beta)\ tplist \simeq (\alpha,\beta,((\alpha,\beta)\ F_1, (\alpha,\beta)\ F_2)\ tplist)\ G$, where $(\alpha,\beta,\gamma)\ G = \beta + \alpha\times\gamma$, $(\alpha,\beta)\ F_1 = \alpha\times\alpha$, and $(\alpha,\beta)\ F_2 = unit + \beta$. 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 $[\mathbf{i}]$, $[\mathbf{j}]$, and $[\mathbf{k}]$, respectively. Moreover, we abbreviate indexed sequences using a horizontal bar; for example, $\overline{\alpha}$ stands for $\alpha_1, \ldots, \alpha_{\mathbf{k}}$,

$$
\begin{tikzcd}[column sep=huge]
(\overline{\alpha}, (\overline{\overline{\alpha}\, F_1})\; T_{\sigma(1)}, \ldots, (\overline{\overline{\alpha}\, F_{\mathbf{j}}})\; T_{\sigma(\mathbf{j})})\; G_i \arrow[rr, shift left, "\mathsf{map}_{G_i}\, \overline{\mathsf{id}}\, \mathsf{Rep}_{\sigma(1)} \ldots \mathsf{Rep}_{\sigma(\mathbf{j})}"] \arrow[d, shift right, "\mathsf{Ctor}_i"'] & & (\overline{\alpha}, (\overline{\overline{\alpha}\, F_1})\; \overset{\boxed{[]}}{raw_{\sigma(1)}}, \ldots, (\overline{\overline{\alpha}\, F_{\mathbf{j}}})\; \overset{\boxed{[]}}{raw_{\sigma(\mathbf{j})}})\; G_i \arrow[ll, shift left, "\mathsf{map}_{G_i}\, \overline{\mathsf{id}}\, \mathsf{Abs}_{\sigma(1)} \ldots \mathsf{Abs}_{\sigma(\mathbf{j})}"] \arrow[d, shift left, "\mathsf{map}_{G_i}\, \mathsf{Leaf}_1 \ldots \mathsf{Leaf}_{\mathbf{k}}\, \uparrow_{1\sigma(1)} \ldots \uparrow_{\mathbf{j}\sigma(\mathbf{j})}"] \\
\overline{\alpha}\; T_i \arrow[u, shift right, "\mathsf{dtor}_i"'] \arrow[r, shift left, "\mathsf{Rep}_i"] & \underset{\boxed{[]}}{\overline{\alpha}\; raw_i} \arrow[l, shift left, "\mathsf{Abs}_i"] \arrow[r, shift left, "\mathsf{unRaw}_i"] & (\underset{\boxed{[]}}{\overline{\overline{\alpha}\; sh}}, \underset{\boxed{[1]}}{\overline{\alpha}\; raw_{\sigma(1)}}, \ldots, \underset{\boxed{[\mathbf{j}]}}{\overline{\alpha}\; raw_{\sigma(\mathbf{j})}})\; G_i \arrow[l, shift left, "\mathsf{Raw}_i"] \arrow[u, shift left, "\mathsf{map}_{G_i}\, \mathsf{unLeaf}_1 \ldots \mathsf{unLeaf}_{\mathbf{k}}\, (\downarrow_{1\sigma(1)}\, []) \ldots (\downarrow_{\mathbf{j}\sigma(\mathbf{j})}\, [])"]
\end{tikzcd}
$$

Fig. 1. Definitions of constructors and destructors

and $\overline{\overline{\alpha}\, F_1}$ stands for $\overline{\alpha}\, F_{11}, \ldots, \overline{\alpha}\, F_{1\mathbf{k}}$. It should be clear from the context which index is omitted.

A *definition* of $\mathbf{i}$ mutual nonuniform datatypes T_i is a system of $\mathbf{i}$ type fixpoint equations

$$\overline{\alpha}\; T_i \simeq \big(\overline{\alpha}, (\overline{\overline{\alpha}\, F_1})\; T_{\sigma(1)}, \ldots, (\overline{\overline{\alpha}\, F_{\mathbf{j}}})\; T_{\sigma(\mathbf{j})}\big)\; G_i \tag{1}$$

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

We start by defining the *shape* types $\overline{\alpha}\; sh_k$ that overapproximate the recursive changes to the type arguments. There are $\mathbf{k}$ shape types, corresponding to the $\mathbf{k}$ type arguments, and they are mutually recursive uniform datatypes:

$$\overline{\alpha}\; sh_k = \mathsf{Leaf}_k\; \alpha_k \mid \mathsf{Node}_{1k}\; (\overline{\overline{\alpha}\; sh}\; F_{1k}) \mid \cdots \mid \mathsf{Node}_{\mathbf{j}k}\; (\overline{\overline{\alpha}\; sh}\; F_{\mathbf{j}k})$$

For *plist*, the *sh* type is *sh*. 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 $\mathbf{i}$ uniform mutually recursive datatypes raw_i that recurse 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 $\overline{\overline{\alpha}\; sh}$:

$$\overline{\alpha}\; raw_i = \mathsf{Raw}_i\; ((\overline{\overline{\alpha}\; sh}, \overline{\alpha}\; raw_{\sigma(1)}, \ldots, \overline{\alpha}\; raw_{\sigma(\mathbf{j})})\; G_i)$$

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 $\boxed{\mathsf{ok}}_k : [\mathbf{j}]\; list \to \overline{\alpha}\; sh_k \to bool$ on the shape types. The type of $\boxed{\mathsf{ok}}_k$ shows an important difference to the *plist* 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 Δ stand for $[\mathbf{j}]\; list$. The additional structure is necessary because different Node_{jk} 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 $\mathsf{Cons}_1\; (u : \alpha)\; (\mathsf{Cons}_2\; (v : \alpha)$ $(\mathsf{Cons}_2\; (w : \alpha \times \alpha)\; (\mathsf{Cons}_1\; (x : (\alpha \times \alpha) \times (\alpha \times \alpha))\; \mathsf{Nil}))) : \alpha\; plist'$. This order of constructors forces x's type to be $\alpha\; F = \alpha\; F_1\; F_2\; F_2$, with $\alpha\; F_1 = \alpha$ and $\alpha\; F_2 = \alpha \times \alpha$. Consequently, x is embedded into $\alpha\; sh$ as $\mathsf{Node}_2\; (\mathsf{map}_{F_2}\; \mathsf{Node}_2\; (\mathsf{map}_{F_2}$ $(\mathsf{map}_{F_2}\; \mathsf{Node}_1)\; (\mathsf{map}_F\; \mathsf{Leaf}\; x)))$, whose shadow is $[2, 2, 1]$.

Formally, the predicates $\boxed{\mathsf{ok}}_k$ are defined together as the least predicates satisfying the rules

$$\boxed{\mathsf{ok}}_k\; []\; (\mathsf{Leaf}_k\; x)$$
$$\mathsf{pred}_{F_{jk}}\; (\boxed{\mathsf{ok}}_1\; u) \ldots (\boxed{\mathsf{ok}}_{\mathbf{k}}\; u)\; f \Longrightarrow \boxed{\mathsf{ok}}_k\; (j \triangleleft u)\; (\mathsf{Node}_{jk}\; f)$$

where $[]$ and $\triangleleft$ are notations for Nil and Cons. To access the recursive components of *sh*, we rely on the predicators associated with the F's. Predicators are monotone. The $\mathbf{i}$ mutual predicates $\mathsf{ok}_i : \Delta \to \overline{\alpha}\; raw_i \to bool$ are defined similarly:

$$\mathsf{pred}_{G_i}\; (\boxed{\mathsf{ok}}_1\; u) \ldots (\boxed{\mathsf{ok}}_{\mathbf{k}}\; u)\; (\mathsf{ok}_{\sigma(1)}\; (1 \triangleleft u)) \ldots (\mathsf{ok}_{\sigma(\mathbf{j})}\; (\mathbf{j} \triangleleft u))\; g \Longrightarrow \mathsf{ok}_i\; u\; (\mathsf{Raw}_i\; g)$$

We access the $\mathbf{k}$ immediate components of the shape type and the $\mathbf{j}$ recursive components of the raw type through the predicator. We write that an element r of type $\overline{\alpha}\; raw_i$ has shadow u if $\mathsf{ok}_i\; u\; r$ holds.

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

We can prove by induction that ok_i is invariant under the map_{raw_i} function.

Lemma 1: $\mathsf{ok}_i\; u\; (\mathsf{map}_{raw_i}\; \overline{f}\; r) \Leftrightarrow \mathsf{ok}_i\; u\; r$.

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 $\mathsf{Ctor}_i : (\overline{\alpha}, (\overline{\overline{\alpha}\, F_1})\; T_{\sigma(1)}, \ldots,$ $(\overline{\overline{\alpha}\, F_{\mathbf{j}}})\; T_{\sigma(\mathbf{j})})\; G_i \to \overline{\alpha}\; T_i$ and a destructor $\mathsf{dtor}_i : \overline{\alpha}\; T_i \to$ $(\overline{\alpha}, (\overline{\overline{\alpha}\, F_1})\; T_{\sigma(1)}, \ldots, (\overline{\overline{\alpha}\, F_{\mathbf{j}}})\; T_{\sigma(\mathbf{j})})\; G_i$ and by showing that they are inverses of each other.

Figure 1 gives diagrammatic definitions of Ctor_i (by composing the functions on the outer arrows) and dtor_i (by composing the functions on the inner arrows). All shape and raw types occurring in the diagram are annotated with their shadows. Abs_i can be applied only to raw elements with shadow $[]$.

The unLeaf_k and unRaw_i functions are inverses of the corresponding constructors; they satisfy $\mathsf{unLeaf}_k\; (\mathsf{Leaf}_k\; a) = a$

and $\mathsf{unRaw}_i\ (\mathsf{Raw}_i\ r) = r$. Note that unLeaf_k is underspecified and (like Abs_i) may be applied only to Leaf_k shape elements with shadow $[]$. Moreover, the definition must bridge the gap between the types $\overline{\overline{\alpha}\,F_j}\ raw_{\sigma(j)}$ of shadow $[]$ and $\overline{\alpha}\ raw_{\sigma(j)}$ of shadow $[j]$ (the rightmost arrows in Figure 1). This must happen recursively (even though the constructor Ctor_i 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 $u \triangleright j$ (i.e., the list u with the element j appended to it). For each fixed j, inlining is implemented by means of $\mathbf{i}$ mutual primitively recursive functions $\uparrow_{ji} : (\overline{\overline{\alpha}\,F_j})\ raw_i \to \overline{\alpha}\ raw_i$, whose definition uses $\mathbf{k}$ mutual primitively recursive functions $\boxed{\Uparrow}_{jk} : (\overline{\overline{\alpha}\,F_j})\ sh_k \to \overline{\alpha}\ sh_k$ on the shape type:

$$\boxed{\Uparrow}_{jk}\ (\mathsf{Leaf}_k\ f) = \mathsf{Node}_{jk}\ (\mathsf{map}_{F_{jk}}\ \overline{\mathsf{Leaf}}\ f)$$
$$\boxed{\Uparrow}_{jk}\ (\mathsf{Node}_{j'k}\ f) = \mathsf{Node}_{j'k}\ (\mathsf{map}_{F_{j'k}}\ \overline{\boxed{\Uparrow}_j}\ f)$$
$$\uparrow_{ji} u\ (\mathsf{Raw}_i\ g) = \mathsf{Raw}_i\ (\mathsf{map}_{G_i}\ \overline{\boxed{\Uparrow}_j}\ \uparrow_{j\sigma(1)}\ \ldots\ \uparrow_{j\sigma(\mathbf{j})}\ g)$$

Inlining is injective. We define the (partial) inverse operations $\downarrow_{ji} : \Delta \to \overline{\alpha}\ raw_i \to (\overline{\overline{\alpha}\,F_j})\ raw_i$ and $\boxed{\Downarrow}_{jk} : \Delta \to \overline{\alpha}\ sh_k \to (\overline{\overline{\alpha}\,F_j})\ sh_k$, which are useful when defining the destructors dtor_i. The additional shadow parameter in $\boxed{\Downarrow}_{ji}$ denotes how many more layers to destruct until we arrive at the last layer of Nodes (with only Leaf constructors below).

$$\boxed{\Downarrow}_{jk}\ []\ (\mathsf{Node}_{jk}\ f) = \mathsf{Leaf}_k\ (\mathsf{map}_{F_{jk}}\ \overline{\mathsf{unLeaf}}\ f)$$
$$\boxed{\Downarrow}_{jk}\ (j' \triangleleft u)\ (\mathsf{Node}_{j'k}\ f) = \mathsf{Node}_{j'k}\ (\mathsf{map}_{F_{j'k}}\ \overline{(\boxed{\Downarrow}_j\ u)}\ f)$$
$$\downarrow_{ji} u\ (\mathsf{Raw}_i\ g) =$$
$$\mathsf{Raw}_i\ (\mathsf{map}_{G_i}\ \overline{(\boxed{\Downarrow}_j\ u)}(\downarrow_{j\sigma(1)}\ (1 \triangleleft u))\ \ldots\ (\downarrow_{j\sigma(\mathbf{j})}\ (\mathbf{j} \triangleleft u))\ g)$$

We establish the expected behavior of $\uparrow_{ji}$ and $\downarrow_{ji}$ 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 $\boxed{\Uparrow}_{jk}$ and $\boxed{\Downarrow}_{jk}$.

Lemma 2:
1) $\mathsf{ok}_i\, u\, r \Longrightarrow \mathsf{ok}_i\,(u \triangleright j)\,(\uparrow_{ji} r)$; 3) $\mathsf{ok}_i\, u\, r \Longrightarrow \downarrow_{ji} u\,(\uparrow_{ji} r) = r$;
2) $\mathsf{ok}_i\,(u \triangleright j)\, r \Longrightarrow \mathsf{ok}_i\, u\,(\downarrow_{ji} u\, r)$; 4) $\mathsf{ok}_i\,(u \triangleright j)\, r \Longrightarrow \uparrow_{ji}(\downarrow_{ji} u\, r) = r$.

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_i and dtor_i.

Theorem 3: $\mathsf{dtor}_i\ (\mathsf{Ctor}_i\ g) = g$ and $\mathsf{Ctor}_i\ (\mathsf{dtor}_i\ t) = t$.

Finally, we prove characteristic theorems for T_i's BNF constants. We focus on the property that map_{T_i} commutes with the constructor Ctor_i. The theorems for the relator, the predicator, and the set functions are proved analogously.

Theorem 4: $\mathsf{map}_{T_i}\ \overline{f}\ (\mathsf{Ctor}_i\ g) = \mathsf{Ctor}_i\ (\mathsf{map}_{R_i}\ \overline{f}\ g)$ where $\overline{\alpha}\ R_i = (\overline{\alpha}, (\overline{\overline{\alpha}\,F_1})\ T_{\sigma(1)}, \ldots, (\overline{\overline{\alpha}\,F_{\mathbf{j}}})\ T_{\sigma(\mathbf{j})})\ G_i$ and map_{R_i} is the map function associated to this composite BNF.

The proof of Theorem 4 relies on commutation properties of map_{raw_i} and $\uparrow_{ji}$ and of map_{sh_k} and $\boxed{\Uparrow}_{jk}$ that can be proved by induction. This is a pervasive pattern when defining recursive functions on nonuniform datatypes.

Lemma 5: $\mathsf{map}_{sh_k}\ \overline{f}\ (\boxed{\Uparrow}_{jk}\ s) = \boxed{\Uparrow}_{jk}\ (\mathsf{map}_{sh_k}\ \overline{(\mathsf{map}_{F_j}\ \overline{f})}\ s)$ and $\mathsf{map}_{raw_i}\ \overline{f}\ (\uparrow_{ji} r) = \uparrow_{ji}\ (\mathsf{map}_{raw_i}\ \overline{(\mathsf{map}_{F_j}\ \overline{f})}\ r)$.

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 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 ok_i are defined as a mutual greatest (or coinductive) fixpoint of the same introduction rule as before for datatypes. The functions $\uparrow_{ji}$ and $\downarrow_{ji}$ 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 ok_i. Similarly, the equational statements (e.g., Lemma 2(3–4) or the raw part of Lemma 5) are proved by coinduction on $=$.

IV. The Nonemptiness 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 [13]. It is based on storing, for each BNF $\overline{\alpha}\ K$ with $\overline{\alpha} = (\alpha_1, \ldots, \alpha_n)$, complete information on its *conditional nonemptiness*, i.e., on which combinations of nonemptiness assumptions for the argument types α_i would be sufficient to guarantee nonemptiness of $\overline{\alpha}\ K$. For example, if $n = 3$ and $\overline{\alpha}\ K$ is $\alpha_1\ stream + \alpha_2 \times \alpha_3$, then for $\overline{\alpha}\ K$ to be nonempty it suffices that either α_1, or both α_2 and α_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 $\alpha_1\ stream + \alpha_2 \times \alpha_3$.

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 set_K operators to capture the action of K on *sets*, as the homonymous constant $K : \alpha_1\ set \to \cdots \to \alpha_n\ set \to (\overline{\alpha}\ K)\,set$, defined by $K\,A_1\ \ldots\ A_n = \{x : \overline{\alpha}\ K \mid \forall i \in [n].\ \mathsf{set}^i_K\ x \subseteq A_i\}$. 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 $\overline{A}$, $\forall i \in I.\ A_i \neq \varnothing$ implies $K\,\overline{A} \neq \varnothing$. A set $\mathscr{I} \subseteq [n]\,set$ of witnesses for K is called *perfect* if for all witnesses $J \subseteq [n]$ there exists a witness $I \in \mathscr{I}$ 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 $I \in \mathscr{I}$.

We fix a definition of $\mathbf{i}$ 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 $\mathscr{I}(G_i)$ and $\mathscr{I}(F_{jk})$. 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 α_i as terminals. The productions of the set-grammar follow the direction of the destructors, $\overline{\alpha}\, T_i \xrightarrow{\mathsf{dtor}_i} (\overline{\alpha}, \overline{\alpha} T_1, \ldots, \overline{\alpha} T_{\mathbf{i}})\, G_i$, with each T_i deriving sets containing the nonterminals $T_{i'}$ and the terminals α_k allowed by witnesses of G_i.

For the nonuniform case, when recursively applying productions $\overline{\alpha}\ T_i \xrightarrow{\mathsf{dtor}_i} (\overline{\alpha}, (\overline{\overline{\alpha}\, F_1})\ T_{\sigma(1)}, \ldots, (\overline{\overline{\alpha}\, F_{\mathbf{j}}})\ T_{\sigma(\mathbf{j})})\ G_i$ 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 $\mathscr{I}(F_{jk})$. We define the set $\mathsf{PolyWit}$, of *polynomial witness sets* (or *polywits*), inductively as follows:

- If $k \in [\mathbf{k}]$, then $\{\{k\}\} \in \mathsf{PolyWit}$.
- If $(j,k) \in [\mathbf{j}] \times [\mathbf{k}]$ and $p_1, \ldots, p_{\mathbf{k}} \in \mathsf{PolyWit}$, then $(p_1, \ldots, p_{\mathbf{k}}) \cdot \mathscr{I}(F_{jk}) \in \mathsf{PolyWit}$.

Polywits are sets of subsets of $[\mathbf{k}]$. In the second clause above, we use the composition $(p_1, \ldots, p_{\mathbf{k}}) \cdot \mathscr{I}(F_{jk})$, which is defined as $\bigcup_{I \in \mathscr{I}(F_{jk})} \{\bigcup_{k' \in I} J_{k'} \mid \overline{J} \in \prod_{k' \in I} p_{k'}\}$. This composition captures the computation of witnesses for the composition of F_{jk} with the BNFs corresponding to the polywits $p_1, \ldots, p_{\mathbf{k}}$.

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

1) $p \longrightarrow I$, where $p \in \mathsf{PolyWit}$ and $I \in p$;
2) $\overline{p}\, t_i \longrightarrow \Gamma_J$ where $i \in [\mathbf{i}]$, $J \in \mathscr{I}(G_i)$ and $\Gamma_J = \{p_k \mid k \in J \cap [\mathbf{k}]\} \cup \{(\overline{p} \cdot \mathscr{I}(F_{j1}), \ldots, \overline{p} \cdot \mathscr{I}(F_{j\mathbf{k}}))\, t_{\sigma(j)} \mid \mathbf{k} + j \in J\}$.

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

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 $\mathsf{Lang}_i(\mathsf{Gr}) \neq \varnothing$ and $\mathsf{Lang}_i(\mathsf{Gr})$ 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 $\mathsf{Lang}_{\infty,i}(\mathsf{Gr})$ 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 ok_i []). For codatatypes, nonemptiness always holds owing to the greatest fixpoint nature of the construction.

Since the *raw*$_i$'s are BNFs, we have a perfect set of witnesses for them, but these will usually fail to satisfy ok_i [] (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 Gr offer perfect sets for $\mathsf{ok}_i\, u$ for any shadow u. We generalize from [] to an arbitrary u because the shadow increases along applications of the *raw*$_i$ destructors. Appendix B provides the details.

For any finite set-grammar Gr, the languages $\mathsf{Lang}_i(\mathsf{Gr})$ and $\mathsf{Lang}_{\infty,i}(\mathsf{Gr})$ are effectively computable by fixpoint iteration [13]. Moreover, iteration needs at most a number of steps equal to the number of nonterminals [13, Section 4.3]. For uniform datatypes, this number is precisely that of mutual types, $\mathbf{i}$; for nonuniform datatypes, it is the larger number $\mathbf{j} \times \mathbf{k} \times |\mathsf{PolyWit}|$, where $|\mathsf{PolyWit}| = O(2^{2^{\mathbf{k}}})$. Fortunately, the worst-case double exponential in $\mathbf{k}$ 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 $\mathbf{k}$.

Consider the following contrived nonuniform codatatype of alternating streams over α_1 and α_2:

$$(\alpha_1, \alpha_2)\ alt \overset{\infty}{=} \mathsf{C}\ \alpha_1\ ((\alpha_2, \alpha_1)\ alt) \mid \mathsf{D}\ \alpha_2\ ((\alpha_2, \alpha_1)\ alt)$$

We have $\mathbf{i} = 1$, $\mathbf{j} = 1$, $\mathbf{k} = 2$, σ 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 $(\alpha_1, \alpha_2, \alpha_3)\ G_1 = \alpha_1 \times \alpha_3 + \alpha_2 \times \alpha_3$. 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 $(①, ②)\, t_1$ in the grammar 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 $(①, ②)\, t_1$. In the left tree, the top production is $(①, ②)\, t_1 \longrightarrow \{①, (②, ①)\, t_1\}$; this is a valid production of type 2, based on G_1's witness $\{1,3\}$. The tree's other production of type 2, $(②, ①)\, t_1 \longrightarrow \{①, (①, ②)\, t_1\}$, 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 *alt*.

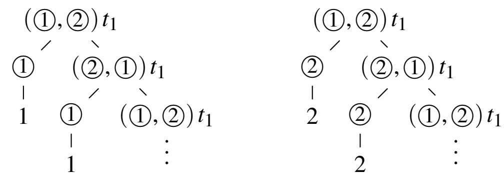

Fig. 2. Derivation trees in the witness set-grammar

$$\begin{tikzcd}
\alpha\ \overset{r}{raw} \arrow[rr, "\mathsf{Abs}"] \arrow[d, "\mathsf{dtor}"'] & & \alpha\ T \arrow[dd, "\mathsf{unT}"] \\
(\alpha\ sh, \alpha\ \overset{r'}{raw})\ G \arrow[d, "\mathsf{unLeaf}"', "\downarrow []"] & & \\
(\alpha, \alpha\ F\ \overset{r''}{raw})\ G & & (\alpha, \alpha\ F\ T)\ G \arrow[ll, "\mathsf{id}"', "\mathsf{Rep}"]
\end{tikzcd}$$

Fig. 3. The *raw* representation of T

$$\begin{tikzcd}
\alpha\ sh\ \overset{r}{raw} \arrow[rr, "\mathsf{Abs}"] \arrow[d, "\mathsf{dtor}"'] & & \alpha\ sh\ T \arrow[dd, "\mathsf{unT}"] \\
(\alpha\ sh\ sh, \alpha\ sh\ \overset{r'}{raw})G \arrow[d, "\mathsf{unLeaf}"', "\downarrow []"] & & \\
(\alpha\ sh, \alpha\ sh\ F\ \overset{r''}{raw})G \arrow[u, "\mathsf{map}_{raw}\,\mathsf{Node}"', "\mathsf{Leaf}", shift left=4] & & (\alpha\ sh, \alpha\ sh\ F\ T)G \arrow[ll, "\mathsf{id}"', "\mathsf{Rep}"]
\end{tikzcd}$$

Fig. 4. Connecting the T and the *raw* components

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 $\alpha\,\mathit{fractal} = \mathsf{Fractal}\ ((\alpha, ((\alpha, \alpha)\ \mathit{alt})\,\mathit{fractal})\ \mathit{alt})$, 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 $\alpha\ T \simeq (\alpha, \alpha\ F\ T)\ G$ as in Section III-B but with $\mathbf{i} = \mathbf{j} = \mathbf{k} = 1$. 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 $\alpha\ raw$, its components are the elements r' of $\mathsf{set}^2_G\ (\mathsf{unRaw}\ r)$—because, in its fixpoint definition, *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 $\alpha\ raw$ is the following HOL theorem, which we will refer to as Ind_{raw}:

$$\forall Q.\ (\forall r : \alpha\ raw.\ (\forall r' \in \mathsf{set}^2_G\ (\mathsf{unRaw}\ r).\ Q\ r') \Longrightarrow Q\ r) \Longrightarrow \\ \forall r : \alpha\ raw.\ Q\ r$$

It states that to show that a predicate Q holds for all $\alpha\ raw$, it suffices to show that Q holds for any element r given that Q holds for the recursive components $\mathsf{set}^2_G\ (\mathsf{unRaw}\ r)$ of r.

As we remarked in Section I, a verbatim translation of Ind_{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 $\mathsf{Q} : \alpha\ raw \to bool$ (and remove the outer $\forall$), 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 $nat\ F\ T$, we could use such an induction theorem (with α instantiated to nat) to wrongly infer that Q is true on $nat\ T$.

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

$$\frac{\forall \alpha.\ \forall t : \alpha\ T.\ (\forall t' \in \mathsf{set}^2_G\ (\mathsf{dtor}\ t).\ \mathsf{P}\ t') \Longrightarrow \mathsf{P}\ t}{\forall \alpha.\ \forall t : \alpha\,T.\ \mathsf{P}\ t}\ \mathsf{Ind}^{\mathsf{P}}_T$$

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

We start by defining Q to be $\mathsf{P} \circ \mathsf{Abs}$ and try to prove $\forall r.\ \mathsf{ok}\ []\ r \Longrightarrow \mathsf{Q}\ r$ using Ind_{raw}, hoping to be able to connect the hypothesis of $\mathsf{Ind}^{\mathsf{P}}_T$ with that of Ind_{raw}. We quickly encounter the following problem, depicted in Figure 3.[1] Suppose $r : \alpha\ raw$ 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 *raw* components $r' : \alpha\ raw$ of r, but rather to elements $r'' : \alpha\ F\ raw$ of the form $\downarrow []\ r'$. This mismatch is a consequence of our representation technique: To represent T's destructor using *raw*'s destructor, we applied the "correction" $\downarrow []$. To cope with it, we appeal to the shape type $\alpha\ sh$, 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 $\alpha\ sh\ T$. For Q, this means switching from $\alpha\ raw$ to $\alpha\ sh\ raw$. As shown in Figure 4, now we can travel from the type $\alpha\ sh\ F\ raw$ back to the type $\alpha\ sh\ raw$, by applying Node to level the nonuniformity F into the larger type sh. For this to work, Q must reflect $\mathsf{map}_{raw}\ \mathsf{Node}$, i.e., have $\mathsf{Q}\,(\mathsf{map}_{raw}\ \mathsf{Node}\ r'')$ imply $\mathsf{Q}\ r''$.

Another issue is that $r''' = \mathsf{map}_{raw}\ \mathsf{Node}\ r''$ is not in the image of Rep: r''' has shadow $[1]$ instead of the required $[]$. We must generalize the goal to arbitrary shadows, i.e., to

[1]Starting with this figure, we replace $\mathsf{map}_G\ f\ g$ arrow annotations with arrows carrying two labels: f to the right in the arrow's direction of travel and g to the left.

$$\begin{tikzcd}
\alpha\; sh\; raw \arrow[r, "\updownarrows\!\!\!\uparrow u"] \arrow[dd, "\mathsf{unRaw}"'] & \alpha\; sh\; raw \arrow[r, "\mathsf{Abs}"] \arrow[d, "\mathsf{unRaw}"'] & \alpha\; sh\; T \arrow[dd, "\mathsf{dtor}"] \\
 & (\alpha\; sh\; sh, \alpha\; sh\; raw)\; G \arrow[d, "\mathsf{id}"', "\updownarrows\,{[]}"] & \\
(\alpha\; sh\; sh, \alpha\; sh\; raw)\; G \arrow[r, "\updownarrows\!\!\!\uparrow (1 \triangleleft u)", "\boxed{\updownarrows\!\!\!\uparrow}\, u"'] & (\alpha\; sh\; sh, \alpha\; sh\; raw)\; G & (\alpha\; sh, \alpha\; sh\; F\; T)\; G \arrow[l, "\mathsf{Leaf}"', "\mathsf{map}_{raw}\; \mathsf{Node} \circ \mathsf{Rep}"]
\end{tikzcd}$$

Fig. 5. Borrowing induction and coinduction from *raw* to T

$\forall r\; u.\; \mathsf{ok}\; u\; r \Longrightarrow \mathsf{Q}'\; u\; r$, for a suitable predicate $\mathsf{Q}' : \Delta \to \alpha\; sh\; raw \to bool$ that extends Q in that $\mathsf{Q}'\; [] = \mathsf{Q}$. To this end, we define $\updownarrows : \Delta \to \alpha\; sh\; raw \to \alpha\; sh\; raw$, an operator that generalizes the trip from r' to r'' to r''' described above to an arbitrary shadow u, and $\updownarrows\!\!\!\uparrow$, the cumulative iteration of $\updownarrows$:

$$\updownarrows\; u\; r = \mathsf{map}_{raw}\; \mathsf{Node}\; (\downarrow u\; r) \qquad \begin{aligned} \updownarrows\!\!\!\uparrow\; []\; r &= r \\ \updownarrows\!\!\!\uparrow\; (u \triangleright 1)\; r &= \updownarrows\; u\; (\updownarrows\!\!\!\uparrow\; u\; r) \end{aligned}$$

Intuitively, we can regard the elements of both $\beta\; sh$ and $\beta\; raw$ as trees with β leaves and whose nodes branch according to F. Then $\updownarrows$ traverses elements of $\alpha\; sh\; raw$ until it reaches their innermost nodes (with only leaves, i.e., elements of $\alpha\; sh$, 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 $\updownarrows\; u\; r$ only if $\mathsf{ok}\; u\; r$).

The *sh* counterparts of the above, $\boxed{\updownarrows} : \Delta \to \alpha\; sh\; sh \to \alpha\; sh\; sh$ and $\boxed{\updownarrows\!\!\!\uparrow}$, 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 $\mathsf{Q}'\; u\; r$ to be $\mathsf{Q}\; (\updownarrows\!\!\!\uparrow\; u\; r)$. 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 $\mathsf{Ind}_T^{\mathsf{P}}$, we have proved $\forall \alpha.\, \forall r : \alpha\; sh\; raw.\, \forall u : \Delta.\; \mathsf{ok}\; u\; r \Longrightarrow \mathsf{Q}'\; u\; r$—in particular, $\forall \alpha.\, \forall r : \alpha\; sh\; raw.\; \mathsf{ok}\; []\; r \Longrightarrow \mathsf{Q}\; r$, which implies $\forall \alpha.\, \forall t : \alpha\; sh\; T.\; \mathsf{P}\; t$. From this, we prove the more general fact $\forall \alpha.\, \forall t : \alpha\; T.\; \mathsf{P}\; t$. It would suffice that P reflects $\mathsf{map}_T\; \mathsf{Leaf} : \alpha\; T \to \alpha\; sh\; T$. We can simply asume that P is *injective-antitone-parametric* (*IAP*), meaning that $\mathsf{P}\; (\mathsf{map}_T\; f\; t)$ implies $\mathsf{P}\; t$ for all $t : \alpha\; T$ and all injective functions $f : \alpha \to \beta$ (including Leaf and Node). In conclusion:

Theorem 8: If P is IAP, then $\mathsf{Ind}_T^{\mathsf{P}}$ is derivable in HOL.

IAP is substantially weaker than (general) parametricity, which for P would mean $\mathsf{P}\; t \Leftrightarrow \mathsf{P}\; (\mathsf{map}_T\; f\; t)$ for all t and arbitrary functions f (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 $\alpha\; T$ 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):

$$\frac{\begin{array}{l}\forall \alpha.\, \forall t_1, t_2 : \alpha\, T. \\ \quad \mathsf{P}\; t_1\; t_2 \Longrightarrow \mathsf{rel}_G\; (=)\; \mathsf{P}\; (\mathsf{dtor}\; t_1)\; (\mathsf{dtor}\; t_2)\end{array}}{\forall \alpha.\, \forall t : \alpha\, T.\; \mathsf{P}\; t_1\; t_2 \Longrightarrow t_1 = t_2}\; \mathsf{Coind}_T^{\mathsf{P}}$$

For this rule to be sound, $\mathsf{P} : \alpha\; T \to \alpha\; T \to bool$ must again interact well with injective functions, but this time in the opposite direction. We say that P is *injective-monotone-parametric* (*IMP*) if $\mathsf{P}\; t_1\; t_2$ implies $\mathsf{P}\; (\mathsf{map}_T\; f\; t_1)\; (\mathsf{map}_T\; g\; t_2)$ for all $t_1, t_2 : \alpha\; T$ and injective functions $f, g : \alpha \to \beta$.

Theorem 9: If P is IMP, then $\mathsf{Coind}_T^{\mathsf{P}}$ 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:

$$\begin{aligned}
&\mathsf{split} : (\alpha \times \beta)\; plist \to \alpha\; plist\; \times \beta\; plist \\
&\mathsf{split}\; \mathsf{Nil} = (\mathsf{Nil}, \mathsf{Nil}) \\
&\mathsf{split}\; (\mathsf{Cons}\; (a, b)\; xs) = \mathsf{let}\; (as, bs) = \mathsf{split}\; (\mathsf{map}_{plist}\; \mathsf{swap}\; xs) \\
&\qquad\qquad\qquad\qquad\quad\;\; \mathsf{in}\; (\mathsf{Cons}\; a\; as, \mathsf{Cons}\; b\; bs)
\end{aligned}$$

Here, the pattern-matched variable xs has type $((\alpha \times \beta) \times (\alpha \times \beta))\; plist$, and the auxiliary swap function is defined as $\mathsf{swap}\; ((a_1, b_1), (a_2, b_2)) = ((a_1, a_2), (b_1, b_2))$. 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 $((\alpha \times \alpha) \times (\beta \times \beta))\; plist$. 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 $\mathsf{map}_{plist}\ \mathsf{swap}\ xs$. However, the modification takes place through the map_{plist} function, which leaves the length of xs unchanged. Hence, such *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 *primitively iterative specifications* can be reduced as well, using a simpler fold combinator. For a uniform datatype $\alpha\ T = \mathsf{Ctor}\ ((\alpha, \alpha\ T)\ G)$ (e.g., $\alpha\ T = \alpha\ list$ with $(\alpha,\beta)\ G = unit + \alpha \times \beta$), the fold combinator has type $((\alpha,\beta)\ G \to \beta) \to \alpha\ T \to \beta$.

A function $\mathsf{f} = \mathsf{fold}\ \mathsf{b}$ for some fixed $\mathsf{b} : (\alpha,\beta)\ G \to \beta$, satisfies the characteristic recursive equation $\mathsf{f}\ (\mathsf{Ctor}\ g) = \mathsf{b}\ (\mathsf{map}_G\ \mathsf{id}\ \mathsf{f}\ g)$. We call b the *blueprint* of f. Note that b describes how to combine the *results* of the recursive calls into a new result of type β. The recursion combinator's blueprint, of type $(\alpha, \alpha\ T \times \beta)\ G \to \beta$, 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

$$\forall Y.\, (\forall \alpha.\, (\alpha, \alpha\ F\ Y)\ G \to \alpha\ Y) \to \beta\ T \to \beta\ Y$$

where the universally quantified type constructor Y captures the positions where α 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 α indicates that the blueprint needs to be truly polymorphic in α.

The primitive iteration schema provided by the above combinator is very restrictive, because it forces the type argument β of T to be fully polymorphic. Neither the split function nor a simple summation of a powerlist can be expressed using that fold. To overcome this limitation, Bird and Paterson [9] propose a *generalized* fold of type

$$\forall X\ Y.\, (\forall \alpha.\, (\alpha\ X, \alpha\ F\ Y)\ G \to \alpha\ Y) \to (\forall \alpha.\, \alpha\ X\ F \to \alpha\ F\ X) \to \beta\ X\ T \to \beta\ Y$$

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

$$\mathsf{f}\ (\mathsf{Ctor}\ g) = \mathsf{b}\ (\mathsf{map}_G\ \mathsf{id}\ (\mathsf{f} \circ \mathsf{map}_T\ \mathsf{a})\ g) \qquad (2)$$

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

$$\begin{aligned} \mathsf{b}\ (\mathsf{Inl}\ ()) &= (\mathsf{Nil}, \mathsf{Nil}) \\ \mathsf{b}\ (\mathsf{Inr}\ ((a, b), ys)) &= \mathsf{let}\ (as, bs) = ys \\ &\ \ \mathsf{in}\ (\mathsf{Cons}\ a\ as, \mathsf{Cons}\ b\ bs) \end{aligned}$$

where Inl and Inr 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):

$$\forall X\ Y.\, (\forall \alpha.\, (\alpha\ X, \alpha\ V\ Y)\ G \to \alpha\ Y) \to (\forall \alpha.\, \alpha\ X\ F \to \alpha\ V\ X) \to \beta\ X\ T \to \beta\ Y$$

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 *recursion procedure* that takes (here, unary) BNFs V, X, and Y,[2] a blueprint $\mathsf{b} : (\alpha\ X, \alpha\ V\ Y)\ G \to \alpha\ Y$ and a swapper $\mathsf{a} : \alpha\ X\ F \to \alpha\ V\ X$ as input and yields a function $\mathsf{f} : \alpha\ X\ T \to \alpha\ Y$ satisfying equation (2).

The procedure defines a recursive function using b and a on the *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., $\mathsf{map}_F\ (\mathsf{map}_X\ f) = \mathsf{map}_X\ (\mathsf{map}_V\ f) \circ \mathsf{a}$ 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 sh_V for V analogously to sh for F, including the constructors Leaf_V, Node_V, their inverses unLeaf_V, unNode_V, and the functions $\boxed{\mathsf{ok}}_V$, $\boxed{\Uparrow}_V$, and $\boxed{\Downarrow}_V$. Second, we lift a to shapes $\boxed{\mathsf{a}} : \Delta \to \alpha\ X\ sh \to \alpha\ sh_V\ X$ by recursion on the shadow:

$$\begin{aligned} \boxed{\mathsf{a}}\ [] &= \mathsf{map}_X\ \mathsf{Leaf}_V \circ \mathsf{unLeaf} \\ \boxed{\mathsf{a}}\ (1 \triangleleft u) &= \mathsf{map}_X\ \mathsf{Node}_V \circ \mathsf{a} \circ \mathsf{map}_F\ (\boxed{\mathsf{a}}\ u) \circ \mathsf{unNode} \end{aligned}$$

Third, we define a *raw* version of our function $\mathsf{f}_{raw} : \Delta \to \alpha\ X\ T \to \alpha\ sh_V\ Y$ by primitive recursion:

$$\mathsf{f}_{raw}\ u\ (\mathsf{Raw}\ g) = \mathsf{b}\ (\mathsf{map}_G\ (\boxed{\mathsf{a}}\ u)\ (\mathsf{map}_Y\ \mathsf{unNode}_V \circ \mathsf{f}_{raw}\ (1 \triangleleft u))\ g)$$

The generalization to sh_V in the return type of f_{raw} is similar to what we did for induction. Finally, we define the function f as $\mathsf{f} = \mathsf{map}_Y\ \mathsf{unLeaf}_V \circ \mathsf{f}_{raw}\ [] \circ \mathsf{Rep}$.

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

[2] Strictly speaking, the boundedness assumption is not needed for X and Y, which implies that $\alpha\ set$ is permitted to occur in those type expressions.

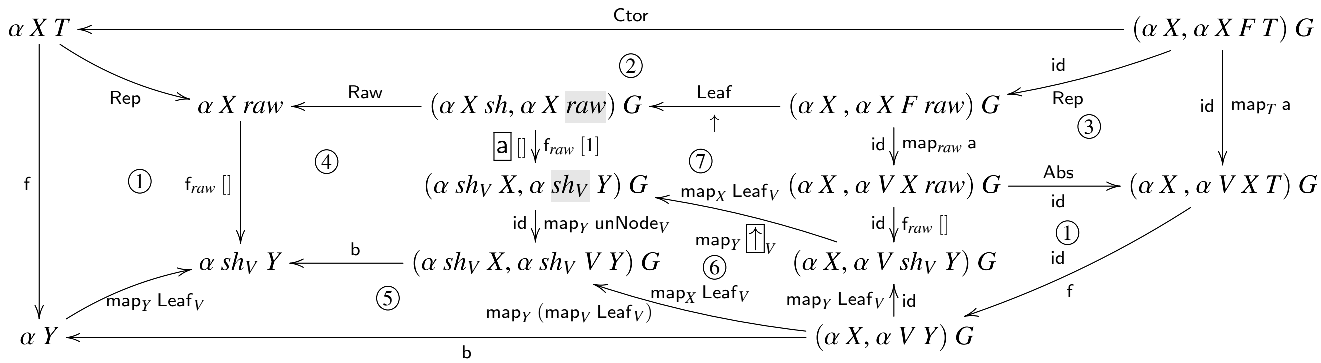

Fig. 6. Proof of the recursive specification of the function f

such as $\mathsf{Leaf}_{(V)}$ and $\mathsf{Node}_{(V)}$ (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_V, and *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 ① (twice), Ctor ②, and map_T ③ as well as the recursive specification of f_{raw} ④. The quadrilateral ⑤ follows from the naturality for injective functions (Leaf_V) of b and ⑥ from the recursive specification of $\boxed{\Uparrow}_V$. The remaining commutative pentagon ⑦ relates f_{raw} and $\Uparrow$ (similarly to Lemma 5 for map_{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 $\boxed{\Uparrow}$ together with the facts that $\boxed{\mathsf{a}}$ and f_{raw} preserve $\boxed{\mathsf{ok}}\, u$ and $\mathsf{ok}\, u$. The proofs rely on the injective-parametricity of a and b.

Lemma 10:

1) $\boxed{\mathsf{ok}}\, u\, s \Longrightarrow \mathsf{pred}_{sh_V}\ (\boxed{\mathsf{ok}}_V\ u)\ (\boxed{\mathsf{a}}\ u\ s)$;
2) $\mathsf{ok}\, u\, r \Longrightarrow \mathsf{pred}_{raw}\ (\boxed{\mathsf{ok}}_V\ u)\ (\mathsf{f}_{raw}\ u\ r)$;
3) $\boxed{\mathsf{ok}}\, u\, s \Longrightarrow \boxed{\mathsf{a}}\ (u \triangleright 1)\ (\boxed{\Uparrow}\ s) = \mathsf{map}_X\ \boxed{\Uparrow}_V\ (\boxed{\mathsf{a}}\ u\ (\mathsf{map}_{sh}\ \mathsf{a}\ s))$;
4) $\mathsf{ok}\, u\, r \Longrightarrow \mathsf{f}_{raw}\ (u \triangleright 1)\ (\Uparrow s) = \mathsf{map}_Y\ \boxed{\Uparrow}_V\ (\mathsf{f}_{raw}\ u\ (\mathsf{map}_{raw}\ \mathsf{a}\ s))$.

Requiring Y to be a functor is very restrictive, because it disallows many recursive functions with parameters. We generalize the entire construction to $\alpha\ Y = \alpha\ Y_1 \to \alpha\ Y_2$, 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 α 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 $\mathsf{b} : \alpha\ Y \to \mathsf{b} : (\alpha\ X, \alpha\ V\ Y)\ G$ and swapper $\mathsf{a} : \alpha\ V\ X \to \alpha\ X\ F$ as inputs and yields the function $\mathsf{f} : \alpha\ Y \to \alpha\ X\ T$, which satisfies $\mathsf{f}\ y = \mathsf{Cons}\ (\mathsf{map}_G\ \mathsf{id}\ (\mathsf{map}_T\ \mathsf{a} \circ \mathsf{f})\ (\mathsf{b}\ y))$.

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 $\alpha\ T = \mathsf{Ctor}\ ((\alpha, \alpha\ F\ T)\ G)$ 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 function definitions and that perform (co)induction.

For datatypes, step 1 starts by defining the type $\alpha\ T$, 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 $\mathsf{Q}\ t$ in terms of a fixed but unknown polymorphic predicate Q that is IAP and inductive (i.e., $(\forall x \in \mathsf{set}^2_G\ g.\ \mathsf{Q}\ x) \Longrightarrow \mathsf{Q}\ (\mathsf{Ctor}\ g)$ holds). Recursion is formalized as a function f defined such that the recursive equation $\mathsf{f}\ (\mathsf{Ctor}\ g) = \mathsf{b}\ (\mathsf{map}_G\ \mathsf{id}\ (f \circ \mathsf{map}_T\ \mathsf{a})\ g)$ 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, 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 `nonuniform_(co)datatype` commands can be used to define nonuniform types. For example, the following definition introduces a type of λ-terms over variables drawn from α, with De Bruijn notation for bound variables [8]:

`nonuniform_datatype` $\alpha\ tm =$
  $\mathsf{Var}\ \alpha \mid \mathsf{App}\ (\alpha\ tm)\ (\alpha\ tm) \mid \mathsf{Lam}\ ((unit + \alpha)\ tm)$

Entering a λ-abstraction (Lam) creates a new variable, which is accommodated by the extended type $\mathit{unit} + \alpha$ consisting of the values $\mathsf{Inl}\ ()$ (the new variable) and $\mathsf{Inr}\ x$ 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_{tm}, 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.[3] For example, the following definition introduces a function join that "flattens" a term whose variables are themselves terms:

`nonuniform_primrecursive` $\mathsf{join} : \alpha\ \mathit{tm}\ \mathit{tm} \to \alpha\ \mathit{tm}$ `where`
$\quad \mathsf{join}\ (\mathsf{Var}\ x) = x$
$\mid \mathsf{join}\ (\mathsf{App}\ s\ t) = \mathsf{App}\ (\mathsf{join}\ s)\ (\mathsf{join}\ t)$
$\mid \mathsf{join}\ (\mathsf{Lam}\ u) = \mathsf{Lam}\ (\mathsf{join}\ (\mathsf{map}_{tm}\ (\lambda x.\ \mathsf{case}\ x\ \mathsf{of}$
$\qquad \mathsf{Inl}\ () \Rightarrow \mathsf{Var}\ (\mathsf{Inl}\ ()) \mid \mathsf{Inr}\ y \Rightarrow \mathsf{map}_{tm}\ \mathsf{Inr}\ y)\ u))$

The command extracts blueprints and swappers from the equations and emits parametricity proof obligations that must be discharged by the user. In the example, the swapper is the λ-expression of type $\mathit{unit} + \alpha\ \mathit{tm} \to (\mathit{unit} + \alpha)\ \mathit{tm}$ that is passed to the outer map_{tm}. 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 λ-terms is substitution: $\mathsf{subst} : (\alpha \to \beta\ \mathit{tm}) \to \alpha\ \mathit{tm} \to \beta\ \mathit{tm}$. Due to the limitation that arguments to recursive functions must be BNFs, we cannot define higher-order functions like subst that depend on a type variable that changes in the recursive calls. But we can define subst as a composition: $\mathsf{subst}\ \sigma = \mathsf{join} \circ \mathsf{map}_{tm}\ \sigma$.

The `nonuniform_(co)induct` commands can be used to prove a lemma by (co)induction. For example, the command

`nonuniform_induct` s `in` *subst_subst*:
$\quad \mathsf{subst}\ \tau\ (\mathsf{subst}\ \sigma\ s) = \mathsf{subst}\ (\mathsf{subst}\ \tau \circ \sigma)\ s$

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 [26]. 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 [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 $\mathsf{R}\ l\ r \Leftrightarrow \exists x\ xs.\ l = \mathsf{const}\ x \wedge r = \mathsf{map}_{pstream}\ (\lambda\_.\ x)\ xs$ in the coinduction proof:

`nonuniform_codatatype` $\alpha\ \mathit{pstream} =$
$\quad \mathsf{Cons}\ \alpha\ ((\alpha \times \alpha)\ \mathit{pstream})$

`nonuniform_primcorecursive` $\mathsf{const} : \alpha \to \alpha\ \mathit{pstream}$
$\quad \mathsf{const}\ x = \mathsf{Cons}\ x\ (\mathsf{const}\ (x, x))$

`nonuniform_coinduct R in` *const_alt*:
$\quad \mathsf{const}\ x = \mathsf{map}_{pstream}\ (\lambda\_ : \alpha.\ x)\ xs$

VIII. Discussion and Related Work

a) Inspiration: We generalized Okasaki's construction [35] to a large class of datatypes. Nordhoff et al. [34] 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 combinators [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 [27].

Our (co)induction principles take advantage of the BNF structure including set operators and relators. They constitute a lightweight alternative to fibration-based approaches [17], [22] 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 [21], [32] 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 [4], [15], [25], [33], [39].

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., $\alpha\ \mathit{bush} = \mathsf{BNil} \mid \mathsf{BCons}\ \alpha\ (\alpha\ \mathit{bush}\ \mathit{bush})$. 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. Appendix D 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 *stree* from Section I is possible because the type $\alpha\ \mathit{fset}$ of finite sets is a BNF. This is excluded with DTT, which restricts datatypes to a predefined grammar. Second, since the foundational

[3] 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 [11] will be updated.

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] (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 [23], [24], [28], datatypes with references [16], 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], [29], and induction in intensional DTT [30]. 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 Engineering and Physical Sciences Research Council (EPSRC) via the grant EP/N019547/1, Verification of Web Services (VOWS). The authors are listed alphabetically.

REFERENCES

[1] M. G. Abbott, T. Altenkirch, and N. Ghani. Representing nested inductive types using W-types. In *ICALP 2004*, vol. 3142 of *LNCS*, pp. 59–71. Springer, 2004.

[2] A. Abel and R. Matthes. Fixed points of type constructors and primitive recursion. In *CSL 2004*, vol. 3210 of *LNCS*, pp. 190–204. Springer, 2004.

[3] A. Abel, R. Matthes, and T. Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.*, 333(1-2):3–66, 2005.

[4] N. Benton, C. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in Coq. *J. Autom. Reasoning*, 49(2):141–159, 2012.

[5] S. Berghofer and M. Wenzel. Inductive datatypes in HOL—Lessons learned in formal-logic engineering. In *TPHOLs '99*, vol. 1690 of *LNCS*, pp. 19–36, 1999.

[6] J. Biendarra. *Functor-Preserving Type Definitions in Isabelle/HOL*. B.Sc. thesis, Technische Universität München, 2015.

[7] R. S. Bird and L. G. L. T. Meertens. Nested datatypes. In *MPC '98*, vol. 1422 of *LNCS*, pp. 52–67. Springer, 1998.

[8] R. S. Bird and R. Paterson. De Bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, 1999.

[9] R. S. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Asp. Comput.*, 11(2):200–222, 1999.

[10] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In *ITP 2014*, vol. 8558 of *LNCS*, pp. 93–110. Springer, 2014.

[11] J. C. Blanchette, F. Meier, A. Popescu, and D. Traytel. Formalization and implementation accompanying this paper. http://matryoshka.gforge.inria.fr/pubs/nonuniform_archive.tgz, 2017.

[12] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion. In *ICFP 2015*, pp. 192–204. ACM, 2015.

[13] J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. In *ESOP 2015*, vol. 9032 of *LNCS*, pp. 359–382. Springer, 2015.

[14] A. Church. A formulation of the simple theory of types. *J. Symb. Logic*, 5(2):56–68, 1940.

[15] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *POPL 2008*, pp. 133–144. ACM, 2008.

[16] N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *TFP 2006*, vol. 7 of *Trends in Functional Programming*, pp. 173–188. Intellect, 2006.

[17] N. Ghani, P. Johann, and C. Fumex. Generic fibrational induction. *Logical Methods in Computer Science*, 8(2), 2012.

[18] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[19] E. L. Gunter. A broader class of trees for recursive type definitions for HOL. In *HUG '93*, vol. 780 of *LNCS*, pp. 141–154. Springer, 1994.

[20] J. Harrison. Inductive definitions: Automation and application. In *TPHOLs '95*, vol. 971 of *LNCS*, pp. 200–213. Springer, 1995.

[21] F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.

[22] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152, 1998.

[23] R. Hinze. Efficient generalized folds. In *Workshop on Generic Programming*, pp. 1–16, 2000. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

[24] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.

[25] A. Hirschowitz and M. Maggesi. Nested abstract syntax in Coq. *J. Autom. Reasoning*, 49(3):409–426, 2012.

[26] B. Huffman and O. Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *CPP 2013*, vol. 8307 of *LNCS*, pp. 131–146. Springer, 2013.

[27] J. Lambek. Subequalizers. *Canadian Mathematical Bulletin*, 13(1):337–349, 1970.

[28] C. E. Martin, J. Gibbons, and I. Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Asp. Comput.*, 16(1):19–35, 2004.

[29] R. Matthes. Recursion on nested datatypes in dependent type theory. In *CiE 2008*, vol. 5028 of *LNCS*, pp. 431–446. Springer, 2008.

[30] R. Matthes. An induction principle for nested datatypes in intensional type theory. *J. Funct. Program.*, 19(3-4):439–468, 2009.

[31] T. F. Melham. Automating recursive type definitions in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 341–386. Springer, 1989.

[32] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Symposium on Programming*, vol. 167 of *LNCS*, pp. 217–228. Springer, 1984.

[33] G. Naves and A. Spiwack. Balancing lists: A proof pearl. In *ITP 2014*, vol. 8558 of *LNCS*, pp. 437–449. Springer, 2014.

[34] B. Nordhoff, S. Körner, and P. Lammich. Finger trees. In *Archive of Formal Proofs*. http://afp.sf.net/entries/Finger-Trees.shtml, 2010.

[35] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[36] L. C. Paulson. A formulation of the simple theory of types (for Isabelle). In *COLOG-88*, vol. 417 of *LNCS*, pp. 246–274. Springer, 1990.

[37] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP '83*, pp. 513–523, 1983.

[38] J. J. M. M. Rutten. Relators and metric bisimulations. *Electr. Notes Theor. Comput. Sci.*, 11:252–258, 1998.

[39] M. Sozeau. PROGRAM-ing finger trees in Coq. In *ICFP '07*, pp. 13–24. ACM, 2007.

[40] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *LICS 2012*, pp. 596–605. IEEE Comput. Soc., 2012.

[41] P. Wadler. Theorems for free! In *FPCA '89*, pp. 347–359. ACM, 1989.

Appendix

A. (Weak) Parametricity and Naturality

We define and connect the various parametricity notions used in the paper. We fix two n-ary BNFs F and G.

A polymorphic constant $\mathsf{c} : \overline{\alpha}\,F$ is *parametric* if $\mathsf{rel}_F\,\overline{R}\,(\mathsf{c} :: \overline{\alpha}\,F)\,(\mathsf{c} : \overline{\beta}\,F)$ holds for all $R_i : \alpha_i \to \beta_i$. We generalize this notion from BNFs to the full function space. (Note that the function space $\alpha \to \beta$ is only a BNF in β, since BNFs are strictly positive functors.) A function $\mathsf{f} : \overline{\alpha}\,F \to \overline{\alpha}\,G$ between two BNFs is *parametric* if $(\mathsf{rel}_F\,\overline{R} \Rrightarrow \mathsf{rel}_G\,\overline{R})\;\mathsf{f}\;\mathsf{f}$ holds for all $\overline{R}$. Here we use the *function space relator* $R \Rrightarrow S$ defined as $\lambda f\,g.\,(\forall a\,b.\,R\,a\,b \Leftrightarrow S\;(f\;a)\;(g\;b))$. In other words, two functions are related by $R \Rrightarrow S$ if for all R-related inputs, the outputs are S-related. Using the function space relator parametricity can be defined for higher-order functions of arbitrary arity as well.

Parametricity is a strong requirement. For example, polymorphic equality is not parametric. A weaker requirement is *injective-parametricity*: $(\mathsf{rel}_F\,\overline{R} \Rrightarrow \mathsf{rel}_G\,\overline{R})\;\mathsf{f}\;\mathsf{f}$ must hold only for those $\overline{R}$ that are graphs of injective functions, i.e., left-total, single-valued relations. Polymorphic equality is injective-parametric, i.e. $(R \Rrightarrow R \Rrightarrow (\Leftrightarrow))\;(=)\;(=)$ for all left-total, single-valued R (where $(\Leftrightarrow)$ is the trivial relator for *bool*).

For some constants, in particular the existential and universal quantifiers, injective-parametricity is still too strong. In general, on predicates $\mathsf{P} : \overline{\alpha}\,F \to bool$, (injective-)parametricity requires $(\mathsf{rel}_F\,\overline{R} \Rrightarrow (\Leftrightarrow))\;\mathsf{P}\;\mathsf{P}$ to hold for all $\overline{R}$ (that are graphs of injective functions). By refining the $\Leftrightarrow$ relation, we obtain two new notions: *injective-monotone parametricity* (IMP) when $\Leftrightarrow$ is replaced with $\Longrightarrow$ and *injective-antitone parametricity* (IAP) when $\Leftrightarrow$ is replaced with $\Longleftarrow$. The existential quantifier is IMP in that $((R \Rrightarrow (\Leftrightarrow)) \Rrightarrow (\Longrightarrow))\;\exists\;\exists$ holds for all left-total, single-valued R. Dually, we have $((R \Rrightarrow (\Leftrightarrow)) \Rrightarrow (\Longleftarrow))\;\forall\;\forall$ for all left-total, single-valued R. Hence, the universal quantifier is IAP. Note that a predicate that is IAP and IMP is also injective-parametric.

Finally, we rephrase the above relational parametricity notions in terms of functions. The rich BNF structure allows us to connect those two worlds. The key notion in the functional world is that of a *natural transformation*: The function $\mathsf{f} : \overline{\alpha}\,F \to \overline{\alpha}\,G$ is a natural transformation if $\mathsf{f} \circ \mathsf{map}_F\,\overline{g} = \mathsf{map}_G\,\overline{g} \circ \mathsf{f}$ holds for all $\overline{g}$. On BNFs, this notion is equivalent to parametricity of f. While it is clear that parametricity implies naturality (by instantiating each R_i with the graph of the corresponding g_i), we sketch a proof of the converse direction.

Let $\mathsf{f} : \overline{\alpha}\,F \to \overline{\alpha}\,G$ be a natural transformation between two BNFs. We first note that, by naturality of both f and set^i_G for all $i \in [n]$, f never increases the set of components of an $x : \overline{\alpha}\,F$:

$$\forall x : \overline{\alpha}\,F.\,\forall i \in [n].\;\mathsf{set}^i_G\;(\mathsf{f}\;x) \subseteq \mathsf{set}^i_F\;x \qquad (\star)$$

Now we prove that f is parametric. For this, let $R_i : \alpha_i \to \beta_i \to bool$ for $i \in [n]$, $x : \overline{\alpha}\,F$ and $y : \overline{\beta}\,F$ such that $\mathsf{rel}_F\,\overline{R}\,x\,y$. From the definition of rel_F, we obtain $z \in \overline{\alpha \times \beta}\,F$ such that (1) $x = \mathsf{map}_F\,\overline{\mathsf{fst}}\,z$, (2) $y = \mathsf{map}_F\,\overline{snd}\,z$, and (3) $\mathsf{set}^i_F\,z \subseteq \{(x,y).\,R_i\,x\,y\}$ for all $i \in [n]$. To show $\mathsf{rel}_G\,\overline{R}\,(\mathsf{f}\;x)\,(\mathsf{f}\;y)$, we unfold the definition of rel_G and prove:

- $\mathsf{map}_G\,\overline{\mathsf{fst}}\,(\mathsf{f}\;z) = \mathsf{f}\;x$ using (1) and the naturality of f,
- $\mathsf{map}_G\,\overline{\mathsf{snd}}\,(\mathsf{f}\;z) = \mathsf{f}\;y$ using (2) and the naturality of f,
- $\mathsf{set}^i_G\,(\mathsf{f}\;z) \subseteq \{(x,y).\,R_i\,x\,y\}$ for all $i \in [n]$ using (3) and ($\star$).

Using similar arguments, we obtain equivalent functional characterizations for the weaker parametricity notions as well:

- f is injective-parametric iff $\mathsf{f} \circ \mathsf{map}_F\,\overline{g} = \mathsf{map}_G\,\overline{g} \circ \mathsf{f}$ holds for all injective functions $\overline{g}$.
- P is IMP iff $\mathsf{P}\;x \Longrightarrow \mathsf{P}\;(\mathsf{map}_F\;\overline{g}\;x)$ holds for all injective functions $\overline{g}$.
- P is IAP iff $\mathsf{P}\,(\mathsf{map}_F\,\overline{g}\,x) \Longrightarrow \mathsf{P}\;x$ holds for all injective functions $\overline{g}$.

B. Proof Sketch of Theorem 6

Since the types T_i do not exist yet, we need to work on the representation types raw_i, and prove a property about the predicates ok_i that represent the to-be-defined types T_i. For this, we introduce a refinement of the notion of witness: Given a shadow $u : \Delta$, a set $I \subseteq [\mathbf{i}]$ is an *u-witness* for raw_i if, for all sets $\overline{A}$, $\forall k \in I.\,A_k \neq \varnothing$ implies $\exists r \in raw_i\,\overline{A}.\,\mathsf{ok}_i\,u\,r$. We prove:

(1) $\mathsf{Lang}_i(\mathsf{Gr})$ (or $\mathsf{Lang}_{\infty,i}(\mathsf{Gr})$) is a perfect sets of $[]$-witnesses for the (co)datatype raw_i.

To prove (1), we are looking for a connection between the grammar Gr and raw_i's (co)recursive specification, in the direction of the destructor

$$\overline{\alpha}\;raw_i \xrightarrow{\mathsf{unRaw}_i} (\overline{\alpha\;sh}, \overline{\alpha}\;raw_{\sigma(1)}, \cdots, \overline{\alpha}\;raw_{\sigma(\mathbf{j})})\;G_i$$

By the definition of ok_i, if $r : \overline{\alpha}\;raw_i$ is such that $\mathsf{ok}_i\;u\;r$, then $\mathsf{dtor}_i\;r$ has, for each $j \in [\mathbf{j}]$, its j-components $r' : \overline{\alpha}\;raw_{\sigma(j)}$ satisfying $\mathsf{ok}_i\;(j \blacktriangleleft u)\;r'$. As discussed in Section III-B, the shadow increment between u and $j \blacktriangleleft u$ encodes the application of the components $\overline{F_j} = (F_{j1}, \ldots, F_{j\mathbf{k}})$, reflected in the grammar's productions of type 2. This suggests a recursive translation of shadows into polywits:

- $\gamma_k\;[] = \{\{k\}\}$;
- $\gamma_k\;(j \blacktriangleleft u) = (\gamma_1\;u, \ldots, \gamma_{\mathbf{k}}\;u) \cdot \mathscr{I}(F_{jk})$.

Now we can formulate a generalization of (1), taking into account arbitrary shadows, not just $[]$. For each nonterminal x, we write $\mathsf{Lang}_x(\mathsf{Gr})$ (or $\mathsf{Lang}_{\infty,x}(\mathsf{Gr})$) for the language (co)generated by x.

(2) For all $u : \Delta$, $\mathsf{Lang}_{(\gamma_k\,u,\ldots,\gamma_k\,u)\,t_i}(\mathsf{Gr})$ (respectively $\mathsf{Lang}_{\infty,(\gamma_k\,u,\ldots,\gamma_k\,u)\,t_i}(\mathsf{Gr})$) is a perfect set of u-witnesses for the (co)datatype raw_i.

Finally, (2) can be proved using standard (uniform) (co)induction and (co)recursion, moving back and forth between Gr-derivation trees and the raw_i's.

For datatypes: That every $I \in \mathsf{Lang}_{(\gamma_k\,u,\ldots,\gamma_k\,u)\,t_i}(\mathsf{Gr})$ is an u-witnesses follows by structural induction on its derivation tree in Gr. Conversely, that for every u-witness J, we have $I \subseteq J$ for some $I \in \mathsf{Lang}_{(\gamma_k\,u,\ldots,\gamma_k\,u)\,t_i}(\mathsf{Gr})$ follows by induction on the definition of ok_i.

For codatatypes: To prove that every $I_0 \in \mathsf{Lang}_{\infty,(\gamma_k\,u_0,\ldots,\gamma_k\,u_0)\,t_{i_0}}(\mathsf{Gr})$ is an u_0-witness, we let $\overline{A}$ be such that $\forall i \in I_0.\,A_i \neq \varnothing$. With u_0, i_0, I_0 and $\overline{A}$ fixed, let Tr be a (possibly infinite) derivation tree of $(\gamma_k\,u_0, \ldots, \gamma_k\,u_0)\,t_{i_0}$ in Gr—thus having I_0 as the set of terminals on its frontier. Let $\Delta_{u_0,i}$ consist of all shadows having u_0 as a prefix and such that the nonterminal $(\overline{\gamma}\,u)\,t_i$ occurs in the tree Tr, where we write $\overline{\gamma}\,u$ for $(\gamma_1\,u, \ldots, \gamma_{\mathbf{k}}\,u)$.

Mutually corecursively (by primitive *raw*-corecursion), we define the functions $w_i : \Delta_{u_0,i} \to \overline{\alpha}\;raw_i$, by

$$w_i\;u = \mathsf{map}_{G_i}\;\overline{\mathsf{id}}\;(w_{\sigma(1)}, \ldots, w_{\sigma(\mathbf{j})})\;g_u$$

where the element $g_u \in (\overline{\alpha\;sh}, \Delta_{u_0,\sigma(1)}, \cdots, \Delta_{u_0,\sigma(\mathbf{j})})\;G_i$ is defined as follows: Let $(\overline{\gamma}\;u)\,t_i \longrightarrow \Gamma_J$ be the (type 2) production in Tr corresponding to the terminal $(\overline{\gamma}\;u)\,t_i$.

- From the definition of Γ_J it follows that, for each j such that $\mathbf{k} + j \in J$, the nonterminal $((\overline{\gamma}\;u) \cdot \mathscr{I}(F_{j1}), \ldots, (\overline{\gamma}\;u) \cdot \mathscr{I}(F_{j\mathbf{k}}))\,t_{\sigma(j)}$, i.e., $(\gamma_1(j \blacktriangleleft u), \ldots, \gamma_{\mathbf{k}}(j \blacktriangleleft u))\,t_{\sigma(j)}$, is also in Tr; hence $j \blacktriangleleft u \in \Delta_{u_0,\sigma(j)}$.
- Also from the definition of Γ_J it follows that, for each $k \in [\mathbf{k}] \cap J$, the nonterminal $\gamma_k\;u$ is also in Tr. Since only type 1 productions are applicable to polywit nonterminals, let $\gamma_k\;u \to I$ be the production from Tr applied to $\gamma_k\;u$. Then I is included in Tr's frontier, i.e., $I \subseteq I_0$. Then, picking some elements $a_i \in A_i$ for $i \in I$, we can define shapes $s_k \in sh_k\;\overline{A}$ for each $k \in [\mathbf{k}]$ that are full trees, i.e., such that $\boxed{\mathsf{ok}}_u\;s_i$ holds.

Thus, we have constructed the elements $j \blacktriangleleft u \in \Delta_{u_0,\sigma(j)}$ for each j such that $\mathbf{k} + j \in J$ and $s_i \in sh_k\;\overline{A}$ (in particular, $s_i : \overline{\alpha}\;sh_k$) for each $k \in [\mathbf{k}]$. Since J is a witness for G_i, we obtain our desired element $g_u \in (\overline{\alpha\;sh}, \Delta_{u_0,\sigma(1)}, \cdots, \Delta_{u_0,\sigma(\mathbf{j})})\;G_i$, which concludes the definition of the w_i's. Because of our choices in the definition, it is now routine to prove the following:

- by rule coinduction on the definition of the ok_i's, that $\mathsf{ok}_i\; u\; (w_i\; u)$ holds;
- by rule induction on the definition of the set operators for *raw*, that $\mathsf{set}_{raw,k}(w_i\; u) \subseteq A_i$ holds, which means that $w_i\; u \in raw_i\; \overline{A}$ holds.

This concludes the proof that I_0 is a witness.

Conversely, to prove that for every u-witness J, we have $I \subseteq J$ for some $I \in \mathsf{Lang}_{\infty,(\gamma_k\; u,\ldots,\gamma_k\; u)\, t_i}(\mathsf{Gr})$, we construct a (possibly infinite) derivation tree whose frontier includes J. The construction proceeds corecursively, by extracting each time the next production to be applied from J and the G_i's witnesses. ☐

C. Fusion Laws

We fix a nonuniform datatype $\alpha\; T = (\alpha, \alpha\; F\; T)\; G$. Let us write $\mathsf{NURec}(X,Y,V,\mathsf{a},\mathsf{b})$ for the polymorphic function $\mathsf{f} : \alpha\; X\; T \to \alpha\; Y$ defined by nonuniform recursion from a blueprint $\mathsf{b} : (\alpha\; X, \alpha\; V\; Y)\; G \to \alpha\; Y$ and a swapper $\mathsf{a} : \alpha\; X\; F \to \alpha\; V\; X$, as in Section VI-A. (NURec cannot be a HOL combinator; it is simply a meta-level notation.)

Theorem 11: The following hold:

Fold fusion: If $\kappa : \alpha\; Y \to \alpha\; Y'$ is such that $\kappa \circ \mathsf{b} = \mathsf{b}' \circ \mathsf{map}_G\; \mathsf{id}\; \kappa$, then $\kappa \circ \mathsf{NURec}(X,Y,V,\mathsf{a},\mathsf{b}) = \mathsf{NURec}(X,Y',V,\mathsf{a},\mathsf{b}')$.

Map fusion: If $\kappa : \alpha\; X' \to \alpha\; X$ is such that $\kappa \circ \mathsf{a}' = a \circ \mathsf{map}_F\; \kappa$, then $\mathsf{NURec}(X,Y,V,\mathsf{a},\mathsf{b}) \circ \mathsf{map}_T\; \kappa = \mathsf{NURec}(X',Y,V,\mathsf{a}',\mathsf{b} \circ \mathsf{map}_G\; \kappa\; \mathsf{id})$.

Proof. For fold fusion, we let $\mathsf{f} = \mathsf{NURec}(X,Y,V,\mathsf{a},\mathsf{b})$ and $\mathsf{f}' = \mathsf{NURec}(X,Y',V,\mathsf{a},\mathsf{b}')$. Defining the predicate P by $\mathsf{P}\; g \Leftrightarrow \mathsf{f}\; g = \mathsf{f}'\; g$, we have that P is IAP (since f and f' are IAP, and so is the equality). Then P follows by nonuniform induction (rule schema $\mathsf{Ind}_T^{\mathsf{P}}$), chasing the left diagram of Figure 7 (where, as usual, we omit the mapping function for G)—induction essentially allows one to assume that $\kappa \circ \mathsf{f} = \mathsf{f}'$ holds on the right of the diagram (inside the set^2_G-componenents) and requires one to prove that it holds for the left of the diagram—this is trivial from the assumption $\kappa \circ \mathsf{b} = \mathsf{b}' \circ \mathsf{map}_G\; \mathsf{id}\; \kappa$ and the functoriality of the involved BNFs.

The case of map fusion is similar—depicted in the right diagram of Figure 7. As can be seen in the upper sub-diagram, here, besides functoriality, we also need that the destructor is a natural transformation (which is guaranteed by our construction and automatically delivered as a theorem when the type $\alpha\; T$ is defined). ☐

The duals of the funsion laws hold when $\alpha\; T$ is a nonuniform codatatype as in Section VI-B.

D. Cross-Type Induction Schema

As discussed in the paper, a main restriction of our work is induction for nonuniform types, where we require IA-parametricity of the predicate. Here, we show how a gentle, provably consistent axiomatic extension of HOL removes this restriction. The axiom does not refer to nonuniform datatypes, or the intricate construction leading to them, or even to BNFs. Rather, it is a general-purpose axiom for cross-type well-founded induction and recursion.

We fix the types $\alpha\; T$, $\alpha\; F$ and M (with the notations T and F not connected to nonuniform datatypes).

Let $\mathsf{P} : \alpha\; T \to bool$ be a polymorphic predicate, for which we want to prove $\forall \alpha.\; \forall t : \alpha\; T.\; \mathsf{P}\; t$. A natural approach would be induction using a measure $\mathsf{m} : \alpha\; T \to M$ which decreases w.r.t. a well-founded relation $\mathsf{r} : M\; set \to M\; set \to bool$. But what if the measure decreases by changing the type, as in $\mathsf{r}\; (\mathsf{m}\; t')\; (\mathsf{m}\; t)$, where $t : \alpha\; T$ and $t' : \alpha\; F\; T$? This is still acceptable, since well-foundedness should still operate across the types $\alpha\; F^n\; T$. Formally, we would like to have the following rule, where $\mathsf{wf}\; \mathsf{r}$ states that r is well-founded:

$$\frac{\begin{array}{l}\forall \alpha.\; \forall t : \alpha\; T.\\ \quad \mathsf{wf}\; \mathsf{r} \wedge (\forall t' : \alpha\; F\; T.\; \mathsf{r}\; (\mathsf{m}\; t')\; (\mathsf{m}\; t) \Longrightarrow \mathsf{P}\; t')\\ \quad \Longrightarrow \mathsf{P}\; t\end{array}}{\forall \alpha.\; \forall t : \alpha\; T.\; \mathsf{P}\; t}\; \mathsf{WFInd}_{T,F,\mathsf{r},\mathsf{m},\mathsf{P}}$$

In HOL, this kind of induction across types is impossible to justify (without requiring parametricity). Yet, it is safe:

Theorem 12: The rule schema WFInd is sound in the standard models of HOL [A4] and in the ground models of Isabelle/HOL [A2]. Hence it is consistent with HOL and Isabelle/HOL.

We treat Isabelle/HOL specially because it allows ad hoc overloading intertwined with type definitions, which is problematic in the standard HOL semantics [A2], [A3].

Proof. In short, the schema is consistent with HOL because it is clearly sound in the set-theoretic interpretation of HOL. The following argument elaborates this idea.

A standard model of HOL fixes a universe $\mathscr{U}$, i.e., a set of sets with some closure properties (e.g., closure under function spaces). It interprets a type constructor such as T as a function on this universe, $[T] : \mathscr{U} \to \mathscr{U}$, a type such as M as an element of the universe, $[M] \in \mathscr{U}$, etc. Moreover, a polymorphic constant such as $\mathsf{m} : \alpha\; T \to M$ is interpreted as a $\mathscr{U}$-indexed family $([\mathsf{m}]_A)_{A \in \mathscr{U}}$ where $[\mathsf{m}](A) : T(A) \to [M]$. Crucially, it interprets the function-space type constructor as the set of *all* functions between the interpretation of its arguments and the type *nat* as a countable set $[nat]$, which with $[0]$ and $[\mathsf{Suc}]$ is isomorphic to the natural numbers.

This means that the scheme WFInd can be justified inside $\mathscr{U}$ as follows: Assuming its conclusion is false and repeatedly using its hypothesis, we obtain the infinite seqeuences $(A_i)_i$ and $(b_i)_i$ such that $A_{i+1} = [F](A_i)$, $b_i \in [T](A_i)$ and $[\mathsf{r}]\; ([\mathsf{m}](A_{i+1})(b_{i+1}))\; ([\mathsf{m}](A_i)(b_i)) = [\mathsf{True}]$. Taking $c_i = [\mathsf{m}](A_i)(b_i)$, this gives us an infinite sequence $(c_i)_i$ such that $c_i \in [M]$ and $[\mathsf{r}]\; (c_{i+1})\; (c_i) = [\mathsf{True}]$. Thanks to standardness, $(c_i)_i$ yields a witness for the formula $\exists c' : nat \to M.\; \forall i : nat.\; \mathsf{r}\; (c'\; i)\; (c'\; i)$, which therefore holds in the model. This contradicts the fact that $[\mathsf{wf}\; \mathsf{r}]$ also holds in the model.

A ground model of Isabelle/HOL only interprets the ground (monomorphic) types and terms, again with a standard interpretation for functions and numbers. A formula is true in such a model iff all its ground substitutions are true. For example, $\forall x : \alpha.\; x = x$ is true because, for all ground types K, the ground formula $\forall x : K.\; x = x$ is true. The argument for why the schema WFInd is sound is similar to the case of standard HOL models, but employing ground types K_i instead of the sets A_i. ☐

With this addition, we can remove the parametricity requirement from Theorem 8:

Theorem 13: The Ind schema is derivable in HOL enriched with the WFInd schema.

Proof. The derivation takes place by instantiating the parameters of $\mathsf{WFInd}_{T,F,\mathsf{r},\mathsf{m},\mathsf{P}}$ using those of $\mathsf{Ind}_T^{\mathsf{P}}$. We take

- T and F as in the nonuniform datatype definition $\alpha\; T = (\alpha, \alpha\; F\; T)\; G$;
- M to be the uniform datatype $M = \mathsf{MCons}\; ((unit, M)\; G)$;
- r to be the immediate subterm relation associated to M, namely $\mathsf{r}\; m\; m' \Leftrightarrow m' \in \mathsf{set}^2_G\; (\mathsf{MCons}\; m)$;
- m to be the composition $\mathsf{rawmeas} \circ \mathsf{Rep}_T$, where $\mathsf{Rep}_T : \alpha\; T \to \alpha\; raw$ is the representation function for T and $\mathsf{rawmeas} : \alpha\; raw \to M$ sends any r to its recursive depth:

$$\mathsf{rawmeas}\; (\mathsf{Raw}\; g) = \mathsf{MCons}\; (\mathsf{map}_G\; (\lambda\_.\; ())\; \mathsf{rawmeas}\; g)$$

With these parameters in place, it is not hard to verify that the assumptions of $\mathsf{WFInd}_{T,F,\mathsf{r},\mathsf{m},\mathsf{P}}$ hold. ☐

In summary, the unrestricted version of nonuniform induction is available in a consistent axiomatic extension of HOL. Users can choose between enabling this axiom or using the more restricted rule that depends on parametricity.

References

[A1] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[A2] O. Kunčar and A. Popescu. A consistent foundation for Isabelle/HOL. In *ITP 2015*, vol. 9236 of *LNCS*, pp. 234–252. Springer, 2015.

$$
\begin{tikzcd}
\alpha\, X\, T \arrow[rr, "\mathsf{dtor}"] \arrow[dd, "\mathsf{f}"'] \arrow[dddr, "\mathsf{f}'"] & & (\alpha\, X, \alpha\, X\, F\, T)\, G \arrow[d, "\mathsf{id}"', "\mathsf{map}_T\ \mathsf{a}"] & \\
& & (\alpha\, X, \alpha\, V\, X\, T)\, G \arrow[d, "\mathsf{id}"', "\mathsf{f}"] \arrow[ddr, bend left, "\mathsf{id}"', "\mathsf{f}'"] & \\
\alpha\, Y \arrow[dr, "\kappa"'] & & (\alpha\, X, \alpha\, V\, Y)\, G \arrow[ll, "\mathsf{b}"'] \arrow[dr, "\mathsf{id}"', "\kappa"] & \\
& \alpha\, Y' & & (\alpha\, X, \alpha\, V\, Y')\, G \arrow[ll, "\mathsf{b}'"']
\end{tikzcd}
\qquad
\begin{tikzcd}
& \alpha\, X'\, T \arrow[rr, "\mathsf{dtor}"] \arrow[dl, "\mathsf{map}_T\ \kappa"'] \arrow[dddl, "\mathsf{f}'"] & & (\alpha\, X', \alpha\, X'\, F\, T)\, G \arrow[dl, "\mathsf{id}"', "\mathsf{map}_{F\ T}\ \kappa"] \arrow[d, "\mathsf{id}"', "\mathsf{map}_T\ \mathsf{a}'"] \\
\alpha\, X\, T \arrow[rr, "\mathsf{dtor}"] \arrow[ddd, "\mathsf{f}"'] & & (\alpha\, X, \alpha\, X\, F\, T)\, G \arrow[d, "\mathsf{id}"', "\mathsf{map}_T\ \mathsf{a}"] & (\alpha\, X', \alpha\, V\, X'\, T)\, G \arrow[dl, "\kappa"', "\mathsf{map}_T\ \kappa"] \arrow[d, "\mathsf{id}"', "\mathsf{f}'"] \\
& & (\alpha\, X, \alpha\, V\, X\, T)\, G \arrow[d, "\mathsf{id}"', "\mathsf{f}"] & (\alpha\, X', \alpha\, V\, Y)\, G \arrow[dl, "\kappa"', "\mathsf{id}"] \\
\alpha\, Y & & (\alpha\, X, \alpha\, V\, Y)\, G \arrow[ll, "\mathsf{b}"'] &
\end{tikzcd}
$$

Fig. 7. Proof of the fusion laws: fold (left) and map (right)

[A3] O. Kunčar and A. Popescu. Comprehending Isabelle/HOL's consistency. In *ESOP 2017*, LNCS. Springer, 2017. To appear. Available at http://andreipopescu.uk/pdf/compr_IsabelleHOL_cons.pdf.

[A4] A. Pitts. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, chapter The HOL Logic, pp. 191–232. In Gordon and Melham [A1], 1993.