Foundational, Compositional (Co)datatypes for Higher-Order Logic

Category Theory Applied to Theorem Proving

Dmitriy Traytel\textsuperscript{1} Andrei Popescu\textsuperscript{1,2} Jasmin Christian Blanchette\textsuperscript{1}
\textsuperscript{1}: Technische Universität München, Germany
\textsuperscript{2}: Institute of Mathematics Simion Stoilow of the Romanian Academy
{traytel, popescua, blanchette}@in.tum.de

Abstract—Interactive theorem provers based on higher-order logic (HOL) traditionally follow the definitional approach, reducing high-level specifications to logical primitives. This also applies to the support for datatype definitions. However, the internal datatype construction used in HOL4, HOL Light, and Isabelle/HOL is fundamentally noncompositional, limiting its efficiency and flexibility, and it does not cater for codatatypes.

We present a fully modular framework for constructing (co)datatypes in HOL, with support for mixed mutual and nested (co)recursion. Mixed (co)recursion enables type definitions involving both datatypes and codatatypes, such as the type of finitely branching trees of possibly infinite depth. Our framework draws heavily from category theory. The key notion is that of a bounded natural functor—an enriched type constructor satisfying specific properties preserved by interesting categorical operations. Our ideas are formalized in Isabelle and implemented as a new definitional package, answering a long-standing user request.

Keywords—Category theory, higher-order logic, interactive theorem proving, (co)datatypes, cardinals

I. INTRODUCTION

Higher-order logic (HOL, Sect. II) forms the basis of several popular interactive theorem provers, notably HOL4 [8], HOL Light [13], and Isabelle/HOL [24]. Its straightforward semantics, which interprets types as sets (collections) of elements, makes it an attractive choice for many computer science and mathematical formalizations.

The theorem provers belonging to the HOL family traditionally encourage their users to adhere to the definitional approach, whereby new types and constants are defined in terms of existing constructs rather than introduced axiomatically. Following the LCF philosophy, theorems can be generated only within a small inference kernel, reducing the amount of code that must be trusted.

The definitional approach is a harsh taskmaster. At the primitive level, a new type is defined by carving out an isomorphic subset from an existing type. Higher-level mechanisms are also available, but behind the scenes they reduce the user-supplied specification to primitive type definitions.

The most important high-level mechanism is undoubtedly the datatype package, which automates the derivation of (freely generated inductive) datatypes. Melham [22] devised such a definitional package already two decades ago. His approach, considerably extended by Gunter [10], [11] and simplified by Harrison [12], now lies at the heart of the implementations in HOL4, HOL Light, and Isabelle/HOL.

Despite having withstood the test of time, the Melham–Gunter approach suffers from a few limitations that impair its usefulness. The most pressing issue is probably its ignorance of codatatypes (the coinductive counterpart of datatypes). Lacking a definitional package to automate the definition of codatatypes, users face a tough choice between tedious manual constructions and risky axiomatizations [7].

Creating a monolithic codatatype package to supplement the datatype package is not an attractive prospect, because many applications need to mix and match datatypes and codatatypes, as in the following nested-(co)recursive specification of finitely branching trees of possibly infinite depth:

\begin{verbatim}
datatype α list = Nil | Cons α (α list)

codatatype α tree1 = Node α ((α tree1) list)
\end{verbatim}

Ideally, users should also be allowed to define (co)datatypes with (co)recursion through well-behaved non-free type constructors, such as the finite set constructor \texttt{fset}:

\begin{verbatim}
codatatype α tree1 = Node α ((α tree1) fset)
\end{verbatim}

This paper presents a fully compositional framework for defining datatypes and codatatypes in HOL, including mutual and nested (co)recursion through an arbitrary combination of datatypes, codatatypes, and other well-behaved type constructors (Sect. III). The underlying mathematical apparatus is taken from category theory. Our type constructors are functors satisfying specific semantic properties; we call them bounded natural functors (BNFs). Unlike all previous approaches implemented in HOL-based provers, our framework imposes no syntactic restrictions on the type constructors that can participate in nested (co)recursion.

The main mathematical contribution of this paper is a novel class of functors—the BNFs—that is closed under an “local” way (Sect. IV). Cardinality reasoning with canonical membership-based well-orders lies beyond HOL’s expressive power, so we need a theory of cardinals that circumvents this limitation. Performing global categorical constructions in a weak, “local” formalism arguably constitutes the logical equivalent of walking on a tightrope.

We have formalized the development in Isabelle/HOL and are proceeding to implement a fully automatic definitional package for (co)datatypes based on these ideas to supplant the existing datatype package (Sect. V).
II. HIGHER-ORDER LOGIC (HOL)

By HOL we mean classical higher-order logic with Hilbert choice, the axiom of infinity, and rank-1 polymorphism. HOL is based on Church’s simple type theory [6]. It is the logic of Gordon’s original HOL system [8] and of its many successors and emulators. To keep the focus on the relevant issues, we present HOL not as a formal system but rather as a framework for expressing mathematics, much in the way that set theory is employed by working mathematicians.

The standard semantics of HOL relies on a universe $\mathcal{U}$ of types, ranged over by $\alpha, \beta, \gamma$, which we view as nonempty collections of elements. Membership of an element $a$ in a type $\alpha$ is written $a : \alpha$. The type unit consists of a single element $(\cdot)$, bool is the Boolean type, and nat is the type of natural numbers. Fixed elements of types, such as $(\cdot)$ : unit, are called constants. Given $\alpha$ and $\beta$, we can form the type $\alpha \to \beta$ of (total) functions from $\alpha$ to $\beta$. If $f : \alpha \to \beta$ and $a : \alpha$, then $f a : \beta$ is the result of applying $f$ to $a$. The types $\alpha + \beta$ and $\alpha \times \beta$ are the disjoint sum and the product of $\alpha$ and $\beta$, respectively. For $n$-ary functions, we generally prefer the curried form $f : a_1 \to \cdots \to a_n \to \beta$ to the tuple form.

HOL supports a restrictive, simply typed flavor of set theory. We write $\alpha$ set for the powertype of $\alpha$, consisting of sets of $\alpha$ elements; it is isomorphic to $\alpha \to \text{bool}$. The universe set of $\alpha$, $U_\alpha : \alpha$ set, is the set consisting of all the elements of $\alpha$. For notational convenience, we sometimes write $\alpha$ instead of $U_\alpha$. Given an element $a : \alpha$ and a set $A : \alpha$ set, $a \in A$ tests whether $a$ belongs to $A$. Although the two concepts are related, set membership is not to be confused with type membership. Given a type $\alpha$ and a predicate $\varphi : \alpha \to \text{bool}$, we can form by comprehension the set $\{a : \alpha. \varphi a\}$ of type $\alpha$ set.

While unit, bool, and nat are types in their own right, set, $\to$, $+$, and $\times$ are type constructors, i.e., functions on the universe of types. The first of these is unary, and the last three are binary. Types are a special case of type constructors, with arity 0. We can introduce new type constructors by combining existing type constructors and comprehension; for example, we can define the ternary type constructor $(a_1, a_2, a_3) F$ as $(a_2 + a_1) \times (a_3 \text{ set})$. Except for infix operators, type constructor application is written in postfix notation (e.g., $a F$), whereas function application is written in prefix notation (e.g., $f a$).

Depending on the context, $(a_1, \ldots, a_n) F$ either denotes the application of $F$ to $(a_1, \ldots, a_n)$ or simply indicates that $F$ is an $n$-ary type constructor. We abbreviate $(a_1, \ldots, a_n) F$ to $\overline{a}_n F$. Given a binary type constructor $(a_1, a_2) F$ and a fixed type $\beta$, $(\cdot, \beta) F$ denotes the unary type constructor sending an arbitrary type $\alpha$ to $(\alpha, \beta) F$, and similarly for $(\beta, \cdot) F$.

As the main primitive way of introducing custom types, HOL lets us carve out from a type $\alpha$ the type corresponding to a nonempty set comprehension $A = \{a : \alpha. \varphi a\}$, yielding a type $\beta$ and an injective function $f : \beta \to \alpha$ whose image is $A$.

Polymorphic constants can be regarded as families of constants indexed by types. For example, the identity function $id : \alpha \to \alpha$ is defined for any type $\alpha$ and corresponds to a family $(id_\alpha)_{\alpha \in \mathcal{U}}$. $Id : (\alpha \times \alpha)$ set is the identity relation. Function composition $\circ$ has type $(\alpha \to \beta) \to (\beta \to \gamma) \to \alpha \to \gamma$. Type arguments can be indicated by a subscript (e.g., $U_\alpha$).

HOL is significantly weaker than the set theories popular as foundations of mathematics, such as Zermelo–Fraenkel with the axiom of choice (ZFC). Some standard mathematical constructions cannot be performed in HOL, notably those dealing with proper classes or families of unboundedly large sets. A typical example is the representation of the HOL semantics, due to the unbounded nature of the simple type hierarchy. Another example is the standard (membership-based) theory of ordinals and cardinals, which involves the well-ordered class of ordinals. Nonetheless, many standard mathematical constructions are local, meaning that they are performed within an arbitrary but fixed universe set. These are particularly well suited to (polymorphic) HOL. Examples include basic algebra and analysis, formal language theory, and structural operational semantics. A large body of mathematics can be expressed adequately in HOL, as witnessed by the extensive library developments in HOL-based provers.

III. DATATYPES IN HOL

The limitations of HOL mentioned above may seem exotic and contrived. Yet our application—datatype definitions—is precisely one of those areas where HOL’s lack of expressiveness is most painfully felt. Category theory offers a powerful, modular methodology for constructing (co)datatypes, but filling the gap between theoretical category theory and theorem proving in HOL, with its simply typed set theory, is challenging; indeed, it is the main concern of this paper.

A. The Melham–Gunter Approach

Melham’s original datatype package [22] is based on a manually defined polymorphic datatype of finite labeled trees, from which simple datatypes are carved out as subtypes. Gunter [10] generalized the package to support mutually recursive datatypes. She also showed how to reduce specifications with nested recursion to mutually recursive specifications. A typical example is the recursive occurrence of a tree $\alpha$ nested in the list type constructor in the definition of finite trees. To define such a type, Gunter unfolds the definition of list, resulting in a mutually recursive definition of trees ($\alpha \text{ tree}_\alpha$) and “lists-of-trees” ($\alpha \text{ tree}_\alpha$ list):

\[
\text{datatype } \alpha \text{ tree}_\alpha = \text{Node}_\alpha (\alpha \text{ tree}_\alpha) \\
\text{and } \alpha \text{ tree}_\alpha \text{ list} = \text{Nil} | \text{Cons} (\alpha \text{ tree}_\alpha) (\alpha \text{ tree}_\alpha \text{ list})
\]

Exploiting an isomorphism, the package translates occurrences of $\alpha \text{ tree}_\alpha - \text{list}$ to $(\alpha \text{ tree}_\alpha)$ list, maintaining to a large extent the illusion of nested recursion. Orthogonally, Gunter [11] extended Melham’s labeled trees with infinite branching, to support positive recursion through functions.
The handling of mutual and nested recursion has several disadvantages, all related to its nonmodularity. Most importantly, it is not clear how to extend the approach to nested recursion and corecursion or to non-free constructors. In addition, some of the internal aspects of the construction are visible to the user (e.g., in the type of the iterator used to define primitive recursive function). Finally, replaying recursive definitions and transferring results via isomorphisms is prohibitively slow for datatypes with many layers of nesting.

B. Bringing HOL Closely to Category Theory

Let $a \ F$ be a unary type constructor. Category theory has elegant devices to define, based on $F$, the associated datatype and codatatype by solving the equation $\alpha \cong a \ F$ (up to isomorphism) in a minimal and maximal way, obtaining the initial $F$-algebra and final $F$-coalgebra, respectively. However, this requires $F$ to be complemented by an action on functions between types, usually called a “map.”

The universe of types $\mathcal{U}$ naturally forms a category where the objects are types and the morphisms are functions between types. We are interested in type constructors $(a_1, \ldots, a_n) \ F$ that are also functors on $\mathcal{U}$, i.e., that are equipped with an action on morphisms commuting with identities and composition. Taking advantage of polymorphism, this action can be expressed as a constant $F \map{id}$ and the constant map function $F \map{f_1} \ldots (g_n \circ f_n) = (F \map{g}) \circ (F \map{f})$.

Let us review some basic functors.

$(n, \alpha)$-constant functor $(C_{n, \alpha}, C\map{map}_{n, \alpha})$: The $(n, \alpha)$-constant functor $(C_{n, \alpha}, C\map{map}_{n, \alpha})$ is the $n$-ary functor consisting of the constant type constructor $(\beta_1, \ldots, \beta_n) C_{n, \alpha} = \alpha$ and the constant map function $C\map{map}_{n, \alpha} f_1 \ldots f_n = \text{id}$.

Sum functor $(+, \otimes)$: Let $\text{fst}: a_1 \times a_2 \to a_1$ and $\text{snd}: a_1 \times a_2 \to a_2$ denote the two standard projection functions. Given $f_1: a_1 \to \beta$ and $f_2: a_2 \to \beta$, let $(f_1, f_2): a_1 + a_2 \to \beta$ be the function $\text{Inl} a_1 f_1$ and $\text{Inr} a_2 f_2$. Given $f_1: a_1 \to \beta_1$ and $f_2: a_2 \to \beta_2$, let $f_1 \otimes f_2: a_1 + a_2 \to \beta_1 + \beta_2$ be $\text{Inl} f_1 f_1 \text{Inr} f_2 f_2$.

Product functor $(\times, \otimes)$: Let $\text{fst}: a_1 \times a_2 \to a_1$ and $\text{snd}: a_1 \times a_2 \to a_2$ denote the two standard projection functions. Given $f_1: a_1 \to \beta_1$ and $f_2: a_2 \to \beta_2$, let $(f_1, f_2): a_1 \times a_2 \to \beta_1 \times \beta_2$ be the function $\text{Inl} a_1 f_1 \otimes \text{Inr} a_2 f_2$. Given $f_1: a_1 \to \beta_1$ and $f_2: a_2 \to \beta_2$, let $f_1 \otimes f_2: a_1 \times a_2 \to \beta_1 \times \beta_2$ be $(f_1 \otimes \text{fst}, f_2 \otimes \text{snd})$.

Function space functor $\text{func}_\alpha$: The given a type $\alpha$, $\beta$ $\text{func}_\alpha = \alpha \to \beta$. For all $f: \beta_1 \to \beta_2$, we define $\text{comp}_f: \beta_1 \text{func}_\alpha \to \beta_2 \text{func}_\alpha$ as $\text{comp}_f g = f \circ g$.

Power type functor $\text{set}$: The function image $f: \alpha \to \beta$ send each set $A$ to the image of $A$ through the function $f: \alpha \to \beta$.

$k$-Power type functor $\text{set}_k$: Given a cardinal $k$, for all types $\alpha$, we define the type $\alpha \text{set}_k$ by comprehension, carving out from $\alpha$ only those sets of cardinality $\leq k$.

While specific map functions are heavily used in HOL theories (e.g., map, image), the theorem provers traditionally do not record the functional structure $F \map{map}$ of $F$ or take advantage of it when defining datatypes. The next examples illustrate the benefits of keeping such additional structure.

Finite lists: The unary type constructor list, which sends each type $\alpha$ to the type $\alpha$ list of lists of $\alpha$ elements, is categorically given as the initial algebra on the second argument of the binary functor $(F, F \map{map})$, where $(\alpha, F) F = \text{unit} + \alpha \times \beta$ and $F \map{map} g = \text{id} \oplus f \circ g$. More precisely, there exists a (polymorphic) folding bijection $\text{fld}: (\alpha, \alpha \text{ list}) F \to \alpha$ list making $(\text{fld}, \alpha \text{ list})$ the initial algebra for the unary functor $(\alpha, \_ ) F$. Here, $\text{fld} = (\text{Nil}, \text{Cons})$, where Nil and Cons are the familiar list operations. The initial algebra property corresponds to the availability of the standard iterator for lists. Then $(\text{list}, \text{map})$ is itself a unary functor.

Finitely branching trees of finite depth: Defining lists is hardly a spectacular achievement. It is the abstract interface to lists that makes category theory relevant: $(\text{list}, \text{map})$ is simply another functor available for nesting in (co)datatype definitions. Assume we want to define finitely branching trees of finite depth. This involves taking the initial algebra $\alpha \text{tree}_F$ on the second argument of the functor $(G, G \map{map})$, where $(\alpha, \beta) G = \alpha \times \beta \text{ list}$ and $G \map{map} f g = f \otimes \text{map} g$. The resulting iterator $\text{iter}$ has the polymorphic type $(\alpha \times \beta \text{ list}) F \to \alpha \text{ tree}_F \to \beta$, and its characteristic equation is $\text{iter} s \circ \text{fld} = s \circ (\text{id} \otimes \text{map} (\text{iter} s))$, where $\text{fld}$ is the folding bijection associated to $\alpha \text{tree}_F$ (Fig. 1). Thus, the “contract” of tree iteration reads as follows: Given tree-like structure on $\beta$ as the function $s: \alpha \times \beta \text{ list} \to \beta$ (viewing $\beta$ as consisting of “abstract trees,” with the constructor $s$), provide a function $\text{iter} s$ such that $\text{iter} s (f \circ \text{trl}) = s (a, \text{map} (\text{iter} s) \text{trl})$ for all $a: \alpha$ and $\text{trl}: (\alpha \text{ tree}_F)$ list. By using the map interface for accessing lists, the characteristic equation for $\text{iter}$ abstracts away from the definition of lists, enabling truly modular nesting of recursive types inside recursive definitions of larger types. The categorical approach also handles nested recursion through corecursion, as illustrated next.

Finitely branching trees of possibly infinite depth: To define trees of possibly infinite depth, we can take the final coalgebra $\alpha \text{tree}_I$ on the second argument of the functor $(G, G \map{map})$ defined above. The resulting coiterator $\text{coiter}$ has polymorphic type $\beta \to a \times \beta \text{ list}$ list to $\beta \to \alpha \text{ tree}_I$, and its characteristic equation is $\text{coiter} s \circ \text{fld} = s \circ (\text{id} \otimes \text{map} (\text{coiter} s)) \circ o$, where $\text{unf}$ is the unfolding bijection associated to $\alpha \text{tree}_I$ (Fig. 2). Normally, we would split $\text{unf}$ in two functions as $\text{unf} = (\text{lab}, \text{sub})$, where, for any $\text{tr}: a \text{ tree}_I \text{lab} \text{tr}: a$ is the label of the root and sub $\text{tr}$ is the list of its subtrees. Then, also splitting any $s: \beta \to a \times \beta \text{ list}$ similarly to $\text{unf}$ in two functions $L$ and $C$, the contract of tree coiteration reads as follows: Given a tree-like structure on $\beta$ consisting of functions $L: \beta \to \alpha$ and $C: \beta \to \beta \text{ list}$, yield a function $\text{coiter}_{(C, L)}$ such that $\text{lab} (\text{coiter}_{(C, L)} b) = L \beta b$ and $\text{sub} (\text{coiter}_{(C, L)} b) = \text{map} (\text{coiter}_{(C, L)} C \beta b)$ for all $b: \beta$.
Unordered finitely branching trees of possibly infinite depth: Assume that we want our finitely branching trees to be unordered. Instead of lists, we can employ finite multisets. We can then define $\alpha$ to be unordered. Instead of lists, we can employ finite multisets. Assume that we want our finitely branching trees $(\alpha, \beta)$ $s\mapsto \beta$

Fig. 1. Iterator for finitely branching trees of finite depth

\[ \alpha \times (\alpha \text{ tree}_F) \text{ list} \xrightarrow{\text{nd}} \alpha \text{ tree}_F \]
\[ \text{id} \otimes \text{map} (\text{iter} \ s) \]
\[ \alpha \times \beta \text{ list} \xrightarrow{s} \beta \]
\[ \text{coiter} \ s \]
\[ \text{id} \otimes \text{map} (\text{coiter} \ s) \]
\[ \alpha \text{ tree}_1 \xrightarrow{\text{unf}} \alpha \times (\alpha \text{ tree}_1) \text{ list} \]

Fig. 2. Coiterator for finitely branching trees of possibly infinite depth

\[ \beta \xrightarrow{s} \alpha \times \beta \text{ list} \]
\[ \text{coiter} \ s \]
\[ \text{id} \otimes \text{map} (\text{coiter} \ s) \]

A different result from Barr [3] states that any quotient functor of an $\omega$-bicontinuous functor admits a weakly final coalgebra obtained from any weakly final coalgebra of the latter. A subclass of $\omega$-bicontinuous that admits HOL-expressible (co)datatype constructions could prove to be an answer to C1–C4 via this result. In fact, the class $\mathcal{K}$ we adopt in this paper includes the class $\mathcal{K}'$ of functors $F$ that are quotients of Fbd-function-space functors, with Fbd a cardinal number depending on $F$. Whether $\mathcal{K}'$ is also a solution to C1–C4 remains for us an open question.

Finally, Hensel and Jacobs [14] propose a modular development of (co)datatypes for datafunctors, namely, functors obtained from constants, $+$, and $\times$ by repeated application of composition, initial algebra, and final coalgebra. Datafunctors satisfy C1–C3 but ostensibly not C4, because the arguments, which employ abstract results on categorical logic and fibrations [15], rely on (co)limits.

IV. Bounded Natural Functors

To accommodate constraints C1–C4 in HOL, we must work in a strict cardinal-bounded fashion, always keeping in sight a universe type able to host the necessary construction.
However, to stay flexible and not commit to a syntactically predetermined class of functors, we cannot a priori fix a
universe type, as required by the Melham–Gunter approach.

For example, there is no type that can accommodate an arbitrary iteration of the countable powertype construction.
Consequently, our functors will carry their cardinal bounds.

A useful means to keep cardinality under control is
the consideration of a natural “atom” structure potentially
available for the HOL type constructors in addition to
the map structure. Namely (assuming (F, Fmap) is unary), we consider
a polymorphic constant $\text{Fset} : \alpha F \to \alpha$ set, where $\text{Fset} x$
consists of all “atoms” of $x$; for example, if $F$ is list, $\text{Fset}$
returns the set of elements in the list.

We think of the elements $x$ of $\alpha F$ as consisting of a
shape together with a content that fills the shape with
elements of $\alpha$, with $\text{Fset} x$ returning this content in flattened
format, as a set (Fig. 3). This suggests that $\text{Fset}$ should
be a natural transformation between the functors $(F, \text{Fmap})$
and $(\text{set}, \text{image})$ (diagram in Fig. 4 commutative for all
$f : \alpha \to \beta$). $\text{Fset}$ allows us to internalize the type constructor
$F$ to sets of elements of given types $\alpha$. Namely, we define
$\text{Fin} : \alpha \text{ set} \to (\alpha \text{ set})$ by $\text{Fin} A = \{x : \alpha F. \ \text{Fset} x \subseteq A\}.

The generalization to $n$-ary functors is straightforward, with
$\text{Fin} A_1 \ldots A_n = \{x : (\alpha_1, \ldots, \alpha_n) F. \ \text{Fset}_i x \subseteq A_i\}.$
In particular, $\text{Fin} A_1 A_2 = \{x : (\alpha_1, \alpha_2) F. \ \text{Fset}_1 x \subseteq A_1, \ \text{Fset}_2 x \subseteq A_2\}$
(where the first occurrence of $\alpha_1$ abbreviates $U_{\alpha_1}$).

Combining the map and set operators and suitable cardinal
bounds, we obtain the following key notion, presented here
for the binary case. A binary bounded natural functor (BNF)
is a tuple $(F, \text{Fmap}, \text{Fset}, \text{Fbd})$, where

- $F$ is a binary type constructor,
- $\text{Fmap} : (\alpha_1 \to \beta_1) \to (\alpha_2 \to \beta_2) \to (\alpha_1, \alpha_2) F \to (\beta_1, \beta_2) F$,
- $\text{Fset}_i : (\alpha_1, \alpha_2) F \to \alpha_i$ set for $i \in \{1, 2\}$,
- $\text{Fbd}$ is an infinite cardinal number,

satisfying the following properties:

- $\text{FUNC} (F, \text{Fmap})$ is a binary functor.
- $\text{NAT}_1$ For all $\alpha_1$, $\text{Fset}_1$ is a natural transformation
  between $(\alpha_1, F, \text{Fmap})$ and $(\text{set}, \text{image})$.
- $\text{NAT}_2$ For all $\alpha_2$, $\text{Fset}_2$ is a natural transformation
  between $(\alpha_2, F, \text{Fmap})$ and $(\text{set}, \text{image})$.

Binary functors suffice to illustrate the functorial structure
of the initial and final algebras, a structure that would be
trivial if we started with unary functors. (The definition of
$n$-ary BNFs is given elsewhere [31].)

Among the above conditions, $\text{FUNC}$ and $\text{NAT}_1$ were
already explained and motivated. WP is a technical con-
dition allowing a smooth treatment of bisimilarity relations,
relevant for coinduction and corecursion [30]; unlike other
(weak) limits, weak pullbacks involve a finite number of
types and are hence expressible in HOL. $\text{CONG}$ states that
$\text{Fmap} f_1 f_2 x = \text{Fmap} g_1 g_2 x$.

The following cardinal-bound conditions hold:

- a. $\forall x : (\alpha_1, \alpha_2) F. \ |\text{Fset} x| \leq \text{Fbd} i$ for $i \in \{1, 2\}$;
- b. $|\text{Fin} A_1 A_2| \leq (|A_1| + |A_2| + 2)\text{Fbd}$.

WP $(F, \text{Fmap})$ preserves weak pullbacks.

$\text{CONG}$ If $\forall a \in \text{Fset}, x. \ f_i a = g_i a$ for all $i \in \{1, 2\}$, then
$\text{Fmap} f_1 f_2 x = \text{Fmap} g_1 g_2 x$.

$\text{CBD}$ The following cardinal-bound conditions hold:

- a. $\forall x : (\alpha_1, \alpha_2) F. \ |\text{Fset} x| \leq \text{Fbd} i$ for $i \in \{1, 2\}$;
- b. $|\text{Fin} A_1 A_2| \leq (|A_1| + |A_2| + 2)\text{Fbd}$.

Among the above conditions, $\text{FUNC}$ and $\text{NAT}_1$ were
already explained and motivated. WP is a technical con-
dition allowing a smooth treatment of bisimilarity relations,
relevant for coinduction and corecursion [30]; unlike other
(weak) limits, weak pullbacks involve a finite number of
types and are hence expressible in HOL. $\text{CONG}$ states that
$\text{Fmap} f_1 f_2 x$ is uniquely determined by the action of $f_i$
unary functors. (The definition of
$n$-ary BNFs is given elsewhere [31].)

Among the above conditions, $\text{FUNC}$ and $\text{NAT}_1$ were
already explained and motivated. WP is a technical con-
condition allowing a smooth treatment of bisimilarity relations,
relevant for coinduction and corecursion [30]; unlike other
(weak) limits, weak pullbacks involve a finite number of
types and are hence expressible in HOL. $\text{CONG}$ states that
$\text{Fmap} f_1 f_2 x$ is uniquely determined by the action of $f_i$
unary functors. (The definition of
$n$-ary BNFs is given elsewhere [31].)

Among the above conditions, $\text{FUNC}$ and $\text{NAT}_1$ were
already explained and motivated. WP is a technical con-
condition allowing a smooth treatment of bisimilarity relations,
relevant for coinduction and corecursion [30]; unlike other
(weak) limits, weak pullbacks involve a finite number of
types and are hence expressible in HOL. $\text{CONG}$ states that
$\text{Fmap} f_1 f_2 x$ is uniquely determined by the action of $f_i$
B. Composition

For composition, we focus on the binary–unary case (without loss of generality). Given unary BNFs $F_i = (F^i, \text{Fmap}^i, \text{Fset}^i, \text{Fbd}^i)$ with $i \in \{1, 2\}$ and a binary BNF $G = (G, \text{Gmap}, \text{Gset}, \text{Gbd})$, their composition is the unary BNF $H = G \circ (F_1, F_2)$ defined as follows:

- $(H, \text{Hmap})$ is the functorial composition of $(G, \text{Gmap})$ with $(F^i, \text{Fmap}^i)$;
- $\text{Hset} = \bigcup_{x \in \text{Gset}^1} \text{Fset}^1(x) \cup \bigcup_{x \in \text{Gset}^2} \text{Fset}^2(x)$;
- $\text{Hbd} = \text{Gbd} * (\text{Fbd}^1 + \text{Fbd}^2)$.

Although we seldom emphasize its role, composition is a pervasive auxiliary operation in interesting (co)datatype definitions. For example, the list-defining BNF $(\alpha, F)$ discussed in Sect. III-B is a composition of basic BNFs.

C. Relators

A key insight due to Rutten [29] is that, thanks to WP, the functor $(F, \text{Fmap})$ has a natural extension to a relator, i.e., a functor on the category of types and binary relations, denoted $\mathcal{R}$. We can express the relator action of $F$ as a polymorphic constant $\text{Rel} : (\alpha_1 \times \alpha_2) \rightarrow (\beta_1 \times \beta_2) \rightarrow \{(\alpha_1, \alpha_2) \rightarrow (\beta_1, \beta_2) \rightarrow \text{F}\}$ defined as $\text{Rel} Q R = \{(\text{Fmap} f, \text{Fmap} g) : f \beta_1 \rightarrow \beta_2, g \alpha_1 \rightarrow \alpha_2\}$.

For reasoning in HOL, it is convenient to take an alternative (equivalent) view of $\text{Rel}$, as an action on curried binary predicates $\text{Fpred} : (\alpha_1 \rightarrow \alpha_2 \rightarrow \text{bool}) \rightarrow (\beta_1 \rightarrow \beta_2 \rightarrow \text{bool}) \rightarrow (\alpha_1 \rightarrow \alpha_2) \rightarrow (\beta_1 \rightarrow \beta_2) \rightarrow \text{F} \rightarrow \text{F}$. $\text{Fpred} \varphi \psi$ should be regarded as the componentwise extension of the predicates $\varphi$ and $\psi$. For example:

- if $F$ is the product functor, $\text{Fpred} \varphi_1 \varphi_2 (a_1, a_2)(b_1, b_2)$
  $\Leftrightarrow \varphi_1 a_1 b_1 \land \varphi_2 a_2 b_2$;
- if $F$ is the sum functor, $\text{Fpred} \varphi_1 \varphi_2 a b = (\exists a^1. \text{fst} a = a^1 \land a = \text{lnr} a^1 \land b = \text{lnr} a^1 \land \varphi_1 a^1 b_1) \lor (\exists a^2. a = \text{lnr} a^2 \land b) \lor (\exists a^2. a = \text{lnr} a^2 \land \varphi_2 a^2 b_2)$.

D. The Categories of (Co)algebras

For this and the next two subsections, we fix a binary BNF $F = (F, \text{Fmap}, \text{Fset}, \text{Fbd})$. We first show how to construct in HOL the initial algebra (or, dually, the final coalgebra) on the second argument—that is, the minimal solution $\alpha \text{IF}$ (or maximal solution $\alpha \text{JF}$) of the equation $\alpha \equiv (\beta, \alpha) \text{F}$. The general constructions involve $n$ ($m + n$)-ary BNFs $F_i$ with type constructors $\beta, \alpha, F_1, \ldots, F_n$ and yield $n$-ary BNFs $\prod_{i=1}^n F_i$ (or $\sum_{i=1}^n F_i$) with their type constructors of the form $\beta \text{IF}_i$ (or $\beta \text{JF}_i$).

Abstractly, the theories of algebras and of coalgebras are dual, allowing a unified treatment of the basic (co)algebraic concepts. However, since the category of types is not self-dual, concrete constructions are often specific to each.

We fix a type $\beta$. A $(\beta, \alpha)$-algebra is a pair $\mathcal{A} = (A, s)$ where:

- $A : \alpha$ is the carrier set of $\mathcal{A}$ (and $\alpha$ is the underlying type of $\mathcal{A}$);
- $s : (\beta, \alpha) \rightarrow \alpha$ is the structural function of $\mathcal{A}$,

such that $A$ is closed under $s$, in that $\forall x \in \text{Fin} \beta A. s x \in A$ (and thus we may regard $s$ as a function $s : \text{Fin} \beta A \rightarrow A$). Dually, a $(\beta, \alpha)$-coalgebra is given by a pair $(\alpha, \cdot, s)$ such that $\forall x \in A. s x \in \text{Fin} \beta A$. Algebras form a category where morphisms $f : \mathcal{A}_1 \rightarrow \mathcal{A}_2$ (or maximal solution $\alpha_1 \rightarrow \alpha_2$) are functions $f : \alpha_1 \rightarrow \alpha_2$ such that the diagram on the left of Fig. 5 is commutative, and dually for coalgebras and the diagram on the right.

In the category of algebras, one can form products of families of algebras having the same underlying type, the carrier set of the product being the product of the carrier sets of the components. Dually, one can form sums of families of coalgebras using sums of sets. An algebra $\mathcal{A}$ is called initial if for all algebras $\mathcal{A}'$ there exists a unique morphism $f : \mathcal{A} \rightarrow \mathcal{A}'$, and weakly initial if we omit the uniqueness requirement. Dually, a coalgebra is final if it admits a unique morphism from any other coalgebra, and weakly final if uniqueness is dropped.

For reasoning, we are looking for a type constructor $\beta \text{IF}$ and function $\text{fld} : (\beta, \beta \text{IF}) \rightarrow (\beta, \beta \text{JF})$ such that the algebra $(\beta \text{IF}, \text{fld})$ is initial (dually, the coalgebra $(\beta \text{JF}, \text{uf})$ is final).

Typically, such a (co)algebra is obtained in two phases:

1. Construction of a weakly initial algebra (weakly final coalgebra) $\mathcal{A}$.
2. Construction of an initial algebra (final coalgebra) as a subalgebra (quotient coalgebra) of $\mathcal{A}$.

In the next two subsections, we discuss the key aspects of these constructions in HOL, both times starting with the simpler phase 2.

E. Initial Algebra

**Initial algebra from weakly initial algebra:** Given an algebra $\mathcal{A} = (A, s)$, let $M_s$ be the intersection of all sets $B$ such that $(B, s)$ is an algebra, and let $\mathcal{M}(\mathcal{A})$, the minimal subalgebra of $\mathcal{A}$, be $(M_s, s)$. It is immediate that there exists at most one morphism from $\mathcal{M}(\mathcal{A})$ to any other algebra. Then, a weakly initial algebra $\mathcal{A}$, if the desired initial $\beta$-algebra is its minimal subalgebra, $\mathcal{M}(\mathcal{A})$. Of course, $\mathcal{M}(\mathcal{A})$ depends on $\beta$ (which was fixed all along). Now $\beta \text{IF}$ is introduced by a type definition, carving out the underlying set of $\mathcal{M}(\mathcal{A})$ as a new type, and the folding map $\text{fld}$ is defined by copying on $\beta \text{IF}$ the structural map of $\mathcal{M}(\mathcal{A})$ (so that in effect $(\beta \text{IF}, \text{fld})$ becomes isomorphic to $\mathcal{M}(\mathcal{A})$).
Construction of a weakly initial algebra: This relies on a crucial lemma about the cardinality of minimal subalgebras, whose proof [31] employs the cardinality assumptions CBD.

Lemma 2: Let \( s : (\beta, \alpha) \to F \to \alpha \). Then \( |M_s| \leq (|\beta| + 2)^{\text{Suc} Fbd} \) (where Suc Fbd is the successor cardinal of Fbd).

Let \( \Theta \) be the set of all algebras \( A \) having as underlying type a type \( \gamma \) of sufficiently large cardinality, \( (|\beta| + 2)^{\text{Suc} Fbd} \); such a type exists, and in fact can be taken to be the very underlying type of this cardinal. The desired weakly initial algebra \( \mathcal{C} \) is the product of all algebras in \( \Theta \). Indeed, by Lemma 2, for any algebra \( \mathcal{B} \), its minimal subalgebra \( \mathcal{M}(\mathcal{B}) \) is isomorphic to one in \( \Theta \), to which \( \mathcal{C} \) has a projection morphism. This gives a morphism from \( \mathcal{C} \) to \( \mathcal{M}(\mathcal{B}) \), hence also one from \( \mathcal{C} \) to \( \mathcal{B} \). We have thus proved:

Prop. 3: \((\beta, IF, \text{fld})\) is the initial \( \beta \)-algebra.

This yields an iterator \( \text{iter} : ((\beta, \alpha) \to F) \to \beta \text{IF} \to \alpha \) such that \( \text{iter} \circ \text{fld} = s \circ \text{Fmap} \text{id} \circ \text{iter}(s) \) (cf. Fig. 1).

Structural induction: The set structure \( \text{Fset} \) of a BNF not only plays an auxiliary role in the datatype constructions but also provides a simple means to express induction abstractly, for arbitrary functors. Since \( \text{fld} \) is a bijection, for any element \( b \in \beta \text{IF} \) there is a unique \( y \in ((\beta, \beta) \text{IF}) F \) such that \( b = \text{unf} y \)—this is an abstract version of case analysis. Then the inductive components of \( b \) are precisely the elements of \( \text{Fset}_2 y \), and we have the following induction principle:

Prop. 4: Let \( \varphi : \beta \text{IF} \to \text{bool} \) and assume \( \forall y. (\forall b \in \text{Fset}_2 y. \varphi(b) \Rightarrow \varphi(\text{fld} y)) \). Then \( \forall b. \varphi(b) \).

For \( F = \text{unit} + \beta \times \alpha \) with \( \text{IF} = \text{list} \) (Sect. III-B), the above is equivalent to the familiar induction principle.

BNF structure: It is standard to define a functorial structure for the initial algebra, namely \( \text{IFmap} f = \text{iter} (\text{fld} \circ (\text{Fmap} f \text{id})) \). As for the set structure, consider \( b \in \beta \text{IF} \). Intuitively, \( \text{Fset} b \) should contain all the \( \text{Fset}_1 \) atoms of \( b \), then the \( \text{Fset}_1 \) atoms of its inductive components, and so on, iteratively. Moreover, as we have seen, delving into the inductive components is achieved by means of \( \text{Fset}_2 \). We are led to defining \( \text{Fset} \) as \( \text{iter} \) accept, i.e., as the unique function making the Fig. 6 diagram commutative, where \( \text{collect} a = \text{Fset}_1 a \cup \bigcup \text{Fset}_2 a \).

Prop. 5: \((\text{IF}, \text{IFmap}, \text{IFset}, 2^{\text{Fbd}})\) is a BNF.

As a BNF, \( \text{IF} \) is also a relator (Sect. C). Importantly for modular reasoning however, we can express \( \text{IFpred} \) directly in terms of \( \text{Fpred} \). Thus, \( \text{IFpred} \) is uniquely determined by the recursive equations \( \text{IFpred} \varphi (\text{fld} x_1) (\text{fld} x_2) \iff \text{Fpred} \varphi (\text{Fmap} \varphi x_1 x_2) \). For example, for the list functor, the above equation splits in the following, according to the relator structure of the component functors (unit, +, and \( \times \)):

- \( \text{list}_\text{pred} \varphi \text{Nil} \text{Nil} \Rightarrow \text{True} \),
- \( \text{list}_\text{pred} \varphi \text{Nil} (\text{Cons} \, b \, \text{bs}) \Rightarrow \text{False} \),
- \( \text{list}_\text{pred} \varphi (\text{Cons} \, a \, \text{as}) \text{Nil} \Rightarrow \text{False} \),
- \( \text{list}_\text{pred} \varphi (\text{Cons} \, a \, \text{as}) (\text{Cons} \, b \, \text{bs}) \iff \varphi a b \land \text{list}_\text{pred} \varphi \text{as bs} \).

revealing \( \text{list}_\text{pred} \) as the componentwise ordering on lists.

F. Final Coalgebra

Final coalgebra from weakly final coalgebra: This follows by the standard co-algebraic theory of bisimulation relations [30]. A bisimulation on a coalgebra \( \mathcal{A} = (A, s) \) is a relation \( R \subseteq A \times A \) such that \( (a, b) \in R \Rightarrow (\forall (a', b') \in R. \exists c \in \text{Fin} \beta R. \text{Fmap} \text{id} \text{fst} z = a \land \text{Fmap} \text{id} \text{snd} z = b) \). Indeed, by \( \beta \text{IF} \) on \( A \), the largest bisimulation on \( A \) is in the types-and-relations category \( \mathcal{R} \) such that \( (a, b) \in R \). Hence composition of bisimulations is a bisimulation, and then it follows easily that the largest bisimulation \( \text{LB}(\mathcal{A}) \) on a coalgebra \( \mathcal{A} \) is an equivalence relation, and that the resulting quotient coalgebra \( \mathcal{A}/_{\text{LB}(\mathcal{A})} \) has the property that any coalgebra has at most one morphism to it.

Now let \( \mathcal{C} \) be a weakly final coalgebra. By the above discussion, via an argument dual to the corresponding one for algebras, we have \( \mathcal{C}/_{\text{LB}(\mathcal{C})} \) final and based on it we define the desired type \( \beta \text{IF} \) and its unfolding bijection \( \text{unf} \).

Construction of a weakly final coalgebra: The abstract construction indicated in Rutten [30], as the sum of all coalgebras over a sufficiently large type (roughly dual to our weakly initial algebra construction), is possible in HOL thanks to our cardinality provisos. However, a more concrete construction gives us a better grip on cardinality, allowing us to check the BNF properties for the resulting coalgebra.

To lighten the presentation, we identify sets with types—for example, we allow ourselves to apply type constructors such as list to sets. Given a prefix-closed subset \( Kl \) of Fbd list and \( kl \in Kl \), we let \( \text{Suc}_{k\text{Lkl}} \), the set of \( k \)-successors of \( kl \), be \{\( kl \circ \text{Nil}, kl \circ \text{Nil} \in Kl \), where \( \circ \) denotes list concatenation and \( \text{Nil} \) the k-singleton list. We define an Fbd-tree to be a pair \((Kl, tr)\), where \( Kl \subseteq \text{Fbd} \) list is prefix closed and \( tr: Kl \rightarrow \text{Fin} \beta \text{Fbd} \) is such that \( \forall kl \in Kl. \text{Fset}_2 (tr kl) =\)

\( \text{collect} \).
Thus, Fbd-trees are at most Fbd-branching trees labeled as follows: Every node is labeled with an element of Fin β Fbd whose set of second-argument atoms consists of precisely the node’s emerging branches. Given a tree (Kl, tr), we define sub_{\,(Kl, tr)}: \{ k. [k] \in Kl \} \to C to send each k to the immediate k-subtree of (Kl, tr), more precisely, sub_{\,(Kl, tr)}k = (Kl′, tr′), where Kl′ = \{ k′l′. [k′] @ k′l′ \in Kl \} and tr′: Kl′ \to Fin β Fbd is defined by tr′ k l′ = tr (\{k\} @ k l′).

The set C of Fbd-trees can be naturally organized as a coalgebra \( C = (C, s) \) defining s (Kl, tr) = Fmap \text{id} sub_{\,(Kl, tr)} (tr Nil). Thus, s (Kl, tr) operates on (Kl, tr)’s root label tr Nil, substituting in its shape the immediate subtrees for the contents. Then \( C \) is shown to be a weakly final coalgebra by roughly the following argument. For each element \( a \) in an algebra (A, t), one defines its behavior tree by iterating the unfolding of a according to t—first a, then \( t a \), then \( t b \) for all \( b \in Fset_t (t a) \), and so on. Thanks to CDB-a, such trees are at most Fbd-branching, hence representable in C. We have thus proved:

\[
\text{Prop. 6:} \quad (β, β, \text{unf}) \text{ is the final } β\text{-coalgebra.}
\]

This yields a coiterator \( \text{coiter} : (a \to (β, β) F) \to a \to β \text{ JF} \) such that unf (\text{coiter} s) = Fmap \text{id} (\text{coiter} s) \circ s (cf. Fig. 2).

\section*{Structural coinduction:} Since LB(\( C \)) is the greatest bisimulation on \( C \), it follows that \( \text{Id} \) is the greatest bisimulation on the quotient coalgebra \( C_{/LB(\mathcal{C})} \). This gives us the following coinduction principle on \( β, β, \text{unf} \) (which is a copy of \( C_{/LB(\mathcal{C})} \)): If \( R \) is a bisimulation relation, then \( R \subseteq \text{Id} \). Viewing bisimilarities via the relator structure (cf. Fig. 7, left) and using the predicate notation, we can rephrase the coinduction principle as follows:

\[
\text{Prop. 7: Let } \varphi : β \text{ JF} \to β \to \text{bool} \text{ and assume } \forall a. \varphi a b \Rightarrow \text{Fpred } \text{Eq } \varphi (\text{unf } a) (\text{unf } b) \text{ (where } \text{Eq} : β \to β \to \text{bool} \text{ is the equality predicate). Then } \forall a. \varphi a b : a \Rightarrow a \Rightarrow b.
\]

\section*{BNF structure:} Again, the functorial structure of the final coalgebra is standard, namely, JFmap \( f = \text{coiter} ((\text{Fmap } \text{id}) \circ \text{unf}) \). Moreover, JFset can be defined by collecting all the Fset results of repeated unfolding, namely JFset \( a = \bigcup_{\text{nat}} \text{collect}_{1, a} \), where collect_{i, a} is defined recursively on i as follows: collect_{0, a} = \theta; collect_{i+1, a} = Fset_i (unf a) \cup \{ \text{collect}_{i, b}. b \in Fset_2 (unf a) \}. Similarly to JFpred, the relation JFpred can be described in terms of Fpred, by JFpred \( \varphi a_1 a_2 \Leftrightarrow \text{Fpred } \varphi (\text{JFpred } \varphi) (\text{unf } a_1) (\text{unf } a_2).

\[
\text{Prop. 8:} \quad (\text{JF, JFmap, JFset, Fbd}^{\text{Fbd}}) \text{ is a BNF.}
\]

\section*{V. Formalization and Implementation}

The results in this paper are formalized in Isabelle/HOL and implemented in ML as a prototypical definitional package, together with a few examples of applications. This development is publicly available [32].

\section*{A. Formalized Metatheory}

Isabelle/HOL proved well suited for formalizing category theory over types, with relevant concepts, including functor and natural transformation, handled in a lightweight, family-free notation as polymorphic types or constants. The main (co)algebraic constructions of this paper correspond to the theories named LFP and GFP in our formal development.

These constructions require a theory of cardinals in HOL, including cardinal arithmetic and regular cardinals. Simple type theory does not cater for ordinals as a canonical collection of well-orders, a very convenient concept for the standard theory of cardinals. Therefore, we worked with well-orders directly, dispersed polymorphically over types, with cardinals defined as well-orders minimal with respect to initial-segment embeddings. This theory and its challenges are presented separately [28].

\section*{B. Definitional Package}

Theorem 1 and its formalization form the basis of a new (co)datatype package for Isabelle/HOL. Users define (co)datatypes using an intuitive high-level specification syntax; internally, the package ensures that each specification corresponds to a BNF, defines the (co)datatype, and proves that the result is itself a BNF.

More specifically, each BNF is represented by an ML record consisting of the polymorphic constants and their properties as proved theorems, stored in Isabelle’s theory database [36, §4.1]. The basic BNFs for unit, +, ×, func, countable sets, and finite multisets are constructed in user space, as they do not require ML; users can construct and register custom BNFs in the same way.

In the simple (nonmutual) case, the package parses the right-hand side of a (co)datatype specification as a composition \( \mathcal{F} \) of already defined BNFs and proves that itself forms a BNF as in Sect. IV-B. Then the package defines the initial algebra or final coalgebra for \( \mathcal{F} \) and establishes automatically their characteristic theorems (for (co)recursion, (co)induction, etc.) and BNF structure as in Sect. IV-E or IV-F. All these are performed by specially tailored Isabelle tactics, whose running time is independent of the amount of nesting (unlike for the Melham–Gunter approach).

\section*{C. Example}

We demonstrate the definitional package on the type of finitely branching trees of possibly infinite depth [32]:

\begin{verbatim}
.codatatype \alpha tree = Node (lab: \alpha) (sub: (\alpha \cdot tree) list)
\end{verbatim}

The declaration syntax allows named selectors (lab and sub).

The command derives the expected characteristic theorems for \( \alpha \cdot tree \), including the coinduction rule

\begin{verbatim}
\forall a. \varphi a b \Rightarrow \text{lab a = lab b } \land \text{list_pred } \varphi (\text{unf } a) (\text{unf } b)
\end{verbatim}

where list_pred \( \varphi \) is the componentwise extension of \( \varphi \) to lists (Sect. IV-E). Corecursive (coiterative) functions can be defined using a convenient syntax; for example, tree reversal is specified below in terms of map and rev on lists:
corec trev where
lab (trev t) = lab t
sub (trev t) = rev (map trev (sub t))

Using the tree coinduction rule and Isabelle’s automation, we can prove the following lemma with a one-line proof:

lemma trev (trev t) = t

The (co)datatype package interacts seamlessly with the existing infrastructure for reasoning about (co)inductive predicates (defined via Knaster–Tarski), as illustrated by the following proof of König’s lemma for a tree. We first need a stream type to represent infinite paths in a tree:

codatatype α strm = SCons (hd: α) (tl: α strm)

Using the existing coinductive package, we can define the notions of an infinite tree and a proper path in a tree as greatest predicates satisfying the equations infinite t ⇐⇒ (∃u ∈ set (sub t). infinite u) and proper_path p t ⇐⇒ hd p = lab t ∧ (∃u ∈ set (sub t). proper_path (tl p) u). The corecursive function kpath uses Hilbert choice (e) to return a witness infinite path:

corec kpath where
hd (kpath t) = lab t
tl (kpath t) = kpath (e u. u ∈ set (sub t) ∧ infinite u)

We can then prove the desired lemma by coinduction:

lemma infinite t ⇒ proper_path (kpath t) t

VI. FURTHER RELATED WORK

Interactive theorem provers include various mechanisms for introducing new types, whether primitive (intrinsic), axiomatic, or definitional [5, p. 3]. In the world of HOL, the primitive type definition mechanism (Sect. II) and the datatype package (Sect. III-A) are the most widely used, but there are many others. Homeier [16] developed a package to define quotient types in HOL4, now ported to Isabelle [20]. Nominal Isabelle [33] extends HOL with infrastructure for reasoning about datatypes containing name binders; Urban is rebasin the onto the quotient package, possibly in union with our (co)datatype package, exploiting the support for non-free constructors. HOLCF, a HOL library for domain theory, has long included an axiomatic package for defining (co)recursive domains; Huffman [19] recast it into a purely definitional package, based on a large enough universal domain—a simplification that unfortunately is not available for general HOL datatypes. The package combines many of the categorical ideas present in our work, notably the modular mixture of recursion via enriched type constructors. Some ideas have yet to be automated in a definitional package: Völker [34] sketches a categorical approach to datatypes that prefigures our work; Vos and Swierstra [35] elaborate an ad hoc construction for recursion through finite sets; and Paulson [26] designed building blocks for codatatypes.

PVS, whose logic is a simple type theory extended with dependent types and subtyping (but without polymorphism), provides monolithic axiomatic packages for datatypes [25] and codatatypes [9]. Hensel and Jacobs [14] illustrate the categorical approach to (co)datatypes in PVS by axiomatic declarations of various flavors of trees (including our treef and tree2) with associated (co)iterators and proof principles. HOLω, which extends HOL4 with higher-rank polymorphism, provides a safe primitive for introducing abstractly specified types [17], [18]. Isabelle/ZF, based on ZFC, reduces (co)datatypes to (co)inductive predicates [27], with no support for mixed (co)recursion; for codatatypes, it relies on a concrete, definitional treatment of non-well-founded objects. In Agda and Coq, (co)datatypes are built into the underlying calculus. Mixed (co)recursion is possible [23] but not the combination with non-free types.

VII. CONCLUSION

We presented a theoretical framework for defining types in higher-order logic. The framework relies on the abstract notion of a bounded natural functor (BNF), consisting of a type constructor plus further categorical structure. BNFs are closed under composition and (co)algebraic fixpoints, providing all the necessary ingredients to define (co)datatypes.

Our solution is foundational: The characteristic (co)datatype theorems are derived from an internal construction, rather than stated as axioms. Unlike the traditional Melham–Gunter approach, our solution is also fully compositional, enabling mutual and nested (co)recursion involving arbitrary combinations of datatypes, codatatypes, and custom BNFs.

There is a large body of previous work on (co)datatypes as (co)algebras in category theory. Our main contribution has been to adapt this work to achieve compatibility with HOL’s restrictive type system. Our ideas are implemented in a prototypical definitional package for Isabelle/HOL. The package is expected to be included in the next official release of the theorem prover, making Isabelle the first HOL-based prover with general support for codatatypes and thereby answering a long-standing user request.

After implementing the original datatype package for Isabelle, Berghofer and Wenzel [5] suggested three areas for future work: codatatypes, non-freeely generated types, and composition of definitional packages. Thirteen years later, their vision is very close to a full materialization. Although we focused on Isabelle, our approach is equally applicable to the other HOL-based theorem provers, such as HOL4 [8] HOL Light [13], and ProofPower–HOL [2].

Methodologically, we found that category theory helped us develop intuitions about the types of HOL, recasting them as richly structured objects rather than mere collections of elements. As a continuation of this program, we want to rebuke the myth that parametricity is inapplicable to HOL by extending BNFs with a parametricity predicate and exploiting their relator nature. We also intend to transfer further category theory insight, such as the (co)induction mixture of Jacobs et al. [14], [15], to the world of theorem provers.
Acknowledgment: We thank Tobias Nipkow for encouraging this work, Brian Huffman, Christian Urban, and Makarius Wenzel for their advice regarding Isabelle package writing, and Andreas Lochbihler for inspiring discussions. The research was supported by the project Security Type Systems and Deduction (grant Ni 491/13-1) as part of the priority program 1496) of the Deutsche Forschungsgemeinschaft (DFG). The third author was supported by the DFG project Quis Custodiet (grant Ni 491/11-2).

REFERENCES