FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

A category theory based (co)datatype package for Isabelle/HOL

Dmytro Traytel

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

A category theory based (co)datatype package for Isabelle/HOL

Eine kategorielle Konstruktion von Ko-/Datentypen in Isabelle/HOL

Author:	Dmytro Traytel
Supervisor:	Prof. Tobias Nipkow, Ph.D.
Advisor:	Andrei Popescu, Ph.D.
Advisor:	Jasmin Christian Blanchette
Date:	April 17, 2012

I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, April 17, 2012 Dmytro Traytel

Abstract

Higher-order logic (HOL) forms the basis of several popular interactive theorem provers. These 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*—a functor 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.

Zusammenfassung

Logik höherer Stufe (engl. higher-order logic, HOL) macht den Kern von mehreren erfolgreichen interaktiven Beweisassistenten wie HOL4, HOL Light and Isabelle/HOL aus. Um diesen Erfolg zu verdienen, müssen die Beweisassistenten vertrauenswürdig sein. Deswegen ist es empfehlenswert den im hohen Maße vertauenswürdigen Kern von HOL nur konservativ zu erweitern, indem man höhere Programmiersprachenkonstrukte auf primitive Konstrukte zurückführt. Das trifft auch auf die Definition induktiver Datentypen zu. Derzeit verwendete Implementierungen erfüllen dieses Kriterium. Sie haben allerdings ihre Nachteile in Bezug auf Flexibilität und Performanz, begründet durch einen nicht modularen Ansatz. Zudem ist es unklar ob mit dem gleichen Ansatz Kodatentypen (die koinduktiven Gegenspieler von Datentypen) erfasst werden können.

Diese Arbeit entwickelt einen vollständig modularen Ansatz zur Konstruktion von Ko-/Datentypen mit Unterstüzung für beliebig verschränkte und verschachtelte Definitionen. Letzteres erlaubt Typdefinitionen, die sowohl induktive als auch koinduktive Komponenten enthalten, wie zum Beispiel der Typ der Bäume mit endlichem Verzweigungsgrad, aber potentiell unendlicher Tiefe.

Unsere Konstruktion basiert auf der Kategorientheorie. Die Grundidee ist es Typkonstruktoren als Funktoren auf der Kategorie der Typen zu betrachten. Mit Hilfe von weiteren semantischen Eigenschaften können wir den Abschluss dieser Funktoren unter Komposition und Konstruktion initialer Algebren und finaler Koalgebren. Letztere entsprechen Definitionen von Ko-/Datentypen auf kategorieller Ebene.

Unsere Ideen sind in Isabelle/HOL formalisiert. Zudem wurde eine Automatisierung der Konstruktion von Datentypen implementiert, die anstelle der vorhandenen Implementierung verwendet werden kann.

Acknowledgements

I want to thank Tobias Nipkow for offering the topic of designing a datatype package for a second time after Stefan Berghofer's work. Despite Stefan's results are very satisfactory and have been used in a productive system for a long time, it was worth testing whether a different approach might be similarly successful.

I am much obliged to Andrei Popescu and Jasmin Christian Blanchette. Both of them were very pleasant advisors. Andrei's patience in introducing me to the world of category theory by drawing certain commutative diagrams over and over again should be honoured. Without his ideas and enthusiasm this work would not have been possible. Jasmin's deep knowledge of the Isabelle system resulted in numerous helpful hints on the implementation design. Moreover, a shift between the working hours of Andrei and Jasmin allowed me to ask questions and get immediate answers literally twenty-four-seven.

Christian Urban deserves my gratitude by being an very early tester of this work and provinding helpful comments and examples.

Last but not least, I thank Anna and the rest of my family for their continuous support during my studies.

Contents

1. Introduction

Inductive datatypes are a ubiquitous high-level specification mechanism in functional programming languages. Our particular functional programming language of interest is that of higher-order logic (HOL, Chap. 2). HOL forms the basis of several popular interactive theorem provers, notably HOL4 [GM93], HOL Light [Har96] and Isabelle/HOL [NPW02] where HOL is introduced following the equation

$$\text{HOL} = \text{Functional Programming} + \text{Logic}.$$

Consequently, the semantics of datatypes is established in HOL by characteristic theorems including an induction principle. The simplest way of providing those theorems to the user is to state them as axioms. The drawback of axioms is the potential introduction of inconsistencies. This motivates the LCF philosophy [GMW79]: Theorems are derived within a small inference kernel, reducing the amount of code that must be trusted. HOL-based provers traditionally follow the LCF philosophy and therefore try to omit axiomatizations of high-level specification mechanisms whenever possible.

Instead, high-level specifications are expressed in terms of existing primitive constructs—this is the definitional approach. At the primitive level, a new type is defined by carving out an isomorphic subset from an existing type. The task of a definitional datatype package consists therefore of the following two steps:

1. Construct a set that is isomorphic to the given high-level specification;
2. Prove the characteristic theorems within the inference kernel.

Melham [Mel89] devised such a definitional package already two decades ago. His approach, considerably extended by Gunter [Gun93, Gun94] and simplified by Harrison [Har95], now lies at the heart of the implementations in HOL4, HOL Light and Isabelle/HOL. After implementing the original datatype package in Isabelle following the Melham–Gunter approach, Berghofer and Wenzel [Ber98, BW99] identified three main limitations, the overcoming of which they suggested as challenges for future work:

1. Codatatypes
2. Composition of definitional packages
3. Non-freely generated types

Codatatypes (the coinductive pendant of datatypes) are not covered by the Melham–Gunter approach. Users face an unappealing choice between tedious manual constructions and risky axiomatizations [DHS05]. A solution to 1 could be a monolithic codatatype package. This is antithetical to goal 2. Many applications require a mixture of datatypes and codatatype, as in the following nested-(co)recursive specification of finitely branching trees of possibly infinite depth:

$\textbf{datatype}\ \alpha\ \mathsf{list} = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\alpha\ \mathsf{list})$
$\textbf{codatatype}\ \alpha\ \mathsf{tree_l} = \mathsf{Node}\ \alpha\ ((\alpha\ \mathsf{tree_l})\ \mathsf{list})$

Finally, 3 demands well-behaved non-free structures (e.g., fset—the type of finite sets of elements of α) being available in (co)datatype declarations. In particular formalizations, it might be much more suitable if the Node type constructor from the above example has unordered children:

$\textbf{codatatype}\ \alpha\ \mathsf{tree_l} = \mathsf{Node}\ \alpha\ ((\alpha\ \mathsf{tree_l})\ \mathsf{fset})$

This thesis 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 (Chap. 3), discarding the discussed limitations.

Our framework draws heavily from category theory and cardinality reasoning. We take advantage of the fact that most type constructors are not only operators on the universe of types but also *functors* satisfying additional semantic properties. We call such functors *bounded natural functor* (*BNFs*) and prove their closure under composition, initial algebra and final coalgebra operations (Chap. 4). The latter two are category-theoretical terminology for datatype and codatatype definitions. 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.

Our development is formalized in Isabelle/HOL. 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.

Beyond the formalization, a prototypical package handles definitions of datatypes automatically, while we are proceeding to implement the automation for the codatatype construction.

The theoretical contributions of this thesis were published in the accepted paper [TPB12]. The material form [TPB12] has been included in this thesis with the permission of the coauthors. The chapters 2, 3, 4 and 6 were taken only with minor changes from the paper. The thesis additionally provides descriptions of selected interesting aspects of the framework that appear trivial when describing the solution categorically, but are essential for a working implementation (Chap. 5).

2. Higher-Order Logic (HOL)

In this thesis, by HOL we mean classical higher-order logic with Hilbert choice, the axiom of infinity and ML-style polymorphism. HOL is based on Church's simple type theory [Chu40, And02]. It is the logic of Gordon's original HOL system [GM93] and of its many successors and emulators. To keep the discussion focused on the relevant issues, we depart from tradition and 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.

2.1. Basics

The standard semantics of HOL relies on a universe $\mathscr{U}$ of *types*, ranged over by α, β, γ, which we view as nonempty collections of elements. Membership of an element a in a type α is written $a : \alpha$. The type unit consists of a single element written $()$, bool is the Boolean type, and nat is the type of natural numbers. Fixed elements of types, such as $() : \mathsf{unit}$, are called *constants*. Given α and β, we can form the type $\alpha \to \beta$ of (total) functions from α to β. 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 α and β, respectively. For functions taking n arguments, we generally prefer the curried form $f : \alpha_1 \to \cdots \to \alpha_n \to \beta$ to the tuple form $f : (\alpha_1 \times \cdots \times \alpha_n) \to \beta$.

HOL supports a restrictive, simply typed flavor of set theory. We write $\alpha\ \mathsf{set}$ for the powertype of α, consisting of sets of α elements; it is isomorphic to $\alpha \to \mathsf{bool}$. The *universe set of* α, $\mathsf{U}_\alpha : \alpha\ \mathsf{set}$, is the set consisting of all the elements of α. For notational convenience, we sometimes write α instead of U_α. Given an element $a : \alpha$ and a set $A : \alpha\ \mathsf{set}$, $a \in A$ 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 α and a predicate $\varphi : \alpha \to \mathsf{bool}$, we can form by comprehension the set $\{a : \alpha.\ \varphi\ a\}$ of type $\alpha\ \mathsf{set}$. Russell's paradox is avoided, because elements of $\alpha\ \mathsf{set}$ cannot be elements of α.

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 $(\alpha_1, \alpha_2, \alpha_3)\ \mathsf{F}$ as $(\alpha_2 + \alpha_1) \times (\alpha_3\ \mathsf{set})$. Except for infix operators, type constructor application is written in postfix notation (e.g., $\alpha\ \mathsf{F}$), whereas function application is written in prefix notation (e.g., $f\ a$). Depending on the context, $(\alpha_1, \ldots, \alpha_n)\ \mathsf{F}$ either denotes the application of F to $(\alpha_1, \ldots, \alpha_n)$ or simply indicates that F is an n-ary type constructor. We abbreviate $(\alpha_1, \ldots, \alpha_n)\ \mathsf{F}$ to $\overline{\alpha}\ \mathsf{F}$. Given a binary type constructor $(\alpha_1, \alpha_2)\ \mathsf{F}$ and a fixed type β, $(\_, \beta)\ \mathsf{F}$ denotes the unary type constructor sending an arbitrary type α to $(\alpha, \beta)\ \mathsf{F}$ and similarly for $(\beta, \_)\ \mathsf{F}$.

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

Where Church's simple type theory only offers monomorphic types, HOL features ML-style (rank-1) polymorphism and type inference. Polymorphic constants can be regarded as families of constants indexed by types. For example, the identity function $\mathsf{id} : \alpha \to \alpha$ is defined for any type α and corresponds to a family $(\mathsf{id}_\alpha)_{\alpha \in \mathscr{U}}$. $\mathsf{Id} : (\alpha \times \alpha)\ \mathsf{set}$ 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_α) if needed.

Hilbert choice is represented by the ε-operator. Given a predicate $P : \alpha \to \mathsf{bool}$, the term $\varepsilon x.\ P\ x$ represents an element of type α that makes P true, if there is any and therefore satisfying the axiom $P\ z \Rightarrow P\ (\varepsilon x.\ P\ x)$.

2.2. Expressiveness

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 (not containable in any fixed set). A typical example is the representation of the HOL semantics, which is impossible in HOL 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. Indeed, a large body of mathematics can be expressed adequately in HOL, as witnessed by the extensive library developments in HOL-based provers.

3. Datatypes in HOL

The limitations of HOL mentioned in Sect. 2.2 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 our main concern.

3.1. The Melham–Gunter Approach

Melham's original datatype package [Mel89] is based on a manually defined polymorphic datatype of finite labeled trees, from which simple datatypes are carved out as subsets. Gunter [Gun93] 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 $\alpha\ \mathsf{tree_F}$ nested in the list type constructor in the definition of finite trees:

$$\textbf{datatype}\ \alpha\ \mathsf{tree_F} = \mathsf{Node}\ \alpha\ ((\alpha\ \mathsf{tree_F})\ \mathsf{list})$$

To define such a type, Gunter unfolds the definition of list, resulting in a mutually recursive definition of trees ($\alpha\ \mathsf{tree_F}$) and "lists-of-trees" ($\alpha\ \mathsf{tree_F\_list}$):

$$\begin{array}{l}\textbf{datatype}\ \alpha\ \mathsf{tree_F} = \mathsf{Node}\ \alpha\ (\alpha\ \mathsf{tree_F\_list}) \\ \textbf{and}\ \alpha\ \mathsf{tree_F\_list} = \mathsf{Nil} \mid \mathsf{Cons}\ (\alpha\ \mathsf{tree_F})\ (\alpha\ \mathsf{tree_F\_list})\end{array}$$

Exploiting an isomorphism, the datatype package translates occurrences of $\alpha\ \mathsf{tree_F\_list}$ to $(\alpha\ \mathsf{tree_F})\ \mathsf{list}$, maintaining to a large extent the illusion of nested recursion. Orthogonally, in order to support positive recursion through functions, Gunter [Gun94] extended Melham's labeled trees with infinite branching.

The handling of mutual and nested recursion has several disadvantages, all related to its non-modularity. 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 prohibitive slow for datatypes with many layers of nesting.

3.2. Bringing HOL Closer to Category Theory

Let $\alpha\ \mathsf{F}$ be a unary type constructor. Category theory has elegant devices to define, based on F, the associated datatype and codatatype by solving the equation $\alpha \cong \alpha\ \mathsf{F}$ (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 $\mathscr{U}$ naturally forms a category where the objects are types and the morphisms are functions between types. We are interested in type constructors $(\alpha_1, \ldots, \alpha_n)\ \mathsf{F}$ that are also *functors* on $\mathscr{U}$, 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 $\mathsf{Fmap} : (\alpha_1 \to \beta_1) \to \ldots \to (\alpha_n \to \beta_n) \to \overline{\alpha}\ \mathsf{F} \to \overline{\beta}\ \mathsf{F}$ satisfying

- $\mathsf{Fmap}\ \mathsf{id} = \mathsf{id}$;
- $\mathsf{Fmap}\ (g_1 \circ f_1)\ \ldots\ (g_n \circ f_n) = (\mathsf{Fmap}\ \overline{g}) \circ (\mathsf{Fmap}\ \overline{f})$.

Let us review some basic functors.

Example 1 (Basic functors)

α-constant functor $(\mathsf{C}_\alpha, \mathsf{Cmap}_\alpha)$***:*** *The α-constant functor $(\mathsf{C}_\alpha, \mathsf{Cmap}_\alpha)$ is the nullary functor consisting of the constant type constructor $\mathsf{C}_\alpha = \alpha$ and the constant map function $\mathsf{Cmap}_\alpha = \mathsf{id}$.*

Sum functor $(+, \oplus)$***:*** *$\alpha_1 + \alpha_2$ consists of a copy $\mathsf{Inl}\ a_1$ of each element $a_1 : \alpha_1$ and a copy $\mathsf{Inr}\ a_2$ of each element $a_2 : \alpha_2$. Given $f_1 : \alpha_1 \to \beta$ and $f_2 : \alpha_2 \to \beta$, let $[f_1, f_2] : \alpha_1 + \alpha_2 \to \beta$ be the function sending $\mathsf{Inl}\ a_1$ to $f_1\ a_1$ and $\mathsf{Inr}\ a_2$ to $f_2\ a_2$. Given $f_1 : \alpha_1 \to \beta_1$ and $f_2 : \alpha_2 \to \beta_2$, let $f_1 \oplus f_2 : \alpha_1 + \alpha_2 \to \beta_1 + \beta_2$ be $[\mathsf{Inl} \circ f_1, \mathsf{Inr} \circ f_2]$.*

Product functor $(\times, \otimes)$***:*** *Let $\mathsf{fst} : \alpha_1 \times \alpha_2 \to \alpha_1$ and $\mathsf{snd} : \alpha_1 \times \alpha_2 \to \alpha_2$ denote the two standard projection functions. Given $f_1 : \alpha \to \beta_1$ and $f_2 : \alpha \to \beta_2$, let $\langle f_1, f_2 \rangle : \alpha \to \beta_1 \times \beta_2$ be the function $a \mapsto (f_1\ a, f_2\ a)$. Given $f_1 : \alpha_1 \to \beta_1$ and $f_2 : \alpha_2 \to \beta_2$, let $f_1 \otimes f_2 : \alpha_1 \times \alpha_2 \to \beta_1 \times \beta_2$ be $\langle f_1 \circ \mathsf{fst}, f_2 \circ \mathsf{snd} \rangle$.*

α-Function space functor $(\mathsf{func}_\alpha, \mathsf{comp}_\alpha)$***:*** *Given a type α, let $\beta\ \mathsf{func}_\alpha = \alpha \to \beta$. For all $f : \beta_1 \to \beta_2$, we define $\mathsf{comp}_\alpha\ f : \beta_1\ \mathsf{func}_\alpha \to \beta_2\ \mathsf{func}_\alpha$ as $\mathsf{comp}_\alpha\ f\ g = f \circ g$.*

Powertype functor $(\mathsf{set}, \mathsf{image})$***:*** *The function $\mathsf{image}\ f : \alpha\ \mathsf{set} \to \beta\ \mathsf{set}$ sends each set A to the image of A through the function $f : \alpha \to \beta$.*

k-Powertype functor $(\mathsf{set}_k, \mathsf{image}_k)$***:*** *Given a cardinal k, for all types α, we define the type $\alpha\ \mathsf{set}_k$ by comprehension, carving out from $\alpha\ \mathsf{set}$ only those sets of cardinality $< k$. For all $f : \alpha \to \beta$, we define $\mathsf{image}_k\ f : \alpha\ \mathsf{set}_k \to \beta\ \mathsf{set}_k$ as the restriction and corestriction of $\mathsf{image}\ f$ via the embeddings of $\alpha\ \mathsf{set}_k$ into $\alpha\ \mathsf{set}$ and of $\beta\ \mathsf{set}_k$ into $\beta\ \mathsf{set}$. The definition of $\mathsf{image}_k\ f$ is correct since image does not increase cardinality. For $k = \aleph_0$, we obtain the finite powertype functor, written $(\mathsf{fset}, \mathsf{fimage})$; for the successor of $\aleph_0$, we obtain the countable powertype functor, written $(\mathsf{cset}, \mathsf{cimage})$.*

While specific map functions are heavily used in HOL theories (e.g., map, image), the theorem provers traditionally do not record the functorial structure Fmap 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 α to the type α list of lists of α elements, is categorically given as the initial algebra on the second argument of the binary functor $(\mathsf{F}, \mathsf{Fmap})$, where $(\alpha,\beta)\ \mathsf{F} = \mathsf{unit} + \alpha \times \beta$ and $\mathsf{Fmap}\ f\ g = \mathsf{id} \oplus f \otimes g$. More precisely, there exists a (polymorphic) *folding bijection* $\mathsf{fld} : (\alpha, \alpha\ \mathsf{list})\ \mathsf{F} \to \alpha\ \mathsf{list}$ making $(\mathsf{fld}, \alpha\ \mathsf{list})$ the initial algebra for the unary functor $(\alpha, \_)\ \mathsf{F}$. Here, $\mathsf{fld} = \langle \mathsf{Nil}, \mathsf{Cons} \rangle$, where Nil and Cons are the familiar list operations. The initial algebra property corresponds to the availability of the standard iterator for lists. Then $(\mathsf{list}, \mathsf{map})$ is itself a unary functor.

Finitely branching trees of finite depth

The ability to define lists is hardly a spectacular achievement. It is the *abstract interface* to lists that makes category theory relevant: $(\mathsf{list}, \mathsf{map})$ 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 $\alpha\ \mathsf{tree}_\mathsf{F}$ on the second argument of the functor $(\mathsf{G}, \mathsf{Gmap})$, where $(\alpha,\beta)\ \mathsf{G} = \alpha \times \beta\ \mathsf{list}$ and $\mathsf{Gmap}\ f\ g = f \otimes \mathsf{map}\ g$. The resulting iterator iter has the polymorphic type $(\alpha \times \beta\ \mathsf{list} \to \beta) \to \alpha\ \mathsf{tree}_\mathsf{F} \to \beta$ and its characteristic equation is $\mathsf{iter}\ s \circ \mathsf{fld} = s \circ (\mathsf{id} \otimes \mathsf{map}\ (\mathsf{iter}\ s))$, where fld is the folding bijection associated to $\alpha\ \mathsf{tree}_\mathsf{F}$ (Fig. 3.1). Thus, the "contract" of tree iteration reads as follows: Given tree-like structure on β as the function $s : \alpha \times \beta\ \mathsf{list} \to \beta$ (viewing β as consisting of "abstract trees," featuring an abstract tree constructor s), provide a function $\mathsf{iter}\ s$ such that $\mathsf{iter}\ s\ (\mathsf{fld}\ (a, trl)) = s\ (a, \mathsf{map}\ (\mathsf{iter}\ s)\ trl)$ for all $a : \alpha$ and $trl : (\alpha\ \mathsf{tree}_\mathsf{F})\ \mathsf{list}$. The characteristic equation of iter abstracts away completely from the definition of lists, using instead the map interface for accessing lists, thereby allowing truly modular nesting of recursive types inside recursive definitions of larger types. Moreover, the categorical approach gracefully handles nested recursion through corecursion, as the next examples illustrate.

Finitely branching trees of possibly infinite depth

To define trees of possibly infinite depth, we can take the final coalgebra $\alpha\ \mathsf{tree}_\mathsf{I}$ on the second argument of the functor $(\mathsf{G}, \mathsf{Gmap})$ defined above. The resulting coiterator coiter has polymorphic type $(\beta \to \alpha \times \beta\ \mathsf{list}) \to \beta \to \alpha\ \mathsf{tree}_\mathsf{I}$ and its characteristic equation is $\mathsf{unf} \circ \mathsf{coiter}\ s \cong (\mathsf{id} \otimes \mathsf{map}\ (\mathsf{coiter}\ s)) \circ s$, where unf is the *unfolding bijection* associated to $\alpha\ \mathsf{tree}_\mathsf{I}$ (Fig. 3.2). Normally, we would split unf in two functions as $\mathsf{unf} = \langle \mathsf{lab}, \mathsf{sub} \rangle$, where, for any $tr : \alpha\ \mathsf{tree}_\mathsf{I}$, $\mathsf{lab}\ tr : \alpha$ is the label of the root and $\mathsf{sub}\ tr$ is the list of its subtrees. Then, also splitting any $s : \beta \to \alpha \times \beta\ \mathsf{list}$ similarly to unf in two functions L and C, the contract of tree coiteration reads as follows: Given a tree-like structure on β consisting of functions $L : \beta \to \alpha$ and $C : \beta \to \beta\ \mathsf{list}$, yield a function $\mathsf{coiter}_{\langle C,L \rangle}$ such that $\mathsf{lab}\ (\mathsf{coiter}_{\langle C,L \rangle}\ b) = L\ b$ and $\mathsf{sub}\ (\mathsf{coiter}_{\langle C,L \rangle}\ b) = \mathsf{map}\ \mathsf{coiter}_{\langle C,L \rangle}\ (C\ b)$ 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 sets (or even finite multisets). We can then define $\alpha\ \mathsf{tree}_\mathsf{I}$ as the

$$\begin{tikzcd}
\alpha \times (\alpha\ \mathsf{tree_F})\ \mathsf{list} \arrow[r, "\mathsf{fld}"] \arrow[d, "\mathsf{id} \otimes \mathsf{map}\ (\mathsf{iter}\ s)"'] & \alpha\ \mathsf{tree_F} \arrow[d, "\mathsf{iter}\ s"] \\
\alpha \times \beta\ \mathsf{list} \arrow[r, "s"] & \beta
\end{tikzcd}$$

Figure 3.1.: Iterator for finitely branching trees of finite depth

$$\begin{tikzcd}
\beta \arrow[r, "s"] \arrow[d, "\mathsf{coiter}\ s"'] & \alpha \times \beta\ \mathsf{list} \arrow[d, "\mathsf{id} \otimes \mathsf{map}\ (\mathsf{coiter}\ s)"] \\
\alpha\ \mathsf{tree_I} \arrow[r, "\mathsf{unf}"] & \alpha \times (\alpha\ \mathsf{tree_I})\ \mathsf{list}
\end{tikzcd}$$

Figure 3.2.: Coiterator for finitely branching trees of possibly infinite depth

final coalgebra of the functor $(\mathsf{H}, \mathsf{Hmap})$, where $(\alpha, \beta)\ \mathsf{H} = \alpha \times \beta\ \mathsf{fset}$ and $\mathsf{Hmap}\ f\ g = f \otimes \mathsf{fimage}\ g$.

3.3. Bringing Category Theory Closer to HOL

Next we focus on devising a proper categorical setting to accommodate (co)datatype definitions. Here is the system of constraints for our desired class $\mathscr{K}$ of functors (perhaps with additional structure) on the universe of types:

- C1 $\mathscr{K}$ contains basic functors, including at least the constant, sum, product and function-space functors.
- C2 All functors in $\mathscr{K}$ *admit* both (a) initial algebras and (b) final coalgebras.
- C3 Class $\mathscr{K}$ is *closed under* (a) initial algebras; (b) final coalgebras; and (c) composition.
- C4 The initial algebra and final coalgebra operations over $\mathscr{K}$ are *expressible* in HOL.

In addition to the above nonnegotiable requirements, we formulate a desideratum:

- D $\mathscr{K}$ contains interesting non-free functors, such as the bounded sets and multisets.

Among the basic functors mentioned in C1, constants, $+$ and $\times$ are needed for constructing even simple datatypes, whereas func_α enables infinite branching. The non-free functors mentioned in D further extend (co)datatypes with permutative structures, among which finite sets and multisets are especially useful in computer science formalizations (e.g., semantics of programming languages).

In C3, closure under initial algebras means the following, say, for binary functors $((\alpha, \beta)\ \mathsf{F}, \mathsf{Fmap})$. If we fix an argument, say, the first, then, by C2, for each fixed type α, there exists the initial F-algebra on the second argument, α IF, for which we can define a map operator IFmap. C3 requires that the unary functor (IF, IFmap) be in $\mathscr{K}$. And similarly for closure under final coalgebras.

C4 is required because we are committed to a definitional framework. Otherwise, we could simply postulate the types corresponding to initial and final coalgebras, together with the necessary (co)iterators and their properties.

The literature does not appear to provide a complete solution for the above system of constraints. An obvious candidate, the class of ω-bicontinuous functors [MA86], satisfies C1–C3 but not C4, because the associated limit construction requires a logic that can express infinite type families (e.g., $(\mathsf{unit}\ \mathsf{F}^n)_n$ for the final coalgebra).

Many results from the literature are concerned only with a given type of construction, and only with admissibility (C2), ignoring closure (C3). Rutten's monograph [Rut00] focuses on coalgebras. It describes a general class of functors on sets, namely, those that preserve weak pullbacks and have a set of generators, or, sufficiently, preserve weak pullbacks and are bounded (in that there exists a cardinal upper bound for the coalgebras generated by any singleton in any of their coalgebras). The main issue with this class of functors is admissibility of initial algebras (C2-a). Closure properties (C3), which Rutten omits to discuss, might also be an issue.

Also focusing on coalgebra, Barr [Bar93, Bar94] proves the existence of a final coalgebra for accessible functors on sets (i.e., functors preserving k-filtered colimits for some k). This result is an internalization to sets of Aczel and Mendler's final coalgebra theorem [AM89] stated for set-based functors on classes. Moreover, Barr produces a bound for the size of the final coalgebra, assuming the existence of a certain large cardinal. However, k-filtered colimits are incompatible with C4 for the same reason ω-limit constructions do and internalizing the construction to a sufficiently large type using the provided cardinal bound is also infeasible, because it requires large cardinals whose existence is not provable in HOL or even ZFC. (C2-a and C3 might also be problematic.)

A different result from Barr [Bar93] states that any quotient functor of an ω-bicontinuous functor admits a weakly final coalgebra obtained from any weakly final coalgebra of the latter. A subclass of ω-bicontinuous that admits HOL-expressible (co)datatype constructions could prove to be an answer to C1–C4 via this result. In fact, the class $\mathscr{K}$ which we adopt includes the class $\mathscr{K}'$ of functors F that are quotients of Fbd-function-space functors, with Fbd a cardinal number depending on F. Whether $\mathscr{K}'$ is also a solution to C1–C4 remains for us an open question.

Finally, Hensel and Jacobs [HJ97] propose a modular development of (co)datatypes for datafunctors, a syntactically specified class consisting of all 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 [HJ98], rely on (co)limits.

4. 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 with themselves.

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 is unary), we consider a polymorphic constant $\mathsf{Fset} : \alpha\ \mathsf{F} \to \alpha\ \mathsf{set}$, where $\mathsf{Fset}\ x$ consists of all "atoms" of x; for example, if F is list, Fset returns the set of elements in the list.

We think of the elements x of $\alpha\ \mathsf{F}$ as consisting of a *shape* together with a *content* that fills the shape with elements of α, with $\mathsf{Fset}\ x$ returning this content in flattened format, as a set (Fig. 4.1). This suggests that Fset should be a natural transformation between the functors (F, Fmap) and (set, image) (diagram in Fig. 4.2 commutative for all $f : \alpha \to \beta$). Fset allows us to internalize the type constructor F to sets of elements of given types α. Namely, we define $\mathsf{Fin} : \alpha\ \mathsf{set} \to (\alpha\ \mathsf{F})\ \mathsf{set}$ by $\mathsf{Fin}\ A = \{x : \alpha\ \mathsf{F}.\ \mathsf{Fset}\ x \subseteq A\}$. The generalization to n-ary functors is straightforward, with $\mathsf{Fin}\ A_1\ \ldots\ A_n = \{x : (\alpha_1, \ldots, \alpha_n)\ \mathsf{F}.\ \bigwedge_i \mathsf{Fset}_i\ x \subseteq A_i\}$. In particular, $\mathsf{Fin}\ \alpha_1\ A_2 = \{x : (\alpha_1, \alpha_2)\ \mathsf{F}.\ \mathsf{Fset}_2\ x \subseteq A_2\}$ (where the first occurrence of α_1 abbreviates U_{α_1}).

Combining the map and set operators and suitable cardinal bounds, we obtain the following key notion.

Definition 1 (Bounded natural functor (BNF)) *An n-ary* bounded natural functor *is a tuple* (F, Fmap, Fset, Fbd)*, where*

- F *is an n-ary type constructor,*
- $\mathsf{Fmap} : (\alpha_1 \to \beta_1) \to \cdots \to (\alpha_n \to \beta_n) \to (\alpha_1, \ldots, \alpha_n)\ \mathsf{F} \to (\beta_1, \ldots, \beta_n)\ \mathsf{F}$,
- $\mathsf{Fset}_i : (\alpha_1, \ldots, \alpha_n)\ \mathsf{F} \to \alpha_i\ \mathsf{set}$ *for* $i \in \{1, \ldots, n\}$,

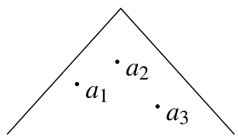

Figure 4.1.: An element x of $\alpha\ \mathsf{F}$ with $\mathsf{Fset}\ x = \{a_1, a_2, a_3\}$

$$\begin{tikzcd} \alpha\,\mathsf{F} \arrow[r, "\mathsf{Fset}"] \arrow[d, "\mathsf{Fmap}\,f"'] & \alpha\,\mathsf{set} \arrow[d, "\mathsf{image}\,f"] \\ \beta\,\mathsf{F} \arrow[r, "\mathsf{Fset}"] & \beta\,\mathsf{set} \end{tikzcd}$$

Figure 4.2.: The "set" natural transformation

- Fbd *is an infinite cardinal number,*

satisfying the following properties for $i \in \{1, \dots, n\}$:

FUNC $(\mathsf{F}, \mathsf{Fmap})$ *is a binary functor.*

NAT$_i$ *For all $\alpha_1, \dots, \alpha_{i-1}, \alpha_{i+1}, \dots, \alpha_n$, Fset_i is a natural transformation between* $((\alpha_1, \dots, \alpha_{i-1}, \_, \alpha_{i+1}, \dots, \alpha_n)\ \mathsf{F}, \mathsf{Fmap})$ *and* $(\mathsf{set}, \mathsf{image})$.

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

CONG *If $\forall a \in \mathsf{Fset}_i\ x.\ f_i\ a = g_i\ a$ for all $i \in \{1, \dots, n\}$, then* $\mathsf{Fmap}\ f_1\ \dots\ f_n\ x = \mathsf{Fmap}\ g_1\ \dots\ g_n\ x$.

CBD *The following cardinal-bound conditions hold:*
a. $\forall x : (\alpha_1, \dots, \alpha_n)\ \mathsf{F}.\ \ |\mathsf{Fset}_i\ x| \leq \mathsf{Fbd}$ *for all $i \in \{1, \dots, n\}$;*
b. $|\mathsf{Fin}\ A_1\ \dots\ A_n| \leq (|A_1| + \dots + |A_n| + 2)^{\mathsf{Fbd}}$.

Among the above conditions, FUNC and NAT$_i$ were already explained and motivated. WP is a technical condition allowing a smooth treatment of bisimilarity relations, relevant for coinduction and corecursion [Rut00]; unlike other (weak) limits, weak pullbacks involve a finite number of types and are hence expressible in HOL. We use a definition of weak pullbacks that restricts the participating functions on given sets. We define the predicate $\mathsf{wpull}\ A\ B_1\ B_2\ f_1\ f_2\ p_1\ p_2$ to hold iff for all $b_1 \in B_1, b_2 \in B_2$ if $f_1\ b_1 = f_2\ b_2$ holds, then there exist an $a \in A$ such that $p_1\ a = b_1 \wedge p_2\ a = b_2$.

Thus, the WP property of an n-ary BNF says that if $\mathsf{wpull}\ A_i\ B_{1i}\ B_{2i}\ f_{1i}\ f_{2i}\ p_{1i}\ p_{2i}$ holds for all $i \in 1 \dots n$, then so does $\mathsf{wpull}\ (\mathsf{Fin}\ A_1\ \dots\ A_n)\ (\mathsf{Fin}\ B_{11}\ \dots\ B_{1n})\ (\mathsf{Fin}\ B_{21}\ \dots\ B_{2n})\ (\mathsf{Fmap}\ f_{11}\ \dots\ f_{1n})\ (\mathsf{Fmap}\ f_{21}\ \dots\ f_{2n})\ (\mathsf{Fmap}\ p_{11}\ \dots\ p_{1n})\ (\mathsf{Fmap}\ p_{21}\ \dots\ p_{2n})$. Our definition is weaker than the standard notion from literature [Rut00], since it does not require p_1, p_2, f_1 and f_2 to form a commutative diagram.

CONG states that $\mathsf{Fmap}\ f_1\ f_2\ x$ is uniquely determined by the action of f_i on the atoms of x, $\mathsf{Fset}_i\ x$—it ensures that Fmap behaves well with respect to Fin. Finally, the cardinality conditions put bounds on the branching (CBD-a) and on the number of elements (CBD-b) of the functor $(\mathsf{F}, \mathsf{Fmap})$ and can be understood in terms of shape and content. Thus, CBD-a states that the F-shapes have no more than Fbd slots for contents. Moreover, CBD-b states that shapes are not too redundant, so that all possible combinations of shape and content do not exceed the number of assignments of contents to slots, $A_1 + A_2 \to \mathsf{Fbd}$. (The $+2$ addition is a technicality that covers the case where $A_1 = A_2 = \varnothing$). We are now ready to state the main theoretical result:

Theorem 1 *The class of BNFs satisfies constraints C1–C4 and desideratum D.*

Proof sketch

We must show that certain basic type constructors form BNFs and that the operations of composition, initial algebra and final coalgebra exist in HOL and have themselves a BNF structure. Sects. 4.1–4.6 below are dedicated to these tasks. □

4.1. Basic Type Constructors

Sect. 3.2 described the basic constructors' map structure. We now present their set structure and cardinal bound, guided by our "shape and content" intuition.

Example 2 (Basic BNFs)

- $\mathsf{F} = \mathsf{C}_\alpha$: $\mathsf{Fset}\ x = \varnothing$; $\mathsf{Fbd} = \aleph_0$.
- $\mathsf{F} = +$: $\mathsf{Fset}_1\ (\mathsf{Inl}\ a_1) = \{a_1\}$, $\mathsf{Fset}_2\ (\mathsf{Inl}\ a_1) = \varnothing$, $\mathsf{Fset}_1\ (\mathsf{Inr}\ a_2) = \varnothing$, $\mathsf{Fset}_2\ (\mathsf{Inr}\ a_2) = \{a_2\}$; $\mathsf{Fbd} = \aleph_0$.
- $\mathsf{F} = \times$: $\mathsf{Fset}_1\ (a_1, a_2) = \{a_1\}$, $\mathsf{Fset}_2\ (a_1, a_2) = \{a_2\}$; $\mathsf{Fbd} = \aleph_0$.
- $\mathsf{F} = \mathsf{func}_\alpha$: $\mathsf{Fset}_1\ g = \mathsf{image}\ g\ \alpha$; $\mathsf{Fbd} = \mathsf{max}\ (|\alpha|\ ,\ \aleph_0)$.
- $\mathsf{F} = \mathsf{set}_k$: $\mathsf{Fset}\ x$ *is the set corresponding to* x *via the embedding of* $\alpha\ \mathsf{set}_k$ *into* $\alpha\ \mathsf{set}$; $\mathsf{Fbd} = \mathsf{max}\ (k,\ \aleph_0)$.

For $\mathsf{F} = \mathsf{set}$ the set structure is clearly $\mathsf{Fset}\ x = x$. Though, set is not a BNF, due to the absence of a proper bound.

4.2. Composition

Although we seldom emphasize its role, composition is a pervasive auxiliary operation in interesting (co)datatype definitions. For example, the list-defining BNF $(\alpha, \beta)\ \mathsf{F}$ discussed in Sect. 3.2 is a composition of basic BNFs ($+$, $\mathsf{C}_{\mathsf{unit}}$ and $\times$).

We describe the process of composing BNFs in extensive detail in Sect. 5.2.

4.3. Relators

A key insight due to Rutten [Rut98] is that, thanks to WP, the functor $(\mathsf{F}, \mathsf{Fmap})$ has a natural extension to a *relator*, i.e., a functor on the category of types and binary relations, denoted $\mathscr{R}$. We can express the relator action of F as a polymorphic constant $\mathsf{Frel} : (\alpha_1 \times \alpha_2)\ \mathsf{set} \to (\beta_1 \times \beta_2)\ \mathsf{set} \to ((\alpha_1, \alpha_2)\ \mathsf{F} \times (\beta_1, \beta_2)\ \mathsf{F})\ \mathsf{set}$ defined as $\mathsf{Frel}\ Q\ R = \{(\mathsf{Fmap}\ \mathsf{fst}\ \mathsf{fst}\ z, \mathsf{Fmap}\ \mathsf{snd}\ \mathsf{snd}\ z).\ z \in \mathsf{Fin}\ Q\ R\}$.

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

- if F is the product functor, $\mathsf{Fpred}\ \varphi_1\ \varphi_2\ (a_1, a_2)\ (b_1, b_2) \Leftrightarrow \varphi_1\ a_1\ b_1 \wedge \varphi_2\ a_2\ b_2$;
- if F is the sum functor, $\mathsf{Fpred}\ \varphi_1\ \varphi_2\ a\ b \Leftrightarrow (\exists a_1\ b_1.\ a = \mathsf{Inl}\ a_1 \wedge b = \mathsf{Inl}\ b_1 \wedge \varphi_1\ a_1\ b_1) \vee (\exists a_2\ b_2.\ a = \mathsf{Inr}\ a_2 \wedge b = \mathsf{Inr}\ b_2 \wedge \varphi_2\ a_2\ b_2)$.

4.4. The Categories of (Co)algebras

For this and the next two sections, we fix a binary BNF $\mathscr{F} = (\mathsf{F}, \mathsf{Fmap}, \mathsf{Fset}, \mathsf{Fbd})$. 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.

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 $\alpha\ \mathsf{IF}$ (or maximal solution $\alpha\ \mathsf{JF}$) of the equation $\alpha \cong (\beta, \alpha)\ \mathsf{F}$. The general constructions involve n $(m+n)$-ary BNFs $\mathscr{F}_i$ with type constructors $(\overline{\beta}, \overline{\alpha})\ \mathsf{F}_i$ where $\overline{\beta} = (\beta_1, \ldots, \beta_m)$ and $\overline{\alpha} = (\alpha_1, \ldots, \alpha_n)$ and yield n m-ary BNFs $\mathscr{IF}_1, \ldots, \mathscr{IF}_n$ (or $\mathscr{JF}_1, \ldots, \mathscr{JF}_n$) with their type constructors of the form $\overline{\beta}\ \mathsf{IF}_i$ (or $\overline{\beta}\ \mathsf{JF}_i$). Some interesting aspects of this general case are sketched in Chap. 5.

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.

Definition 2 ((Co)algebra and morphism) *For a fixed type β, a (β-)*algebra *is a pair $\mathscr{A} = (A, s)$ where:*

- $A : \alpha\ \mathsf{set}$ *is the* carrier set *of $\mathscr{A}$ (and α is the* underlying type *of $\mathscr{A}$),*
- $s : (\beta, \alpha)\ \mathsf{F} \to \alpha$ *is the* structural function *of $\mathscr{A}$,*

such that A is closed under *s, in that $\forall x \in \mathsf{Fin}\,\beta\,A.\ s\ x \in A$ (and thus we may regard s as a function $s : \mathsf{Fin}\,\beta\,A \to A$).*

*Dually, a (β-)*coalgebra *is given by a pair $(A : \alpha\ \mathsf{set}, s : \alpha \to (\beta, \alpha)\ \mathsf{F})$ such that $\forall x \in A.\ s\ x \in \mathsf{Fin}\,\beta\,A$. Algebras form a category where* morphisms *$f : \mathscr{A}_1 = (A_1 : \alpha_1\ \mathsf{set}, s_1) \to \mathscr{A}_2 = (A_2 : \alpha_2\ \mathsf{set}, s_2)$ are functions $f : \alpha_1 \to \alpha_2$ such that the diagram on the left of Fig. 4.3 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.

Definition 3 (Initial algebras/final coalgebras) *An algebra $\mathscr{A}$ is called* initial *if for all algebras $\mathscr{A}'$ there exists a unique morphism $f : \mathscr{A} \to \mathscr{A}'$ 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.*

We are looking for a type constructor $\beta\ \mathsf{IF}$ (dually, $\beta\ \mathsf{JF}$) and function $\mathsf{fld} : (\beta, \beta\ \mathsf{IF}) \to \beta\ \mathsf{IF}$ (dually, $\mathsf{unf} : \beta\ \mathsf{JF} \to (\beta, \beta\ \mathsf{JF})$) such that the algebra $(\beta\ \mathsf{IF}, \mathsf{fld})$ is initial (dually, the coalgebra $(\beta\ \mathsf{JF}, \mathsf{unf})$ is final).

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

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

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

$$\begin{tikzcd}
\mathsf{Fin}\,\beta\,A_1 \arrow[r, "s_1"] \arrow[d, "\mathsf{Fmap\ id}\,f"'] & A_1 \arrow[d, "f"] \\
\mathsf{Fin}\,\beta\,A_2 \arrow[r, "s_2"] & A_2
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{Fin}\,\beta\,A_1 \arrow[d, "\mathsf{Fmap\ id}\,f"'] & A_1 \arrow[l, "s_1"'] \arrow[d, "f"] \\
\mathsf{Fin}\,\beta\,A_2 & A_2 \arrow[l, "s_2"']
\end{tikzcd}$$

Figure 4.3.: Algebra morphism (left) and coalgebra morphism (right)

4.5. Initial Algebra

Initial algebra from weakly initial algebra

Given an algebra $\mathscr{A} = (A, s)$, let M_s be the intersection of all sets B such that (B, s) is an algebra and let $\mathscr{M}(\mathscr{A})$, the *minimal subalgebra* of $\mathscr{A}$, be (M_s, s). It is immediate that there exists at most one morphism from $\mathscr{M}(\mathscr{A})$ to any other algebra. Then, given a weakly initial algebra $\mathscr{C}$, the desired initial β-algebra is its minimal subalgebra, $\mathscr{M}(\mathscr{C})$. Of course, $\mathscr{M}(\mathscr{C})$ depends on β (which was fixed all along). Now $\beta\,\mathsf{IF}$ is introduced by a type definition, carving out the underlying set of $\mathscr{M}(\mathscr{C})$ as a new type and the folding map fld is defined by copying on $\beta\,\mathsf{IF}$ the structural map of $\mathscr{M}(\mathscr{C})$ (so that in effect $(\beta\,\mathsf{IF}, \mathsf{fld})$ becomes isomorphic to $\mathscr{M}(\mathscr{C})$).

Construction of a weakly initial algebra

This relies on a crucial lemma about the cardinality of minimal subalgebras, whose proof employs the BNF cardinality assumptions CBD.

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

Proof: The definition of M_s "from above," as an intersection, is not helpful for establishing a cardinal bound. We need an alternative construction of M_s "from below," as a union. For this, we define the family $(K_i)_{i < \mathsf{Suc\ Fbd}}$ by transfinite recursion as follows:

- $K_i = \bigcup_{j<i} K_j$, if i is a limit ordinal (thus, $K_0 = \varnothing$);
- $K_{i+1} = K_i \cup \{s\ x.\ \mathsf{Fset}_2\ x \subseteq K_i\}$.

Let $K_\infty = \bigcup_{i < \mathsf{Suc\ Fbd}} K_i$. We must prove $M_s = K_\infty$. First, $K_\infty \subseteq M_s$ follows easily by induction on i using that M_s is an algebra. For the harder inclusion $K_\infty \subseteq M_s$, it suffices to show that K_∞ is an algebra. Let x be such that $x \in \mathsf{Fin}\,\beta\,K_\infty$, i.e., $\mathsf{Fset}_2\ x \subseteq K_\infty$. Since $\mathsf{Suc\ Fbd}$ is a regular cardinal and, by CBD-a, $|\mathsf{Fset}_2\ x| < \mathsf{Suc\ Fbd}$, we obtain $i < \mathsf{Suc\ Fbd}$ such that $\mathsf{Fset}_2\ x \subseteq K_i$. Hence $s\ x \in K_{i+1} \subseteq K_\infty$, as desired. It then suffices to show $|K_\infty| \leq (|\beta| + 2)^{\mathsf{Suc\ Fbd}}$. The stronger property $\forall i < \mathsf{Suc\ Fbd}.\ |K_i| \leq (|\beta| + 2)^{\mathsf{Suc\ Fbd}}$ follows by induction on i, via CBD-b and cardinal arithmetic. □

Let Θ be the set of all algebras $\mathscr{A}$ having as underlying type a type γ of sufficiently large cardinality, $(|\beta| + 2)^{\mathsf{Suc\ Fbd}}$; such a type exists and in fact can be taken to be the very underlying type of this cardinal. The desired weakly initial algebra $\mathscr{C}$ is the product of all algebras in Θ. Indeed, by Lemma 2, for any algebra $\mathscr{B}$, its minimal subalgebra

$\mathscr{M}(\mathscr{B})$ is isomorphic to one in Θ, to which $\mathscr{C}$ has a projection morphism. This gives a morphism from $\mathscr{C}$ to $\mathscr{M}(\mathscr{B})$, hence also one from $\mathscr{C}$ to $\mathscr{B}$. We have thus proved:

Prop. 3 $(\beta\,\mathsf{IF}, \mathsf{fld})$ *is the initial β-algebra.*

This yields an iterator $\mathsf{iter} : ((\beta,\alpha)\,\mathsf{F} \to \alpha) \to \beta\,\mathsf{IF} \to \alpha$ such that

$$\mathsf{iter}\ s \circ \mathsf{fld} = s \circ \mathsf{Fmap}\ \mathsf{id}\ (\mathsf{iter}\ s)$$

holds (cf. Fig. 3.1).

Structural induction

The set structure 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 fld is a bijection, for any element $b \in \beta\,\mathsf{IF}$ there is a unique $y \in (\beta, \beta\,\mathsf{IF})\,\mathsf{F}$ such that $b = \mathsf{unf}\ y$—this is an abstract version of case analysis. Then the inductive components of b are precisely the elements of $\mathsf{Fset}_2\ y$ and we have the following induction principle:

Prop. 4 *Let* $\varphi : \beta\ \mathsf{IF} \to \mathsf{bool}$ *be a predicate and assume* $\forall y.\ (\forall b \in \mathsf{Fset}_2\ y.\ \varphi\ b) \Rightarrow \varphi\,(\mathsf{fld}\ y)$. *Then* $\forall b.\ \varphi\ b$.

For $\mathsf{F} = \mathsf{unit} + \beta \times \alpha$ with $\mathsf{IF} = \mathsf{list}$ (Sect. 3.2), the above is equivalent to the familiar induction principle.

BNF structure

It is standard to define a functorial structure for the initial algebra, namely $\mathsf{IFmap}\ f = \mathsf{iter}\ (\mathsf{fld} \circ (\mathsf{Fmap}\ f\ \mathsf{id}))$. As for the set structure, consider $b \in \beta\,\mathsf{IF}$. Intuitively, $\mathsf{IFset}\ b$ should contain all the Fset_1 atoms of b, then the 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 Fset_2. We are led to defining IFset as $\mathsf{iter}\ \mathsf{collect}$, i.e., as the unique function making the Fig. 4.4 diagram commutative, where $\mathsf{collect}\ a = \mathsf{Fset}_1\ a \cup \bigcup \mathsf{Fset}_2\ a$.

Prop. 5 $(\mathsf{IF}, \mathsf{IFmap}, \mathsf{IFset}, 2^{\mathsf{Fbd}})$ *is a BNF.*

As a BNF, IF is also a relator (Sect. 4.3). Importantly for modular reasoning however, we can express IFpred directly in terms of Fpred. Thus, IFpred is uniquely determined by the recursive equations $\mathsf{IFpred}\ \varphi\ (\mathsf{fld}\ x_1)\ (\mathsf{fld}\ x_2) \Leftrightarrow \mathsf{Fpred}\ \varphi\ (\mathsf{IFpred}\ \varphi)\ x_1\ x_2$. For example, for the list functor, the above equation splits in the following, according to the relator structure of the component functors (unit, $+$, and $\times$):

- $\mathsf{list\_pred}\ \varphi\ \mathsf{Nil}\ \mathsf{Nil} \Leftrightarrow \mathsf{True}$,
- $\mathsf{list\_pred}\ \varphi\ \mathsf{Nil}\ (\mathsf{Cons}\ b\ bs) \Leftrightarrow \mathsf{False}$,
- $\mathsf{list\_pred}\ \varphi\ (\mathsf{Cons}\ a\ as)\ \mathsf{Nil} \Leftrightarrow \mathsf{False}$,
- $\mathsf{list\_pred}\ \varphi\ (\mathsf{Cons}\ a\ as)\ (\mathsf{Cons}\ b\ bs) \Leftrightarrow \varphi\ a\ b \wedge \mathsf{list\_pred}\ \varphi\ as\ bs$,

revealing list_pred as the componentwise ordering on lists.

$$
\begin{tikzcd}
(\beta, \beta\ \mathsf{IF})\ \mathsf{F} \arrow[r, "\mathsf{fld}"] \arrow[d, "\mathsf{Fmap\ id\ IFset}"'] & \beta\ \mathsf{IF} \arrow[d, "\mathsf{IFset}"] \\
(\beta, \beta\ \mathsf{set})\ \mathsf{F} \arrow[r, "\mathsf{collect}"] & \beta\ \mathsf{set}
\end{tikzcd}
$$

Figure 4.4.: Set structure for IF

$$
\begin{tikzcd}
A \arrow[d, "s"'] & R \arrow[l, "\mathsf{fst}"'] \arrow[r, "\mathsf{snd}"] \arrow[d, dashed] & A \arrow[d, "s"] & A \arrow[r, leftrightarrow, "R"] \arrow[d, "s"'] & A \arrow[d, "s"] \\
\mathsf{Fin}\,\beta\,A & \mathsf{Fin}\,\beta\,R \arrow[l, "\mathsf{Fmap\ id\ fst}"'] \arrow[r, "\mathsf{Fmap\ id\ snd}"] & \mathsf{Fin}\,\beta\,A & \mathsf{Fin}\,\beta\,A \arrow[r, leftrightarrow, "\mathsf{Frel\ Id\ R}"] & \mathsf{Fin}\,\beta\,A
\end{tikzcd}
$$

Figure 4.5.: Bisimulation

4.6. Final Coalgebra

Final coalgebra from weakly final coalgebra

This follows by the standard coalgebraic theory of bisimulation relations [Rut00]. A bisimulation on a coalgebra $\mathscr{A} = (A, s)$ is a relation $R \subseteq A \times A$ such that $\forall (a, b) \in R.\ \exists z \in \mathsf{Fin}\,\beta\,R.\ \mathsf{Fmap\ id\ fst}\ z = a \wedge \mathsf{Fmap\ id\ fst}\ z = b$, i.e., such that in Fig. 4.5 (left) there exists a function along the dotted arrow making the two diagrams commutative. This abstract concept covers the natural ad hoc notions of bisimulation for concrete functors [Rut00]. A bisimulation R is in effect an endomorphism on A in the types-and-relations category $\mathscr{R}$ such that $(a, b) \in R$ implies $(s\ a, s\ b) \in \mathsf{Frel\ Id}\ R$—Fig. 4.5 (right). Hence composition of bisimulations is a bisimulation and then it follows easily that the largest bisimulation $\mathsf{LB}(\mathscr{A})$ on a coalgebra $\mathscr{A}$ is an equivalence relation and that the resulting quotient coalgebra $\mathscr{A}/_{\mathsf{LB}(\mathscr{A})}$ has the property that any coalgebra has at most one morphism to it.

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

Construction of a weakly final coalgebra

The abstract construction indicated in Rutten [Rut00], 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 next 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 $\mathsf{Suc}_{Kl,kl}$, the set of *Kl-successors of kl*, be $\{kl\ @\ [k].\ kl\ @\ [k] \in Kl\}$, where @ denotes list concatenation and $[k]$ the k-singleton list. We define an

Fbd-*tree* to be a pair (Kl, tr), where $Kl \subseteq \mathsf{Fbd}$ list is prefix closed and $tr : Kl \to \mathsf{Fin}\,\beta\,\mathsf{Fbd}$ is such that $\forall kl \in Kl.\ \mathsf{Fset}_2\ (tr\ kl) = \mathsf{Suc}_{Kl,kl}$. Thus, Fbd-trees are at most Fbd-branching trees labeled as follows: Every node is labeled with an element of $\mathsf{Fin}\,\beta\,\mathsf{Fbd}$ whose set of second-argument atoms consists of precisely the node's emerging branches. Given a tree (Kl, tr), we define $\mathsf{sub}_{(Kl,tr)} : \{k.\ [k] \in Kl\} \to C$ to send each k to the immediate k-subtree of (Kl, tr), more precisely, $\mathsf{sub}_{(Kl,tr)}\ k = (Kl', tr')$, where $Kl' = \{kl'.\ [k] @ kl' \in Kl\}$ and $tr' : Kl' \to \mathsf{Fin}\,\beta\,\mathsf{Fbd}$ is defined by $tr'\ kl' = tr\ ([k] @ kl')$.

The set C of Fbd-trees can be naturally organized as a coalgebra $\mathscr{C} = (C, s)$ defining $s\ (Kl, tr) = \mathsf{Fmap}\ \mathsf{id}\ \mathsf{sub}_{(Kl,tr)}\ (tr\ \mathsf{Nil})$. Thus, $s\ (Kl, tr)$ operates on (Kl, tr)'s root label tr Nil, substituting in its shape the immediate subtrees for the contents. Then $\mathscr{C}$ 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 $b \in \mathsf{Fset}_2\ (t\ a)$ and so on. Thanks to CBD-a, such trees are at most Fbd-branching, hence representable in C. We have thus proved:

Prop. 6 $(\beta\ \mathsf{JF}, \mathsf{unf})$ *is the final* β*-coalgebra.*

This yields a coiterator $\mathsf{coiter} : (\alpha \to (\beta, \alpha)\ \mathsf{F}) \to \alpha \to \beta\ \mathsf{JF}$ such that

$$\mathsf{unf}\ (\mathsf{coiter}\ s) = \mathsf{Fmap}\ \mathsf{id}\ (\mathsf{coiter}\ s) \circ s$$

holds (cf. Fig. 3.2).

Structural coinduction

Since $\mathsf{LB}(\mathscr{C})$ is the greatest bisimulation on $\mathscr{C}$, it follows that Id is the greatest bisimulation on the quotient coalgebra $\mathscr{C}/_{\mathsf{LB}(\mathscr{C})}$. This gives us the following coinduction principle on $(\beta\ \mathsf{JF}, \mathsf{unf})$ (which is a copy of $\mathscr{C}/_{\mathsf{LB}(\mathscr{C})}$): If R is a bisimulation relation, then $R \subseteq \mathsf{Id}$. Viewing bisimilarities via the relator structure (cf. Fig. 4.5, left) and using the predicate notation, we can rephrase the coinduction principle as follows:

Prop. 7 *Let* $\varphi : \beta\ \mathsf{JF} \to \beta\ \mathsf{JF} \to \mathsf{bool}$ *be a binary predicate and assume* $\forall a\ b.\ \varphi\ a\ b \Rightarrow \mathsf{Fpred}\ \mathsf{Eq}\ \varphi\ (\mathsf{unf}\ a)\ (\mathsf{unf}\ b)$ *(where* $\mathsf{Eq} : \beta \to \beta \to \mathsf{bool}$ *is the equality predicate). Then* $\forall a\ b.\ \varphi\ a\ b \Rightarrow a = b$.

BNF structure

Again, the functorial structure of the final coalgebra is standard, namely, $\mathsf{JFmap}\ f = \mathsf{coiter}\ ((\mathsf{Fmap}\ f\ \mathsf{id}) \circ \mathsf{unf})$. Moreover, JFset can be defined by collecting all the Fset_1 results of repeated unfolding, namely $\mathsf{Fset}_1\ a = \bigcup_{i \in \mathsf{nat}} \mathsf{collect}_{i,a}$, where $\mathsf{collect}_{i,a}$ is defined recursively on i as follows:

- $\mathsf{collect}_{0,a} = \varnothing$;
- $\mathsf{collect}_{i+1,a} = \mathsf{Fset}_1\ (\mathsf{unf}\ a) \cup \bigcup\{\mathsf{collect}_{i,b}.\ b \in \mathsf{Fset}_2\ a\}$.

Similarly to IFpred, the relator JFpred can be described in terms of Fpred, by $\mathsf{JFpred}\ \varphi\ a_1\ a_2 \Leftrightarrow \mathsf{Fpred}\ \varphi\ (\mathsf{JFpred}\ \varphi)\ (\mathsf{unf}\ a_1)\ (\mathsf{unf}\ a_2)$.

Prop. 8 $(\mathsf{JF}, \mathsf{JFmap}, \mathsf{JFset}, \mathsf{Fbd}^{\mathsf{Fbd}})$ *is a BNF.*

5. Implementation as a 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.

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 not presented in this thesis but can be found in [PT12].

All proofs that are performed by the package are using specially tailored Isabelle tactics, whose running time is independent of the amount of nesting (unlike for the Melham–Gunter approach).

In the following sections, we describe some interesting aspects of the concrete implementation, that are not apparent in the theoretical description (Chap. 4). Sect. 5.1 introduces the important distinction between *live* and *dead* variables, while Sect. 5.2 describes the task of proving the closure of BNFs under composition. Further, we cover the requirement of proving non-emptiness of newly defined types in HOL (Sect. 5.3) and consider the fixed point construction in their full generality (Sect. 5.4).

5.1. ML-representation of BNFs

Each BNF is represented by an ML record bnf consisting of the polymorphic constants and their properties as proved theorems, stored in Isabelle's theory database [WW07, §4.1]. The basic BNFs for unit, $+$, $\times$, func_α, fset, countable sets and finite multisets are constructed in the user space, as they do not require ML; users can construct and register custom BNFs in the same way.

Type constructors that are defined as BNFs may depend on two kinds of type variables. We refer to the functorial arguments of a type constructor as *live* variables. Those are exactly the type variables that are acted on by the given map function. The action is required to be covariant. All other variable dependencies of a type constructor are *dead*. Thus, the sum type constructor $\alpha_1 + \alpha_2$ has two live and zero dead variables, while the function space type constructor $\beta\ \mathsf{func}_\alpha$ has one live variable (β) and one dead variable (α). We consistently use the prefix notation for live variables and postfix indexed notation for dead variables. In general, the type constructor $(\beta_1, \dots, \beta_n)\ \mathsf{F}_{\alpha_1,\dots,\alpha_m}$ indicates a BNF having n live and m dead variables. The *arity* of a BNF denotes the number of live variables.

The distinction between live and dead type variables is barely visible in the mathematical descriptions, but crucial in HOL. For instance, the type of the cardinal bound may not depend on the live type variables, but only on the dead ones. Otherwise, the (co)algebraic fixed points would depend on variables on which these fixed points are taken, making the construction impossible. This is just another example of walking on a tightrope in HOL.

5.2. From user specifications to BNFs

The first task of a definitional package is to parse the high-level user specification. After abstracting out concrete syntax sugar, the user specification in our case is a system of fixed point equations on types. For example, the declaration of the list datatype corresponds to taking the least fixed point of the equation $\tau = \mathsf{unit} + \alpha \times \tau$.

Before resolving fixed point equations, we need to prove that each right hand side of such an equation is a BNF. In order automate this task, we have identified four simple operations on BNFs: compose, lift, kill and permute. In the following subsections, we describe the contracts of those operations (input, precondition, output) and how they are combined to obtain a BNF from a right hand side of a fixed point equation.

5.2.1. Compose

The compose operation extends the standard functorial composition to respect the set structure and the cardinal bound conditions of a BNF.

INPUTS: n-ary BNF $\mathscr{G} = (\mathsf{G}, \mathsf{Gmap}, \mathsf{Gset}, \mathsf{Gbd})$ and n m-ary BNFs $\mathscr{F}_i = (\mathsf{F}^i, \mathsf{Fmap}^i, \mathsf{Fset}^i, \mathsf{Fbd}^i)$

PRECONDITION: The $\mathscr{F}_i$'s have all the same live variables $\overline{\alpha}$

OUTPUT: m-ary BNF $\mathscr{H} = \mathscr{G} \circ (\mathscr{F}_1, \ldots, \mathscr{F}_n)$ defined as follows:

- $\overline{\alpha}\ \mathsf{H} = (\overline{\alpha}\ \mathsf{F}^1, \ldots, \overline{\alpha}\ \mathsf{F}^n)\ \mathsf{G}$;
- $\mathsf{Hmap}\ f_1\ \ldots\ f_m = \mathsf{Gmap}\ (\mathsf{Fmap}^1\ f_1\ \ldots\ f_m)\ \ldots\ (\mathsf{Fmap}^n\ f_1\ \ldots\ f_m)$;
- $\mathsf{Hset}_j\ y = \bigcup_{i=1}^{n} \left(\bigcup_{x \in \mathsf{Gset}_i\ y} \mathsf{Fset}^i_j\ x \right)$ for $j \in \{1, \ldots, m\}$;
- $\mathsf{Hbd} = \mathsf{Gbd} * (\mathsf{Fbd}^1 + \cdots + \mathsf{Fbd}^n)$.

5.2.2. Lift

The lift operation adds new live variable to a BNF.

INPUTS: Natural number k and n-ary BNF $\mathscr{F} = (\mathsf{F}, \mathsf{Fmap}, \mathsf{Fset}, \mathsf{Fbd})$ with live variables $\alpha_1, \ldots, \alpha_n$

PRECONDITION: $\{\alpha_1, \ldots, \alpha_n\} \cap \{\alpha_{n+1}, \ldots, \alpha_{n+k}\} = \varnothing$

OUTPUT: $(k+n)$-ary BNF $\mathscr{H}$ defined as follows:

- $(\alpha_{n+1}, \ldots, \alpha_{n+k}, \alpha_1, \ldots, \alpha_n)\ \mathsf{H} = (\alpha_1, \ldots, \alpha_n)\ \mathsf{F}$;
- $\mathsf{Hmap}\ f_{n+1}\ \ldots\ f_{n+k}\ f_1\ \ldots\ f_n = \mathsf{Fmap}\ f_1\ \ldots\ f_n$;
- $\mathsf{Hset}_i\ y = \begin{cases} \varnothing & \text{if } 1 \le i \le k \\ \mathsf{Fset}_{i-k}\ y & \text{if } k < i \le n; \end{cases}$
- $\mathsf{Hbd} = \mathsf{Fbd}$.

5.2.3. Kill

The kill operation turns some live variable of a BNF into dead variables.

INPUTS: Natural number k and n-ary BNF $\mathscr{F} = (\mathsf{F}, \mathsf{Fmap}, \mathsf{Fset}, \mathsf{Fbd})$ with live variables $\alpha_1, \ldots, \alpha_n$

PRECONDITION: $k \le n$

OUTPUT: $(n-k)$-ary BNF $\mathscr{H}$ defined as follows:

- $(\alpha_{k+1}, \ldots, \alpha_n)\ \mathsf{H}_{\alpha_1,\ldots,\alpha_k} = (\alpha_1, \ldots, \alpha_n)\ \mathsf{F}$;
- $\mathsf{Hmap}\ f_{k+1}\ \ldots\ f_n = \mathsf{Fmap}\ \underbrace{\mathsf{id}\ \ldots\ \mathsf{id}}_{k}\ f_{k+1}\ \ldots\ f_n$;
- $\mathsf{Hset}_i = \mathsf{Fset}_{i+k}$ for $i \in \{1, \ldots, n-k\}$;
- $\mathsf{Hbd} = \mathsf{Fbd} * (|\mathsf{U}_{\alpha_1}| + \cdots + |\mathsf{U}_{\alpha_k}|)$.[1]

5.2.4. Permute

The permute operation changes the order of live variables of a BNF.

INPUTS: Permutation $\sigma \in \mathfrak{S}_n$ and n-ary BNF $\mathscr{F} = (\mathsf{F}, \mathsf{Fmap}, \mathsf{Fset}, \mathsf{Fbd})$

OUTPUT: n-ary BNF $\mathscr{H}$ defined as follows:

- $(\alpha_1, \ldots, \alpha_n)\ \mathsf{H} = (\alpha_{\sigma(1)}, \ldots, \alpha_{\sigma(n)})\ \mathsf{F}$;
- $\mathsf{Hmap}\ f_1\ \ldots\ f_n = \mathsf{Fmap}\ f_{\sigma(1)}\ \ldots\ f_{\sigma(n)}$;
- $\mathsf{Hset}_i = \mathsf{Fset}_{\sigma(i)}$ for $i \in \{1, \ldots, n\}$;
- $\mathsf{Hbd} = \mathsf{Fbd}$.

5.2.5. Assembling a BNF

Next we show how to assemble a BNF form a given type. Note that the compose operation (Subsect. 5.2.1) assumes the same arities for the BNFs $\mathscr{F}_i$ on the right hand side of the composite, while in general one may need to compose BNFs having different arities m_i. This case is reducible to the above definition of composition via the lift operation. Furthermore, the forgetful kill operation must be used to ensure that dead variables of a BNF do not occur at live positions of an other BNF participating in composition. To

[1] Recall that U_α denotes the universe set of the type α.

keep the operations lift and kill as simple as possible we assume that live variables are sequantially ordered according to their occurrence in the type of the map function and let these operations work on the first live variables according to this ordering. The permute operation is used to change the ordering of live variables.

Further, we allow dead variables to be substituted with any type. All variables that occur in those substitutions are considered dead. The following procedure handles the general case of composition.

INPUT: Type $((\alpha^1_1,\ldots,\alpha^1_{m_1})\ \mathsf{F}^1_{\tau^1_1,\ldots,\tau^1_{d_1}},\ldots,(\alpha^n_1,\ldots,\alpha^n_{m_n})\ \mathsf{F}^n_{\tau^n_1,\ldots,\tau^n_{d_n}})\ \mathsf{G}_{\tau^0_1,\ldots,\tau^0_{d_0}}$

PRECONDITION: $\mathscr{G} = (\mathsf{G}_{\beta^0_1,\ldots,\beta^0_{d_0}}, \mathsf{Gmap}, \mathsf{Gset}, \mathsf{Gbd})$ and $\mathscr{F}_i = (\mathsf{F}^i_{\beta^i_1,\ldots,\beta^i_{d_i}}, \mathsf{Fmap}^i, \mathsf{Fset}^i,$ $\mathsf{Fbd}^i)$ for $i \in \{1,\ldots,n\}$ are BNFs

OUTPUT: BNF $\mathscr{H}$ that corresponds to the input type

PROCEDURE:

1. a) $\delta_1,\ldots,\delta_{k_\delta} \leftarrow$ all variables from $\tau^0_1,\ldots,\tau^0_{d_0},\ldots,\tau^n_1,\ldots,\tau^n_{d_n}$
   b) For all $i \in \{1,\ldots,n\}$
      i. $n_1,\ldots,n_{k_i} \leftarrow$ positions of elements of $\{\delta_1,\ldots,\delta_{k_\delta}\} \cap \{\alpha^i_1,\ldots,\alpha^i_{m_i}\}$ in $\alpha^i_1,\ldots,\alpha^i_{m_i}$
      ii. pick $\sigma \in \mathfrak{S}_{m_i}$ such that $\sigma(j) = n_j$ for $j \in \{1,\ldots,k_i\}$
      iii. $\overline{\mathscr{F}^i} \leftarrow \mathsf{kill}\ k_i\ (\mathsf{permute}\ \sigma\ \mathscr{F}^i)$
2. a) $\gamma_1,\ldots,\gamma_{k_\gamma} \leftarrow$ all live variables from $\overline{\mathscr{F}^1},\ldots,\overline{\mathscr{F}^n}$
   b) For all $i \in \{1,\ldots,n\}$
      i. $n_1,\ldots,n_{k_i} \leftarrow$ positions of live variables of $\overline{\mathscr{F}^i}$ in $\gamma_1,\ldots,\gamma_{k_\gamma}$
      ii. pick $\sigma \in \mathfrak{S}_{k_\gamma}$ such that $\sigma(n_j) = j$ for $j \in \{k_i+1,\ldots,k_\gamma\}$
      iii. $\overline{\overline{\mathscr{F}^i}} \leftarrow \mathsf{permute}\ \sigma\ (\mathsf{lift}\ (k_\gamma - k_i)\ \overline{\mathscr{F}^i})$
3. $\overline{\mathscr{H}} \leftarrow \mathsf{compose}\ \mathscr{G}\ \overline{\overline{\mathscr{F}^1}}\ \ldots\ \overline{\overline{\mathscr{F}^n}}$
4. $\mathscr{H} \leftarrow$ apply substitution $\{\beta^i_j \mapsto \tau^i_j \mid i \in \{0,\ldots,n\}, j \in \{1,\ldots,d_i\}\}$ to $\overline{\mathscr{H}}$

To obtain a proof that a type is BNF we simply traverse the type recursively, applying the above procedure in postorder. The base cases of this recursion are handled by basic BNFs.

Example 3 *We demonstrate the general composition of BNFs on the input type $\alpha \to \beta + \alpha \times \gamma$ or using the notation from the procedure description $(\beta\ \mathsf{F}^1_\alpha, (\alpha,\beta)\ \mathsf{F}^2)\ \mathsf{G}$ where $\mathsf{F}^1 = \to$, $\mathsf{F}^2 = \times$ and $\mathsf{G} = +$. First the procedure collects all dead variables that are participating in composition. In our example this is only α. There is another live occurence of α, which is transformed into a dead occurence by step 1b. Then, all remaining live variables β,γ are collected and the BNFs F^1 and F^2 are lifted to have exactly those variables as their live variables by step 2b. Finally, we have reduced the original task of composition such that it can be handled by the simple* compose *operation (step 3). The last step is not required in our example since there are no complex types that are substituting the dead variables of the given BNFs.*

5.3. Non-Emptiness Witnesses for Datatypes

New types are defined in HOL via a representing non-empty set. While this is not an issue for codatatypes, as final coalgebras are always non-empty, an initial algebra might be empty. For instance, initial algebras of the product BNF for both variables are empty. In other words, the minimal solution of the fixed point equation $\alpha = \alpha \times \beta$ (or $\beta = \alpha \times \beta$) is $\alpha = \varnothing$ (or $\beta = \varnothing$).

Hence, we need to extend the BNF structure with a facility that allows to distinguish between BNFs with empty and non-empty initial algebras and actually prove non-emptiness in the latter case. We achieve this by maintaining additional constants—*non-emptiness witnesses*—with certain proved properties about them.

Definition 4 (Non-emptiness witness) *A* non-emptiness witness *for an n-ary BNF* $\mathscr{F} = (\mathsf{F}, \mathsf{Fmap}, \mathsf{Fset}, \mathsf{Fbd})$ *is a constant* $\mathsf{Fwit}_{[i_1,\ldots,i_k]}$ *of type* $\alpha_{i_1} \to \cdots \to \alpha_{i_k} \to (\alpha_1, \ldots \alpha_n)\mathsf{F}$ *such that* $I = \{i_1, \ldots, i_k\}$ *is a subset of* $\{1, \ldots, n\}$ *and for all* $j \in \{1, \ldots, n\}$ *it holds*

$$\begin{array}{ll} \mathsf{Fset}_j\ (\mathsf{Fwit}\ a_{i_1}\ \ldots\ a_{i_k}) \subseteq \{a_j\} & \text{if } j \in I, \\ \mathsf{Fset}_j\ (\mathsf{Fwit}\ a_{i_1}\ \ldots\ a_{i_k}) = \varnothing & \text{if } j \notin I. \end{array}$$

We say $\mathsf{Fwit}_{[i_1,\ldots,i_k]}$ depends *on variables in I. We write* Fwit *instead of* $\mathsf{Fwit}_{[]}$*, if I is empty.*

The intuition behind this definition is: If for all j a_{i_j} is an algebra element, then so is the witness $\mathsf{Fwit}\ a_{i_1}\ \ldots\ a_{i_k}$. We will sketch how this helps in proving non-emptiness later in this section, but first we define the witnesses for basic type constructors.

Example 4 (Non-emptiness witnesses for the basic BNFs)

- $\mathsf{F} = \mathsf{C}_\alpha$*:* $\mathsf{Fwit} = \varepsilon a.\ a \in \mathsf{U}_\alpha$.
- $\mathsf{F} = +$*:* $\mathsf{Fwit}_{[1]} = \mathsf{Inl}$, $\mathsf{Fwit}_{[2]} = \mathsf{Inr}$.
- $\mathsf{F} = \times$*:* $\mathsf{Fwit}_{[1,2]}\ a\ b = (a, b)$.
- $\mathsf{F} = \mathsf{func}_\alpha$*:* $\mathsf{Fwit}_{[1]}\ b = \lambda a.\ b$.
- $\mathsf{F} = \mathsf{set}_k$*:* $\mathsf{Fwit} = \varnothing$

Non-emptiness witnesses integrate nicely with the simple BNF operations. In particular, their properties are preserved through composition, which corresponds to “plugging” the witnesses of $\mathscr{F}^i$ into the witnesses of $\mathscr{G}$ (of course with respect to types).

The witnesses enable us to prove non-emptiness of arbitrary algebras. We illustrate this on the example of the list generating fixed point equation $\alpha = \mathsf{unit} + \beta \times \alpha$.

Lemma 9 *Let* $\mathscr{A} = (A, s)$ *be a* β*-algebra for the type constructor* $(\beta, \alpha)\ \mathsf{F} = \mathsf{unit} + \beta \times \alpha$*. Then* $A \neq \varnothing$*.*

Proof:

We show the existence of an element $x \in \mathsf{Fin}\ \mathsf{U}_\beta\ A$ constructively. Then, $s\ x \in A \neq \varnothing$ follows by the definition of an algebra.

An element x of type $\mathsf{unit} + \beta \times \alpha$ is in $\mathsf{Fin}\ \mathsf{U}_\beta\ A$ if and only if $\mathsf{Fset}_2\ x \subseteq A$. For this concrete example it is easy to see that $\mathsf{Fset}_2 = [\lambda z.\ \varnothing, \lambda z.\ \{\mathsf{snd}\ z\}]$ clearly indicating what the x should be, but we should not rely on such ad hoc observations in general. Instead we use the composed witnesses for $(\beta, \alpha)\ \mathsf{F}$. Those are the following two: $\mathsf{Fwit} = \mathsf{Inl}\ (\varepsilon a.\ a \in \mathsf{U}_{\mathsf{unit}})$ and $\mathsf{Fwit}_{[1,2]}\ a\ b = \mathsf{Inr}\ (a, b)$. By the witness properties we have $\mathsf{Fset}_1\ \mathsf{Fwit} = \mathsf{Fset}_2\ \mathsf{Fwit} = \varnothing$, $\mathsf{Fset}_1\ \big(\mathsf{Fwit}_{[1,2]}\ a\ b\big) \subseteq \{a\}$ and $\mathsf{Fset}_2\ \big(\mathsf{Fwit}_{[1,2]}\ a\ b\big) \subseteq \{b\}$. The first equation already serves its purpose to prove $\mathsf{Fset}_2\ \mathsf{Fwit} = \varnothing \subseteq A$ and therefore $\mathsf{Fwit} \in \mathsf{Fin}\ \mathsf{U}_\beta\ A$. □

Keeping all constructable witnesses is overly redundant. For instance, if we have two witnesses $\mathsf{Fwit}_{[i_1,\ldots,i_k]}$ and $\mathsf{Fwit}_{[j_1,\ldots,j_l]}$ such that $\{i_1, \ldots, i_k\} \subseteq \{j_1, \ldots, j_l\}$, we will always prefer $\mathsf{Fwit}_{[i_1,\ldots,i_k]}$ in proofs of algebras being non-empty. Using this observations we can define a partial order on witnesses. Keeping the set of witnesses of a BNF minimized with respect to this partial order is sufficient for the purpose of proving non-emptiness of algebras.

5.4. Fixed Point Operations

In the general case, to define mutually recursive (co)datatypes we need to resolve a system of n equations:

$$\begin{array}{rcl} \alpha_{i_1} & = & (\alpha_1, \ldots, \alpha_{m_{i_1}+n})\ \mathsf{F}^1 \\ & \ldots & \\ \alpha_{i_n} & = & (\alpha_1, \ldots, \alpha_{m_{i_n}+n})\ \mathsf{F}^n \end{array}$$

with $i_1, \ldots, i_n$ all distinct.

Just as in the general case of composition, in order to automate this task, it is easier to work in a normalized setting. Therefore, we assume that all involved BNFs $\mathscr{F}^i$ share the same variables and the fixed point variables (we call them *active*) are the last n live variables:

$$\begin{array}{rcl} \alpha_{m+1} & = & (\alpha_1, \ldots, \alpha_{m+n})\ \mathsf{F}^1 \\ & \ldots & \\ \alpha_{m+n} & = & (\alpha_1, \ldots, \alpha_{m+n})\ \mathsf{F}^n \end{array}$$

Consequently, the first m live variable are called *passive*. We say the variable α_{m+j} is *associated* with the BNF F^j.

Note that the normalized setting is obtained from the original one using the steps 1 and 2 of the procedure in Subsect. 5.2.5.

5.4.1. Least Fixed Point

The package defines the notions of algebras and morphisms dependent on this fixed normalized setting. This must happen each time during a datatype definition, as e.g., the type of an algebra predicate constant depends on m and n, which is not representable with the simple types of HOL in the general case.

To check whether the given specification constitutes a valid datatype we need to prove non-emptiness of the defined algebras. In the proof that we have shown in the case of the list-defining BNF (Lemma 9), a witness not depending on any variable arises from composition. Such witnesses or more generally witnesses that depend only on a subset of the passive live variables lead directly to a proof of non-emptiness, and are therefore referred to as *direct witnesses*. In the general case some of the given BNFs might not have a direct witness. For instance, if none of the BNFs has direct witnesses, the datatype specification is not valid. *Indirect witnesses* (i.e., witnesses depending on active live variables) must be composed with other witnesses of the BNFs associated with those dependencies. We use the following procedure to check if the known witnesses provide a proof of non-emptiness.

INPUT: Sets $\mathscr{I}^j = \{I_1^j, \ldots, I_{k_j}^j\}$ for $j \in \{1, \ldots, n\}$ such that for all $k \in \{1, \ldots, k_j\}$, it holds $\{i_1^j, \ldots, i_{l_k}^j\} = I_k^j \subseteq \{1, \ldots, m+n\}$ and there is a non-emptiness witness $\mathsf{Fwit}^j_{[i_1^j, \ldots, i_{l_k}^j]}$ for F^j

RECURSIVE DEFINITION:

$$\begin{array}{lcl} X_0 & = & \varnothing \\ X_{i+1} & = & X_i \cup \{j.\ \exists k \in \{1, \ldots, k_j\}.\ I_k^j \subseteq \{1, \ldots, m\} \cup \{m+j.\ j \in X_i\}\} \end{array}$$

OUTPUT: X_n

Each j that is contained in X_n was included at some iteration justified by the existence of a witness $\mathsf{Fwit}^j_{I_k^j}$. We call those witnesses the *right* witnesses for j. Intuitively, X_1 represents those BNFs, that have direct right witnesses, X_2 adds those that needs to be composed with right witnesses of X_1, X_3 adds those that needs to be composed with right witnesses of X_2 and so on. If the system of fixed point equations yields a valid datatype, all BNFs should be represented in our set after at most n iterations. In other words, $X_n = \{1, \ldots, n\}$ should hold. The proof of non-emptiness of algebras then easily follows by usage of the additionally stored right witnesses.

The minimal algebras K_∞^j for the mutual case ($j \in \{1, \ldots, n\}$) are defined from below.

Definition 5 (K_∞^j) *Given n BNFs $\mathscr{F}^j$ and n structural maps s_j, we define simultaneously n families $(K_i^j)_{i < \mathsf{Suc\ Fbd}}$ by transfinite recursion as follows for $j \in \{1, \ldots, n\}$:*

- $K_i^j = \bigcup_{i' < i} K_{i'}^j$, *if i is a limit ordinal (thus, $K_0^j = \varnothing$);*
- $K_{i+1}^j = K_i^j \cup \{s_j\ x.\ x \in \mathsf{Fin}^j\ \alpha_1\ \ldots\ \alpha_m\ K_i^1\ \ldots\ K_i^n\}$.

Then, we define K_∞^j as $\bigcup_{i < \mathsf{Suc\ Fbd}} K_i^j$.

Further, the package proves the minimal algebras M_{s_j} being equal to K_∞^j as in the proof of Lemma 2 (Sect. 4.5), defines initial algebras, registers their carrier sets as the new types IF^j and their structural maps as the fld^j-operations. From the fact that the algebras $(\mathsf{IF}^j, \mathsf{fld}^j)$ are initial, characteristic theorems (including the following induction rule) are defined.

$$\begin{tikzcd}
(\overline{\alpha}, \overline{\alpha}\,\mathsf{IF}^1,\ \ldots, \overline{\alpha}\,\mathsf{IF}^n)\,\mathsf{F}^j \arrow[r, "\mathsf{fld}^j"] \arrow[d, "\mathsf{Fmap}^j\ \mathsf{id} \ldots \mathsf{id}\ (\mathsf{iter}^1\ s_1 \ldots s_n) \ldots (\mathsf{iter}^n\ s_1 \ldots s_n)"'] & \overline{\alpha}\,\mathsf{IF}^j \arrow[d, "\mathsf{iter}^j\ s_1 \ldots s_n"] \\
(\overline{\alpha}, \beta_1, \ldots, \beta_n)\,\mathsf{F}^j \arrow[r, "s_j"] & \beta_j
\end{tikzcd}$$

Figure 5.1.: Iterator for the initial algebra $\overline{\alpha}\,\mathsf{IF}^j$

$$\begin{tikzcd}
(\overline{\alpha}, \overline{\alpha}\,\mathsf{IF}^1,\ \ldots, \overline{\alpha}\,\mathsf{IF}^n)\,\mathsf{F}^j \arrow[r, "\mathsf{fld}^j"] \arrow[d, "\mathsf{Fmap}^j\ \mathsf{id} \ldots \mathsf{id}\ \langle \mathsf{id}, \mathsf{rec}^1\ s_1 \ldots s_n\rangle \ldots \langle \mathsf{id}, \mathsf{rec}^n\ s_1 \ldots s_n\rangle"'] & \overline{\alpha}\,\mathsf{IF}^j \arrow[d, "\mathsf{rec}^j\ s_1 \ldots s_n"] \\
(\overline{\alpha}, \overline{\alpha}\,\mathsf{IF}^1 \times \beta_1, \ldots, \overline{\alpha}\,\mathsf{IF}^n \times \beta_n)\,\mathsf{F}^j \arrow[r, "s_j"] & \beta_j
\end{tikzcd}$$

Figure 5.2.: Recursor for the initial algebra $\overline{\alpha}\,\mathsf{IF}^j$

Theorem 10 **(fld-induction)** *Let* $\varphi_1 : \overline{\alpha}\,\mathsf{IF}^1 \to \mathsf{bool}, \ldots, \varphi_n : \overline{\alpha}\,\mathsf{IF}^n \to \mathsf{bool}$ *be predicates and assume* $\forall y.\ (\bigwedge_{k=1}^{n} \forall b \in \mathsf{Fset}^j_{m+k}\ y.\ \varphi_k\ b) \Rightarrow \varphi_j\ (\mathsf{fld}^j\ y)$ *for all* $j \in \{1, \ldots, n\}$*. Then* $\forall b_1, \ldots, b_n.\ \varphi_1\ b_1\ \wedge\ \ldots\ \wedge\ \varphi_n\ b_n$*.*

Additionally, iterator and recursor constants (iter^j and rec^j) are defined in a way such that the diagrams 5.1 and 5.2 are commutative for all $j \in \{1, \ldots, n\}$.

Recursion is a more powerful definition principle than iteration, allowing, at recursion time, the consideration of not only elements of the target type (i.e., results computed so far), but also the original values of the source type. For example, the predecessor function on natural numbers cannot be defined by iteration without introducing auxiliary arguments, but it is definable by a trivial recursion.

All this infrastructure is not only presented to the user, but also heavily used in proving that all IF^j's can themselves be endowed with a BNF-structure.

Theorem 11 $\mathscr{IF}_j = (\mathsf{IF}^j, \mathsf{IFmap}^j, \mathsf{IFset}^j, \mathsf{IFbd})$ *defined by:*

- $\mathsf{IFmap}^j\ f_1\ \ldots\ f_m =$
  $\mathsf{iter}^j\ \big(\mathsf{fld}^1 \circ \mathsf{Fmap}^1\ f_1\ \ldots\ f_m\ \mathsf{id}\ \ldots\ \mathsf{id}\big) \ldots (\mathsf{fld}^n \circ \mathsf{Fmap}^n\ f_1\ \ldots\ f_m\ \mathsf{id}\ \ldots\ \mathsf{id})$*;*
- $\mathsf{IFset}^j_i =$
  $\mathsf{iter}^j \left(\lambda z.\ \mathsf{Fset}^1_i\ z \cup \bigcup_{k=m+1}^{m+n} \bigcup \mathsf{Fset}^1_k\ z\right) \ldots \left(\lambda z.\ \mathsf{Fset}^n_i\ z \cup \bigcup_{k=m+1}^{m+n} \bigcup \mathsf{Fset}^n_k\ z\right)$
  for $i \in \{1, \ldots, m\}$*;*
- $\mathsf{IFbd} = 2^{\max\{\mathsf{Fbd}^1, \ldots, \mathsf{Fbd}^n\}}$.

is a BNF for all $j \in \{1, \ldots, n\}$*.*

5.4.2. Greatest Fixed Point

Dually to the least fixed point construction, the package defines notions of coalgebras, morphisms and bisimulations locally with respect to the fixed setting. The concrete tree coalgebra construction from Sect. 4.6 can be easily extended to the general mutual case. The final coalgebras $(\mathsf{JF}^j, \mathsf{unf}^j)$ are then the quotients of the tree coalgebras to the greatest bisimulation. Characteristic theorems (including the following coinduction rule) are derived from the finality properties.

Theorem 12 (unf**-coinduction)** *Let* $\varphi_1 : \overline{\alpha}\ \mathsf{JF}^1 \to \overline{\alpha}\ \mathsf{JF}^1 \to \mathsf{bool}, \ldots, \varphi_n : \overline{\alpha}\ \mathsf{JF}^n \to \overline{\alpha}\ \mathsf{JF}^n \to \mathsf{bool}$ *be binary predicates and* $x_1, y_1 : \mathsf{JF}^1, \ldots, x_n, y_n : \mathsf{JF}^n$ *such that* $\varphi_1\ x_1\ y_1 \wedge \cdots \wedge \varphi_n\ x_n\ y_n$ *holds. Further, assume:*

$$\forall x, y.\, \varphi_j\ x\ y \Rightarrow \left(\exists z. \begin{array}{ll} \mathsf{Fmap}^j\ \mathsf{id} \ldots \mathsf{id}\ \mathsf{fst} \ldots \mathsf{fst}\ z = \mathsf{unf}^j\ x & \wedge \\ \mathsf{Fmap}^j\ \mathsf{id} \ldots \mathsf{id}\ \mathsf{snd} \ldots \mathsf{snd}\ z = \mathsf{unf}^j\ y & \wedge \\ \bigwedge_{k=1}^{n} \forall (a, b) \in \mathsf{Fset}^j_{m+k}\ z.\ \varphi_k\ a\ b & \end{array} \right)$$

for all $j \in \{1, \ldots, n\}$. *Then* $x_1 = y_1 \wedge \cdots \wedge x_n = y_n$.

Exploiting the relator structure of the F^j's, we can express coinduction more compactly in terms of Fpred^j and the binary equality predicate Eq.

Corollary 13 (Fpred**-coinduction)** *Let* $\varphi_1 : \overline{\alpha}\ \mathsf{JF}^1 \to \overline{\alpha}\ \mathsf{JF}^1 \to \mathsf{bool}, \ldots, \varphi_n : \overline{\alpha}\ \mathsf{JF}^n \to \overline{\alpha}\ \mathsf{JF}^n \to \mathsf{bool}$ *be binary predicates and* $x_1, y_1 : \mathsf{JF}^1, \ldots, x_n, y_n : \mathsf{JF}^n$ *such that* $\varphi_1\ x_1\ y_1 \wedge \cdots \wedge \varphi_n\ x_n\ y_n$ *holds. Further, assume:*

$$\forall x, y.\, \varphi_j\ x\ y \Rightarrow \mathsf{Fpred}^j\ \mathsf{Eq}\ \ldots\ \mathsf{Eq}\ \varphi_1\ \ldots\ \varphi_n\ (\mathsf{unf}^j\ x)\ (\mathsf{unf}^j\ y)$$

for all $j \in \{1, \ldots, n\}$. *Then* $x_1 = y_1 \wedge \cdots \wedge x_n = y_n$.

Coinduction "up to equality" is a syntactic strengthening of the raw coinduction principle of Fpred-coinduction that reduces the coinduction proof task to disjunction with equality (we write '$|$' for disjunction of binary predicates).

Corollary 14 (Fpred**-"up-to"-coinduction)** *Let* $\varphi_1 : \overline{\alpha}\ \mathsf{JF}^1 \to \overline{\alpha}\ \mathsf{JF}^1 \to \mathsf{bool}, \ldots, \varphi_n : \overline{\alpha}\ \mathsf{JF}^n \to \overline{\alpha}\ \mathsf{JF}^n \to \mathsf{bool}$ *be binary predicates and* $x_1, y_1 : \mathsf{JF}^1, \ldots, x_n, y_n : \mathsf{JF}^n$ *such that* $\varphi_1\ x_1\ y_1 \wedge \cdots \wedge \varphi_n\ x_n\ y_n$ *holds. Further, assume:*

$$\forall x, y.\, \varphi_j\ x\ y \Rightarrow \mathsf{Fpred}^j\ \mathsf{Eq}\ \ldots\ \mathsf{Eq}\ (\varphi_1 \mid \mathsf{Eq})\ \ldots\ (\varphi_n \mid \mathsf{Eq})\ (\mathsf{unf}^j\ x)\ (\mathsf{unf}^j\ y)$$

for all $j \in \{1, \ldots, n\}$. *Then* $x_1 = y_1 \wedge \cdots \wedge x_n = y_n$.

The coiterator and corecursor constants are defined making the diagrams 5.3 and 5.4 commutative.

As the final step, the package defines the BNF structure for JF^j as follows and proves the BNF properties using coinduction and other characteristic theorems.

Theorem 15 $\mathscr{JF}_j = (\mathsf{JF}^j, \mathsf{JFmap}^j, \mathsf{JFset}^j, \mathsf{JFbd})$ *defined by:*

$$\begin{array}{ccc}
\beta_j & \xrightarrow{\quad s_j \quad} & (\overline{\alpha}, \beta_1, \ldots, \beta_n)\ \mathsf{F}^j \\
{\scriptstyle \mathsf{coiter}^j\ s_1\ \ldots\ s_n}\downarrow & & \downarrow{\scriptstyle \mathsf{Fmap}^j\ \mathsf{id}\ \ldots\ \mathsf{id}\ (\mathsf{coiter}^1\ s_1\ \ldots\ s_n)\ \ldots\ (\mathsf{coiter}^n\ s_1\ \ldots\ s_n)} \\
\overline{\alpha}\ \mathsf{JF}^j & \xrightarrow{\quad \mathsf{unf}^j \quad} & (\overline{\alpha}, \overline{\alpha}\ \mathsf{JF}^1,\ \ldots, \overline{\alpha}\ \mathsf{JF}^n)\ \mathsf{F}^j
\end{array}$$

Figure 5.3.: Coiterator for the final coalgebra $\overline{\alpha}\ \mathsf{JF}^j$

$$\begin{array}{ccc}
\beta_j & \xrightarrow{\quad s_j \quad} & (\overline{\alpha}, \overline{\alpha}\ \mathsf{JF}^1 + \beta_1, \ldots, \overline{\alpha}\ \mathsf{JF}^n + \beta_n)\ \mathsf{F}^j \\
{\scriptstyle \mathsf{corec}^j\ s_1\ \ldots\ s_n}\downarrow & & \downarrow{\scriptstyle \mathsf{Fmap}^j\ \mathsf{id}\ \ldots\ \mathsf{id}\ [\mathsf{id}, \mathsf{corec}^1\ s_1\ \ldots\ s_n]\ \ldots\ [\mathsf{id}, \mathsf{corec}^n\ s_1\ \ldots\ s_n]} \\
\overline{\alpha}\ \mathsf{JF}^j & \xrightarrow{\quad \mathsf{unf}^j \quad} & (\overline{\alpha}, \overline{\alpha}\ \mathsf{JF}^1,\ \ldots, \overline{\alpha}\ \mathsf{JF}^n)\ \mathsf{F}^j
\end{array}$$

Figure 5.4.: Corecursor for the final coalgebra $\overline{\alpha}\ \mathsf{JF}^j$

- $\mathsf{JFmap}^j\ f_1\ \ldots\ f_m =$
  $\mathsf{coiter}^j\ \left(\mathsf{Fmap}^1\ f_1\ \ldots\ f_m\ \mathsf{id}\ \ldots\ \mathsf{id}\ \circ\ \mathsf{unf}^1\right)\ \ldots\ (\mathsf{Fmap}^n\ f_1\ \ldots\ f_m\ \mathsf{id}\ \ldots\ \mathsf{id}\ \circ\ \mathsf{unf}^n)$*;*
- $\mathsf{JFset}_i^j\ a = \bigcup_{k \in \mathsf{nat}} \mathsf{collect}_i^j\ a\ k$ *for* $i \in \{1, \ldots, m\}$ *where, for each i, the family* $\left(\mathsf{collect}_i^j\right)_{j \in \{1,\ldots,n\}}$ *is defined mutually by recursion on* nat*:*
  $$\mathsf{collect}_i^j\ a\ 0 = \varnothing$$
  $$\mathsf{collect}_i^j\ a\ (k+1) = \mathsf{Fset}_i^j\ (\mathsf{unf}^j\ a) \cup \bigcup_{l=m+1}^{m+n} \bigcup_{b \in \mathsf{Fset}_l^j(\mathsf{unf}^j\ a)} \mathsf{collect}_i^{l-m}\ b\ k;$$
- $\mathsf{JFbd} = \left(\max\{\mathsf{Fbd}^1, \ldots, \mathsf{Fbd}^n\}\right)^{\max\{\mathsf{Fbd}^1,\ldots,\mathsf{Fbd}^n\}}$.

is a BNF for all $j \in \{1, \ldots, n\}$.

5.4.3. Transferring the Non-Emptiness Witnesses along Fixed Point Constructions

Theorems 11 and 15 show that our package is modular, i.e., the (co)datatypes defined by greatest/least fixed point operations can be used in further (co)datatype definitions. However, the modularity statement is not complete without the consideration of non-emptiness witnesses. For instance, if we try to define a datatype (of unlabeled finitely branching trees) as the least fixed point of $\alpha = \alpha$ list,[2] the package will not succeed in proving non-emptiness of the new type unless it knows some non-emptiness witnesses for the list BNF. To obtain such a witness, we need to transport the information that

[2]where α list was itself defined as the least fixed point of $\beta = \mathsf{unit} + \alpha \times \beta$

we already have for the list-defining BNF $\mathsf{unit} + \alpha \times \beta$ into the list BNF itself. There is not much choice on how to do this—the fld-bijection is the only reasonable candidate. Therefore, the desired witness for list is $\mathsf{fld}\ (\mathsf{Inl}\ (\varepsilon a.\ a \in \mathsf{U}_{\mathsf{unit}}))$.

In the general case, a witness for F^j may depend on live variables. The dependencies on passive live variables of F^j are transported to the IF^j type without change. This is not possible for active live variables, since the fixed points don't depend on them. Instead, if Fwit^j depends on the k-th active live variable, a witness for IF^k must be plugged as an argument in Fwit before folding it to a witness for IF^j using fld^j. Applying this procedure recursively resembles generating words with a context free grammar. Indeed, the language of the following grammar (with the start symbol a_{m+j}) is the set of all IF^j witnesses:

TERMINALS: fld^j, Fwit^j_I, $a_1, \ldots, a_m$ for $j \in \{1, \ldots, n\}$, $I \subseteq \{1, \ldots, m+n\}$

NON-TERMINALS: $a_{m+1}, \ldots, a_{m+n}$

PRODUCTIONS: $a_{m+j} \longrightarrow \mathsf{fld}^j\ (\mathsf{Fwit}^j_{[i_1,\ldots,i_k]}\ a_{i_1}\ \ldots\ a_{i_k})$ for $j \in \{1, \ldots, n\}$ and all known witnesses $\mathsf{Fwit}^j_{[i_1,\ldots,i_k]}$ for F^j

Of course, the subset of the terminals $a_1, \ldots, a_m$ that are occurring in a word of this grammar are exactly the dependent variables of the corresponding IF^j witness.

Since we are interested in a minimal non-redundant set of witnesses (c.f. the end of Subsect. 5.3), it is enough to consider only the words that were derived using each production at most once.

This transfer of witnesses also applies to greatest fixed points, replacing IF by JF in the above construction and defining fld^j as the inverse of unf^j (which is a bijection).

6. Further Related Work

Some related work has already been covered in previous sections. Here we take a more systematic look at prior art, whether or not it has influenced our own work.

Interactive theorem provers include various mechanisms for introducing new types, which can be characterized as primitive (intrinsic), axiomatic, or definitional [BW99, p. 3]. In the world of HOL, the primitive type definition mechanism (Sect. 2.1) and the datatype package (Sect. 3.1) are the most widely used, but there are many others. Homeier [Hom05] developed a package to define quotient types (i.e., types whose elements correspond to equivalence classes of a base type) in HOL4, now ported to Isabelle/HOL [KU11]. Nominal Isabelle [Urb08] extends HOL with infrastructure for reasoning about datatypes containing name binders; (i.e., values are equal modulo renaming of their bound variables). Urban is currently rebasing it on the quotient package, possibly in unison 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 [Huf09] recast it into a purely definitional package, based on a large enough universal domain—a useful simplification in the context of domain theory, 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 constructors. Some ideas have yet to be automated in a definitional package: Völker [Völ95] sketches a categorical approach to datatypes that prefigures our work; Vos and Swierstra [VS02] elaborate an ad hoc construction for recursion through finite sets; and Paulson [Pau97] 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 [OS93] and codatatypes [Got07]. Hensel and Jacobs [HJ97] illustrate the categorical approach to (co)datatypes in PVS by axiomatic declarations of various flavors of trees (including our $\mathsf{tree}_{\mathsf{F}}$ and $\mathsf{tree}_{\mathsf{I}}$) with associated (co)iterators and proof principles. HOL_{ω}, which extends HOL4 with higher-rank polymorphism, provides a safe primitive for introducing abstractly specified types [Hom11]. Isabelle/ZF, based on ZFC, reduces (co)datatypes to (co)inductive predicates [Pau00], 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 [NUB11] but not the combination with non-free types.

7. Conclusion

This thesis presented the design and implementation of a definitional (co)datatype package in higher-order logic. Our work is motivated by long-existing limitations of the current datatype package based on a predefined universal type—namely the lack of codatatypes, the non-modular handling of nested recursion by unfolding and the impossibility of employing non-free types in datatype declarations.

In this work, we have tackled the problem from another angle: Our approach is based on category theory. In categorical terms datatypes correspond to initial algebras and codatatypes to final coalgebras. Both notions are well understood. However, performing those global categorical constructions in HOL is a difficult task. We achieve the construction maintaining a structural invariant on types that are participating in (co)datatype declarations. This invariant—incarnated by the notion of a bounded natural functor (BNF)—presents HOL type constructors as functors with additional categorical structure rather than functions between unstructured collections of types. Basic type constructors have a natural BNF structure. Moreover, BNFs are closed under initial algebra, final coalgebra and composition operations. This makes our approach fully compositional and enables an arbitrary mixture of (co)datatype and custom BNFs.

Of course, the prototypical implementation still needs to reach the level of usability of the existing datatype package. Also, the codatatype construction is not fully automated yet. Nevertheless, the automation developed so far already enables the usage of non-free structures in datatypes (cf. Appendix A) and the positive effects of the modularity of our approach are also already visible (cf. Appendix B).

As ongoing work we are automating the codatatype package and working on the integration of the package in Isabelle. Furthermore, we plan to exploit the relator structure of BNFs to obtain a notion of parametric constants in HOL. For those constants, we will be able to prove Wadler's free theorems [Wad89] literally for free in HOL.

Bibliography

[AM89] Peter Aczel and Nax Paul Mendler. A final coalgebra theorem. In *CTCS '89*, volume 389 of *LNCS*, pages 357–365. Springer, 1989.

[And02] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, volume 27 of *Applied Logic*. Springer, 2nd edition, 2002.

[Bar93] Michael Barr. Terminal coalgebras in well-founded set theory. *Theor. Comput. Sci.*, 114(2):299–315, 1993.

[Bar94] Michael Barr. Additions and corrections to "Terminal coalgebras in well-founded set theory." *Theor. Comput. Sci.*, 124:189–192, 1994.

[Ber98] Stefan Berghofer. Definitorische Konstruktion induktiver Datentypen in Isabelle/HOL (In German). Master's thesis, Technische Universität München, 1998.

[BW99] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL—Lessons learned in formal-logic engineering. In *TPHOLs '99*, volume 1690 of *LNCS*, pages 19–36, 1999.

[Chu40] Alonzo Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[DHS05] Till Mossakowski Daniel Hausmann and Lutz Schröder. Iterative circular coinduction for CoCasl in Isabelle/HOL. In *FASE '05*, LNCS, pages 341–356, 2005.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[Got07] Hanne Gottliebsen. Co-inductive proofs for streams in PVS. In *TPHOLs '07 (Emerging Trends)*, pages 113–127, 2007.

[Gun93] Elsa L. Gunter. Why we can't have SML-style `datatype` declarations in HOL. In *TPHOLs '92*, volume A-20 of *IFIP Transactions*, pages 561–568. North-Holland/Elsevier, 1993.

[Gun94] Elsa L. Gunter. A broader class of trees for recursive type definitions for HOL. In *HUG '93*, volume 780 of *LNCS*, pages 141–154. Springer, 1994.

[Har95] John Harrison. Inductive definitions: Automation and application. In *TPHOLs '95*, volume 971 of *LNCS*, pages 200–213. Springer, 1995.

[Har96] John Harrison. HOL Light: A tutorial introduction. In *FMCAD '96*, volume 1166 of *LNCS*, pages 265–269. Springer, 1996.

[HJ97] Ulrich Hensel and Bart Jacobs. Proof principles for datatypes with iterated recursion. In *Category Theory and Computer Science*, pages 220–241, 1997.

[HJ98] Claudio Hermida and Bart Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152, 1998.

[Hom05] Peter V. Homeier. A design structure for higher order quotients. In *TPHOLs '05*, volume 3603 of *LNCS*, pages 130–146. Springer, 2005.

[Hom11] Peter Vincent Homeier. Abstract data types in HOL-Omega. http://permalink.gmane.org/gmane.comp.mathematics.hol/1168, 18 Apr. 2011.

[Huf09] Brian Huffman. A purely definitional universal domain. In *TPHOLs '09*, volume 5674 of *LNCS*, pages 260–275. Springer, 2009.

[KU11] Cezary Kaliszyk and Christian Urban. Quotients revisited for Isabelle/HOL. In *SAC '11*, pages 1639–1644. ACM, 2011.

[MA86] Ernest G. Manes and Michael A. Arbib. *Algebraic Approaches to Program Semantics*. Springer, 1986.

[Mel89] Thomas F. Melham. Automating recursive type definitions in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer, 1989.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NUB11] Keiko Nakata, Tarmo Uustalu, and Marc Bezem. A proof pearl with the fan theorem and bar induction—Walking through infinite trees with mixed induction and coinduction. In *APLAS '11*, volume 7078 of *LNCS*, pages 353–368. Springer, 2011.

[OS93] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical Report CSL-93-9R, C.S. Lab., SRI International, 1993.

[Pau97] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Log. Comput.*, 7(2):175–204, 1997.

[Pau00] Lawrence C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In *Proof, Language, and Interaction*, pages 187–212. MIT Press, 2000.

[PT12] Andrei Popescu and Dmitriy Traytel. Ordinals and cardinals in HOL. http://www21.in.tum.de/~traytel/lics12_card.tgz, 2012.

[Rut98] J. J. M. M. Rutten. Relators and metric bisimulations. *Electr. Notes Theor. Comput. Sci.*, 11:252–258, 1998.

[Rut00] J. J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theor. Comput. Sci.*, 249:3–80, 2000.

[TPB12] Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. Foundational, compositional (co)datatypes for higher-order logic—Category theory applied to theorem proving. In *LICS*, 2012. To appear.

[Urb08] Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reasoning*, 40(4):327–356, 2008.

[Völ95] N. Völker. On the representation of datatypes in Isabelle/HOL. Isabelle Users Workshop, 1995.

[VS02] Tanja E. J. Vos and S. Doaitse Swierstra. Inductive data types with negative occurrences in HOL. Thirty Five Years of Automath, 2002.

[Wad89] Philip Wadler. Theorems for free! In *FPCA '89*, pages 347–359. ACM, 1989.

[WW07] Makarius Wenzel and Burkhart Wolff. Building formal method tools in the Isabelle/Isar framework. In *TPHOLs '07*, volume 4732 of *LNCS*, pages 352–367. Springer, 2007.

A. Recasting Berghofer's Running Example

In his master's thesis [Ber98], Stefan Berghofer demonstrates the treatment of nested recursion on the example of a very simple recursive term datatype. A term is either a variable identifier or a function identifier applied to other terms. The goal is to prove a simple theorem about substitutions of variable identifiers.

In the following sections, we define the term datatype in three different ways: first using the old datatype package, second our new datatype package, and finally using an alternate orderless representation of function arguments by the finite set type $'a$ fset instead of the list type $'a$ list.

Of course, the usability features of the old datatype package are more advanced than those of our package. The sugared syntax for datatypes is convenient for functional programming languages, in contrast to our raw fixed point equations. We are working on implementing the sugared syntax on top of our raw syntax which is a conceptually easy task but essential for the usefulness of the package.

Nevertheless, some of the discussed benefits of our approach are already visible at this small example even with the prototypical version of the new package. For instance, the third definition using the type $'a$ fset is not possible with the old package.

A.1. Old Datatype Package by Berghofer/Wenzel

The old datatype package unfolds the definition of the $'a$ list datatype in the following declaration, making the latter mutually recursive.

datatype ($'a$, $'b$) TRM = Var $'a$ | App $'b$ ($'a$, $'b$) TRM list

definition subst-TRM **where**
subst-TRM f = TRM-rec-1 f (λb xs ys. App b ys) Nil (λx xs y ys. Cons y ys)

definition subst-TRM-list **where**
subst-TRM-list f = TRM-rec-2 f (λb xs ys. App b ys) Nil (λx xs y ys. Cons y ys)

The user of the old datatype package must be aware of the replacement of nested recursion by mutual recursion. For instance, even if we are only interested in the first statement that is not involving subst-TRM-list of the following theorem, we need to prove a stronger version in order to strengthen the induction hypothesis. Stating the right theorem is hereby the creative part, while the automation takes care of the rest.

theorem subst-TRM (subst-TRM f $\circ$ g) t = subst-TRM f (subst-TRM g t)
and subst-TRM-list (subst-TRM f $\circ$ g) ts = subst-TRM-list f (subst-TRM-list g ts)
by (induct t **and** ts) simp-all

A.2. New Datatype Package

A.2.1. Short Library of Lists

For demonstration purposes we define the list datatype using the new package. Of course, we could also reuse the existing type since it is registered as a BNF.

lfp newLIST: $'l = \text{unit} + 'a \times 'l$

Note that the raw interface of fixed point equations does not include name bindings for datatype constructors. Instead we define the constructors manually and also prove some characteristic theorems of their interaction with the automatically derived map and set constants of the newly defined type. Future versions of the new datatype package will perform all of these operations automatically.

definition newNil **where** newNil = newLIST-fld (Inl ())
definition newCons **where** newCons x xs = newLIST-fld (Inr (x, xs))

definition newLIST-all **where**
newLIST-all P = newLIST-iter (sum-case (λxs. True) (λ(x, xs). P x $\wedge$ xs))

lemmas newLIST-defs =
newLIST.defs newLIST.iter newLIST.fld-diff newNil-def newCons-def newLIST-all-def
newLISTRTC.defs sum.defs prod.defs ID.defs collect-def-raw

lemma newLIST-map f newNil = newNil
by (simp add: newLIST-defs)

lemma newLIST-map f (newCons x xs) = newCons (f x) (newLIST-map f xs)
by (simp add: newLIST-defs)

lemma newLIST-set newNil = {}
by (simp add: newLIST-defs)

lemma newLIST-set (newCons x xs) = {x} $\cup$ newLIST-set xs
by (simp add: newLIST-defs)

lemma newLIST-all P newNil = True
by (simp add: newLIST-defs)

lemma newLIST-all P (newCons x xs) = (P x $\wedge$ newLIST-all P xs)
by (simp add: newLIST-defs)

lemma newLIST-induct:
fixes xs :: 'a newLIST
assumes IB: P newNil **and** IH: $\bigwedge$x xs. P xs $\Longrightarrow$ P (newCons x xs)
shows P xs
proof (induct rule: newLIST.fld-induct)

fix xs :: unit + 'a × 'a newLIST
assume raw-IH: $\bigwedge$a. a ∈ newLISTRTC-set2 xs $\Longrightarrow$ P a
show P (newLIST-fld xs)
proof (cases xs)
case (Inl a) **with** IB **show** ?thesis **by** (simp add: newNil-def)
next
case (Inr b)
then obtain y ys **where** yys: newLIST-fld xs = newCons y ys
by (auto simp add: newLIST-defs intro: prod.exhaust)
hence ys ∈ newLISTRTC-set2 xs **by** (simp add: newLIST-defs)
with raw-IH **have** P ys **by** blast
with IH **have** P (newCons y ys) **by** blast
with yys **show** ?thesis **by** simp
qed
qed

lemma newLIST-all-cong:
newLIST-all (λx. f x = g x) xs $\Longrightarrow$ newLIST-map f xs = newLIST-map g xs
by (induct xs rule: newLIST-induct) auto

lemma newLIST-all-mono: ⟦newLIST-all P xs; $\bigwedge$x. P x $\Longrightarrow$ Q x⟧ $\Longrightarrow$ newLIST-all Q xs
by (induct xs rule: newLIST-induct) auto

A.2.2. Term Datatype

lfp newTRM: 't = 'a + 'b × 't newLIST

definition newVar **where** newVar a = newTRM-fld (Inl a)
definition newApp **where** newApp b ts = newTRM-fld (Inr (b, ts))

lemmas newTRM-defs =
newTRM.iter newTRM.fld-diff newVar-def newApp-def
newTRMRTC.defs sum.defs prod.defs ID.defs collect-def-raw

lemma newTRM-induct:
fixes t :: ('a, 'b) newTRM
assumes IB: $\bigwedge$a. P (newVar a) **and** IH: $\bigwedge$b ts. newLIST-all P ts $\Longrightarrow$ P (newApp b ts)
shows P t
proof (induct rule: newTRM.fld-induct)
fix t :: 'a + 'b × (('a, 'b) newTRM) newLIST
assume raw-IH: $\bigwedge$a. a ∈ newTRMRTC-set3 t $\Longrightarrow$ P a
show P (newTRM-fld t)
proof (cases t)
case (Inl a) **with** IB **show** ?thesis **by** (simp add: newVar-def)
next
case (Inr app)
then obtain b ts **where** bts: newTRM-fld t = newApp b ts

```
    by (auto simp add: newTRM-defs intro: prod.exhaust)
  hence newLIST-all (λt'. t' ∈ newTRMRTC-set3 t) ts (is newLIST-all (?P t) ts)
  proof (induct ts arbitrary: t rule: newLIST-induct)
    case (2 x xs)
    hence x ∈ newTRMRTC-set3 t by (simp add: newTRM-defs)
    moreover
    from 2(2) have ∗: newTRMRTC-set3 (Inr (b, xs)) ⊆ newTRMRTC-set3 t
      by (auto simp add: newTRM-defs)
    from 2(1) have newLIST-all (?P (Inr (b, xs))) xs by (simp add: newTRM-defs)
    hence newLIST-all (?P t) xs by (rule newLIST-all-mono) (rule set-mp[OF ∗])
    ultimately show ?case by simp
  qed simp
  with raw-IH have newLIST-all P ts by (induct ts rule: newLIST-induct) auto
  with IH have P (newApp b ts) by blast
  with bts show ?thesis by simp
 qed
qed
```

```
definition subst-newTRM where
 subst-newTRM f = newTRM-iter (sum-case f (newTRM-fld o Inr))
```

```
lemma subst-newTRM f (newVar a) = f a
by (simp add: subst-newTRM-def newTRM-defs)
```

```
lemma subst-newTRM f (newApp b ts) = newApp b (newLIST-map (subst-newTRM f) ts)
by (simp add: subst-newTRM-def newTRM-defs)
```

With the new datatype, there is no mutual recursion going on in this example. The user can state the theorem that she wants to prove and not a stronger one. On the other hand, some creativity is required to select the right simplification lemmas for the proof.

```
theorem subst-newTRM (subst-newTRM f ∘ g) t = subst-newTRM f (subst-newTRM g t)
by (induct t rule: newTRM-induct)
 (auto simp add: newLIST-all-cong newLIST.map-comp' comp-def)
```

A.3. Usage of Finite Sets in Datatypes

Non-free structures such as finite sets are not allowed in the old datatype declarations. Since finite sets can be declared as BNFs, the new datatype package handles them smoothly.

```
lfp fsetTRM: 't = 'a + 'b × 't fset
```

```
definition fsetVar where fsetVar a = fsetTRM-fld (Inl a)
definition fsetApp where fsetApp b ts = fsetTRM-fld (Inr (b, ts))
```

```
lemmas fsetTRM-defs =
 fsetTRM.iter fsetTRM.fld-diff fsetVar-def fsetApp-def
```

fsetTRMRTC.defs sum.defs prod.defs ID.defs fset.defs collect-def-raw

lemma fsetTRM-induct:
fixes t :: ('a, 'b) fsetTRM
assumes
IB: ⋀a. P (fsetVar a) **and**
IH: ⋀b ts. (∀ x ∈ fset ts. P x) ⟹ P (fsetApp b ts)
shows P t
proof (induct rule: fsetTRM.fld-induct)
fix t :: 'a + 'b × (('a, 'b) fsetTRM) fset
assume raw-IH: ⋀a. a ∈ fsetTRMRTC-set3 t ⟹ P a
show P (fsetTRM-fld t)
proof (cases t)
case (Inl a) **with** IB **show** ?thesis **by** (simp add: fsetVar-def)
next
case (Inr app)
then obtain b ts **where** bts: fsetTRM-fld t = fsetApp b ts
by (auto simp add: fsetTRM-defs intro: prod.exhaust)
hence ∀ x ∈ fset ts. x ∈ fsetTRMRTC-set3 t **by** (simp add: fsetTRM-defs)
with raw-IH **have** ∀ x ∈ fset ts. P x **by** blast
with IH **have** P (fsetApp b ts) **by** blast
with bts **show** ?thesis **by** simp
qed
qed

definition subst-fsetTRM **where**
subst-fsetTRM f = fsetTRM-iter (sum-case f (fsetTRM-fld o Inr))

lemma subst-fsetTRM f (fsetVar a) = f a
by (simp add: subst-fsetTRM-def fsetTRM-defs)

lemma subst-fsetTRM f (fsetApp b ts) = fsetApp b (fset-map (subst-fsetTRM f) ts)
by (simp add: subst-fsetTRM-def fsetTRM-defs)

lemma fset-map-cong: ∀ x ∈ fset X. f x = g x ⟹ fset-map f X = fset-map g X
by (rule fset.map-cong) (simp only: fset.defs)

theorem subst-fsetTRM (subst-fsetTRM f ∘ g) t = subst-fsetTRM f (subst-fsetTRM g t)
by (induct t rule: fsetTRM-induct)
(auto simp add: fset.map-comp′ comp-def intro: arg-cong[OF fset-map-cong])

B. Comparison of the Datatype Packages with Focus on Nested Recursion

Nested recursion is handled by the Melham–Gunter approach by unfolding definitions of nested datatype and thereby simulating nested recursion by mutual recursion. Unfolding is a source of non-modularity. It has negative influences on the flexibility and performance. We demonstrate this by considering iterated nested recursion. First we compare the current Isabelle datatype package implemented by Berghofer and Wenzel using the Melham–Gunter approach to our package on a nullary datatype, that has a single constructor with iterated application of the list type constructor on the recursive type argument. We measure the CPU time that is needed to process the **datatype** command. The measured times are given in seconds.

n	**datatype** $d = C\ (d\ \overbrace{\mathsf{list} \ldots \mathsf{list}}^{n \text{ times}})$ old package	new package
1	1.144	1.244
2	1.788	1.608
3	2.680	2.008
4	5.704	2.620
5	6.152	3.148
6	8.569	3.620
7	11.461	4.280
8	19.181	4.832
9	24.738	5.684

The measurements confirm that our modular handling of nested recursion pays off. One could argue that examples that are nesting nine levels of lists are not very realistic. First, this is not necessarily true—complex formalizations may require complex datatypes. Second, the gain of time is also visible for smaller depth of nesting, if we increase the number of constructors. We have performed the measurements for two and three constructors.

n_1	n_2	**datatype** $d = C_1\ (d\ \overbrace{\mathsf{list} \ldots \mathsf{list}}^{n_1 \text{ times}}) \mid C_2\ (d\ \overbrace{\mathsf{list} \ldots \mathsf{list}}^{n_2 \text{ times}})$ old package	new package
2	2	4.620	3.764
3	3	13.045	4.802
4	4	19.373	5.980

n_1	n_2	n_3	**datatype** $d = C_1\ (d\ \overbrace{\mathsf{list} \ldots \mathsf{list}}^{n_1 \text{ times}}) \mid C_2\ (d\ \overbrace{\mathsf{list} \ldots \mathsf{list}}^{n_2 \text{ times}}) \mid C_3\ (d\ \overbrace{\mathsf{list} \ldots \mathsf{list}}^{n_3 \text{ times}})$	
			old package	new package
2	2	2	8.845	5.948
3	3	3	23.241	8.033
4	4	4	38.826	9.917

Datatypes with an even much bigger number of constructors are common in Isabelle formalizations. A final example provided to us by Christian Urban in a private communication justifies this. It is taken from a formalisation of some parts of UNIX. The example consideres the following type declarations:

type_synonym α t_sprocess = α list × t_process option

type_synonym α t_sfile = α list × t_file option

type_synonym α t_ssocket = α list × t_socket option

type_synonym α t_smsg = α list × t_msg option

datatype t_event_s =
  Open_s (t_event_s t_sprocess) (t_event_s t_sfile) t_open_flags
| CloseFFd_s (t_event_s t_sprocess) (t_event_s t_sfile)
| CloseSFd_s (t_event_s t_sprocess) (t_event_s t_ssocket)
| UnLink_s (t_event_s t_sprocess) (t_event_s t_sfile)
| Rmdir_s (t_event_s t_sprocess) (t_event_s t_sfile)
| Mkdir_s (t_event_s t_sprocess) (t_event_s t_sfile)
| Truncate_s (t_event_s t_sprocess) (t_event_s t_sfile)
| FTruncate_s (t_event_s t_sprocess) (t_event_s t_sfile)
| ReadFile_s (t_event_s t_sprocess) (t_event_s t_sfile)
| WriteFile_s (t_event_s t_sprocess) (t_event_s t_sfile)
| Execve_s (t_event_s t_sprocess) (t_event_s t_sfile)
| CreateMsg_s (t_event_s t_sprocess)
| SendMsg_s (t_event_s t_sprocess) (t_event_s t_smsg)
| RecvMsg_s (t_event_s t_sprocess) (t_event_s t_smsg)
| RemoveMsg_s (t_event_s t_sprocess) (t_event_s t_smsg)

Hereby, all undeclared types are either type synonyms without arguments or non-recursive datatype and therefore not interesting when considering nested recursion. The depth of nesting is only one for the above datatype, but the time the old datatype package requires to process the declaration is beyond one hour. In contrast, the new datatype package processes the example in less then 75 seconds.