Bounded natural functors and their application to describe binding-aware syntaxes

Márk Széles
email: mark.szeles@ru.nl

MFoCS Seminar
Radboud University Nijmegen

25 January 2022

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

Category Theory Applied to Theorem Proving

Dmitriy Traytel
Technische Universität München
Munich, Germany

Andrei Popescu
Technische Universität München
Munich, Germany
Institute of Mathematics Simion Stoilow
Bucharest, Romania

Jasmin Christian Blanchette
Technische Universität München
Munich, Germany

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

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

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

I. Introduction

Higher-order logic (HOL, Sect. II) forms the basis of several popular interactive theorem provers, notably HOL4 [10], HOL Light [16], and Isabelle/HOL [27]. Its straightforward semantics, which interprets types as sets (collections) of elements, makes it an attractive choice for many computer science and mathematical formalizations.

The theorem provers belonging to the HOL family traditionally encourage their users to adhere to the definitional approach, whereby new types and constants are defined in terms of existing constructs rather than introduced axiomatically. Following the LCF philosophy [11], theorems can be generated only within a small inference kernel, reducing the amount of code that must be trusted.

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

The most important high-level mechanism is undoubtedly the datatype package, which automates the derivation of (freely generated inductive) datatypes. Melham [26] had already devised such a definitional package two decades ago. His approach, considerably extended by Gunter [13], [14] and simplified by Harrison [15], now lies at the heart of the implementations in HOL4, HOL Light, and Isabelle/HOL.

Despite having withstood the test of time, the Melham–Gunter approach suffers from a few limitations that impair its usefulness. The most pressing issue is probably its ignorance of codatatypes (the coinductive counterpart of datatypes). Lacking a definitional package to automate the definition of codatatypes, users face a tough choice between tedious manual constructions and risky axiomatizations [17].

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

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

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

$$\texttt{codatatype}\ \alpha\ \mathsf{tree} = \mathsf{Node}\ \alpha\ ((\alpha\ \mathsf{tree})\ \mathsf{fset})$$

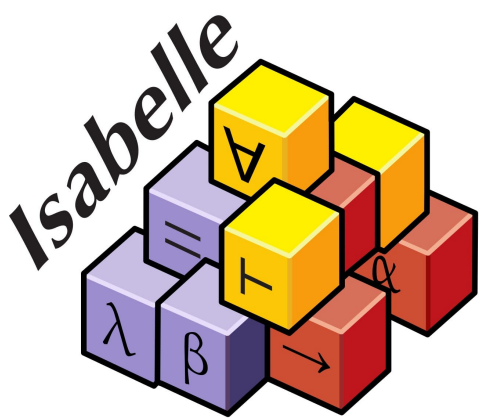
This paper presents a fully compositional framework for defining datatypes and codatatypes in HOL, including mutual and nested (co)recursion through an arbitrary combination of datatypes, codatatypes, and other interesting type constructors (Sect. III). The underlying mathematical apparatus is taken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

The main mathematical contribution of this paper is a novel class of functors—the BNFs—that is closed under the initial algebra, final coalgebra, and composition operations and that allows initial and final constructions in a sufficiently

Bounded natural functors: motivation

Aim: defining a definitional datatype package for Isabelle/HOL, that is

- Compositional
- Supports both datatypes and codatatypes, and allows mixing those
- Allows some well-behaved non-free types (e.g. finite multisets)



Functors

A functor has the following structure

- $\mathsf{F} : \mathsf{Set} \to \mathsf{Set}$
- $\mathsf{map}_F : (\mathsf{A} \to \mathsf{B}) \to \mathsf{F\ A} \to \mathsf{F\ B}$
- $\mathsf{map}_F\ \mathsf{id} = \mathsf{id}$
- $\mathsf{map}_F\ (\mathsf{g} \circ \mathsf{f}) = \mathsf{map}_F\ \mathsf{g} \circ \mathsf{map}_F\ \mathsf{f}$

Example: List is a functor

$$\mathsf{map}_{List}\ \mathsf{g}\ [x_1, ..., x_n] = [g(x_1), ..., g(x_n)]$$

Algebras of a functor

F : $Set \to Set$

- An F-algebra is a pair, $(A : Set, \alpha : F(A) \to A)$
- A map of F-algebras (A, α) and (B, β) is a map $f : A \to B$ making the following diagram commute:

$$\begin{array}{ccc} F(A) & \xrightarrow{map_F\ f} & F(B) \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ A & \xrightarrow{\quad f \quad} & B \end{array}$$

Initial algebra of a functor

- An algebra $(I, ctor)$ is initial, if for all algebras (A, α), there is a unique map $f : (I, ctor) \to (A, \alpha)$.
- If it exists, the initial algebra is unique up to isomorphism
- Lambek's theorem: *ctor* is an isomorphism
- I is the least fixed point of F :
  $I = \mu X.F(X)$

$$\begin{array}{ccc} F(I) & \xrightarrow{map_F\ f} & F(A) \\ \downarrow{\scriptstyle ctor}_{\cong} & & \downarrow{\scriptstyle \alpha} \\ I & \xrightarrow{!f} & A \end{array}$$

Datatypes as initial algebras (example)

$F : Set \to Set$
$F(X) = 1 + X$

- $\mathbb{N}$ is the initial algebra of F:

$$\begin{array}{ccc} 1 + \mathbb{N} & \xrightarrow{map_F\ f} & 1 + A \\ \downarrow{\scriptstyle ctor}_{\cong} & & \downarrow{\scriptstyle \alpha} \\ \mathbb{N} & \xrightarrow{!f} & A \end{array}$$

- We can use initiality to define recursive functions out of $\mathbb{N}$
- Example:
  - $A = Bool$
  - $\alpha(\iota_1(*)) = tt$
  - $\alpha(\iota_2(b)) = not(b)$
  - The unique function f is *isEven*

List is container-like and bounded

- $\text{set}_{List,A} : \text{List A} \to \mathcal{P}(\text{A})$
- $\text{set}_{List,A}\,[x_1, ..., x_n] = \{x_1, ..., x_n\}$
- set_{List} is a natural transformation $List \to \mathcal{P}$, i.e. for all $f : A \to B$ the following square commutes:

$$\begin{array}{ccc} \text{List } A & \xrightarrow{set_{List,A}} & \mathcal{P}(A) \\ \downarrow{\scriptstyle map_{List}\ f} & & \downarrow{\scriptstyle map_{P}\ f} \\ \text{List } B & \xrightarrow{set_{List,B}} & \mathcal{P}(B) \end{array}$$

- For all A, $|\text{set}_{List,A}\,xs| < bd_{List} = \aleph_0$

List has a relator structure

$\mathrm{rel}_{List} : Rel(A, B) \to Rel(\mathrm{List}(A), \mathrm{List}(B))$

$(xs, ys) \in rel_{List}\ R \iff$
$\exists (zs \in List(A \times B)).$
$\quad (\mathrm{set}_{List} zs \subseteq R \wedge map_{List}\ \pi_1\ zs = xs \wedge map_{List}\ \pi_2\ zs = ys),$

where $Rel(X, Y)$ denotes $\mathcal{P}(X \times Y)$

The relator commutes with relation composition, and preserves equality relations

Example:

- $\mathrm{R} = \{(1, 2), (3, 4), (5, 6)\}$
- $[1, 3, 3]$ and $[2, 4, 4]$ are related by the relator, $zs = [(1, 2), (3, 4), (3, 4)]$.
- $[1, 3, 5]$ and $[2, 4]$ are NOT related.
- $[1, 3, 3]$ and $[4, 2, 4]$ are NOT related.

Bounded natural functors (BNFs)

A (unary) bounded natural functor is a functor $F : Set \to Set$, with the following properties:

- F is container-like: there exists a natural transformation $set_F : F \to P$
- F is bounded: $\forall A, \forall x \in F(A), |set_{F,A}(x)| < bd_F$, for some fixed infinite cardinal bd_F (independent of A).
- F is equipped with a relator rel_F, that commutes with relation composition and preserves equality relations

This definition can be generalized to functors with multiple arguments.

Properties of bounded natural functors

Bounded natural functors (BNFs) are a suitable basis for a datatype package for Isabelle/HOL

- BNFs have initial algebras and final coalgebras
- BNFs are closed under composition, initial algebras, and final coalgebras
- Basic functors are BNFs (e.g. constant, sum, product)
- Some interesting non-free functors are also BNFs (e.g. finite multiset)
- The initial algebras and final coalgebras of BNFs are expressible in HOL (the proof uses the assumption about bd_F)

Bindings as Bounded Natural Functors

JASMIN CHRISTIAN BLANCHETTE, Vrije Universiteit Amsterdam, the Netherlands and Max-Planck-Institut für Informatik, Germany
LORENZO GHERI, Middlesex University London, UK
ANDREI POPESCU, Middlesex University London, UK and Institute of Mathematics Simion Stoilow of the Romanian Academy, Romania
DMITRIY TRAYTEL, ETH Zürich, Switzerland

We present a general framework for specifying and reasoning about syntax with bindings. Abstract binder types are modeled using a universe of functors on sets, subject to a number of operations that can be used to construct complex binding patterns and binding-aware datatypes, including non-well-founded and infinitely branching types, in a modular fashion. Despite not committing to any syntactic format, the framework is "concrete" enough to provide definitions of the fundamental operators on terms (free variables, alpha-equivalence, and capture-avoiding substitution) and reasoning and definition principles. This work is compatible with classical higher-order logic and has been formalized in the proof assistant Isabelle/HOL.

CCS Concepts: • **Theory of computation → Logic and verification**; **Higher order logic**; **Type structures**;

Additional Key Words and Phrases: syntax with bindings, inductive and coinductive datatypes, proof assistants

ACM Reference Format:
Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. 2019. Bindings as Bounded Natural Functors. *Proc. ACM Program. Lang.* 3, POPL, Article 22 (January 2019), 34 pages. https://doi.org/10.1145/3290335

1 INTRODUCTION

The goal of this paper is to systematize and simplify the task of constructing and reasoning about variable binding and variable substitution, namely the operations of binding variables into terms and of replacing them with other variables or terms in a well-scoped fashion. These mechanisms play a fundamental role in the metatheory of programming languages and logics.

There is a lot of literature on this topic, proposing a wide range of binding formats (e.g., Pottier [2006], Sewell et al. [2010], Urban and Kaliszyk [2012], Weirich et al. [2011]) and reasoning mechanisms (e.g., Kaiser et al. [2017], Chlipala [2008], Pitts [2006], Felty et al. [2015a], Urban et al. [2007]). The POPLmark formalization challenge [Aydemir et al. 2005] has received quite a lot of attention in the programming language and interactive theorem proving communities. And

Authors' addresses: Jasmin Christian Blanchette, Department of Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, Amsterdam, 1081 HV, the Netherlands, j.c.blanchette@vu.nl, Research Group 1, Max-Planck-Institut für Informatik, Saarland Informatics Campus E1 4, Saarbrücken, 66123, Germany; Lorenzo Gheri, School of Science and Technology, Middlesex University London, The Burroughs, London, NW4 4BT, UK, lg571@live.mdx.ac.uk; Andrei Popescu, Middlesex University London, School of Science and Technology, The Burroughs, London, NW4 4BT, UK, A.Popescu@mdx.ac.uk, Institute of Mathematics Simion Stoilow of the Romanian Academy, Calea Grivitei 21, Bucharest, 010702, Romania; Dmitriy Traytel, Institute of Information Security, Department of Computer Science, ETH Zürich, Universitätstrasse 6, Zürich, 8092, Switzerland, traytel@inf.ethz.ch.

2475-1421/2019/1-ART22
https://doi.org/10.1145/3290335

Syntax of the λ-calculus

Let α be the set of variables.

$$t ::= \mathit{Var}\ \alpha \mid \lambda\alpha.t \mid t\ t$$

We could describe this syntax by a functor:

$$F(\alpha, \tau) = \alpha + (\alpha \times \tau) + (\tau \times \tau),$$

where τ is the set of possible terms. Then we can construct terms as

$$T(\alpha) := \mu\tau.F(\alpha, \tau)$$

However we also want to encode binding information!

Binding-aware syntax of the λ-calculus

We distinguish between term variables (β_1) and variables used in a λ-binder (α_1).

We distinguish between possible subterms, by considering whether a top binding variable may bind in a subterm.

We describe this structure by a functor:

$$F(\beta_1, \alpha_1, \tau_1, \tau_2) = \beta_1 + (\alpha_1 \times \tau_1) + (\tau_2 \times \tau_2),$$

where τ_1, τ_2 are the sets of possible terms. A top binding variable may bind in a term of kind τ_1, but not in a term of kind τ_2.

Binding-aware syntax of the λ-calculus

$$F(\beta_1, \alpha_1, \tau_1, \tau_2) = \beta_1 + (\alpha_1 \times \tau_1) + (\tau_2 \times \tau_2),$$

- F is a bounded natural functor
- The binding dispatcher relation θ expresses in which kind of terms the top binding variables may bind.

$$\theta = \{(\alpha_1, \tau_1)\}$$

Defining actual terms

We define the actual terms with variables in α by taking the least fixed point of F:

$$T(\alpha) = \mu\,\tau.F(\alpha, \alpha, \tau, \tau).$$

We have an isomorphism

$$ctor : \alpha + (\alpha \times T(\alpha)) + (T(\alpha) \times T(\alpha)) \xrightarrow{\sim} T(\alpha).$$

If $t : T(\alpha)$, there exists an x, such that $t = \text{ctor } x$.

- topFree_1 t := $set_{F,\beta_1}\ x$
- topBind_1 t := $set_{F,\alpha_1}\ x$
- rec_1 t := $set_{F,\tau_1}\ x$
- rec_2 t := $set_{F,\tau_2}\ x$,

Binding-aware syntax of the λ-calculus with let

Let α be the set of variables.

$$t ::= \mathit{Var}\ \alpha \mid \lambda\alpha.t \mid t\ t \mid \mathit{let}\ \alpha\ = t\ \mathit{in}\ t$$

$$F(\beta_1, \alpha_1, \tau_1, \tau_2) = \beta_1 + (\alpha_1 \times \tau_1) + (\tau_2 \times \tau_2) + (\alpha_1 \times \tau_2 \times \tau_1)$$

- F is a bounded natural functor
- The binding dispatcher relation θ expresses, in which kind of terms the top binding variables may bind.

$$\theta = \{(\alpha_1, \tau_1)\}$$

Map restriction on a functor

For an $f : X \to X$, let $A_f = \{x \in X \mid f(x) \neq x\}$, the support of f is defined as

$$supp(f) = A_f \cup f(A_f)$$

We restrict functoriality of F

- to small-support endofunctions on term variables (β).
- to small-support endobijections on binding variables (α).

Binder types

A binder type is formed from a BNF, that

- has term-variable, binding-variable and potential term inputs
- has restricted functoriality
- is equipped with a binding dispatcher relation between term inputs and binding variable inputs.

Towards α-equivalence (top bound variables)

$\mathsf{topBind}_{i,j}$ is the set of top binding variables of kind i, which may bind in terms of kind j. In our case:

- $\mathsf{topBind}_{1,1}\ \mathsf{t} = \mathsf{topBind}_1\ \mathsf{t}$
- $\mathsf{topBind}_{1,2}\ \mathsf{t} = \emptyset$

To generally define $\mathsf{topBind}_{i,j}$, we use the binding dispatcher relation θ:

$$\mathsf{topBind}_{i,j}\ t = \begin{cases} \mathsf{topBind}_i\ t, & \text{if } (\alpha_i, \tau_j) \in \theta \\ \emptyset, & \text{otherwise} \end{cases}$$

Towards α-equivalence (free variables)

If $t : T(\alpha)$, there exists an x, such that $t = \text{ctor } x$.
We define free variables of t inductively:

- $a \in \text{topFree}_1 x \Rightarrow a \in \text{FVars}_1\ x$
- $t \in \text{rec}_j\, x \wedge a \in \text{FVars}_1\ t \setminus \text{topBind}_{1,j}\, x \Rightarrow a \in \text{FVars}_1\ x$

To express renaming via an endobijection f, we define
$\text{map}_T : (\alpha \xrightarrow{\sim} \alpha) \to T(\alpha) \to T(\alpha)$

$$\text{map}_T\ f\ (\text{ctor } x) = \text{ctor}\ (\text{map}_F\ f\ f\ (\text{map}_T\ f)\ (\text{map}_T\ f)\ x)$$

α-equivalence for the untyped λ-calculus

Let $t_1 = \text{ctor } x_1$, $t_2 = \text{ctor } x_2$.
If we can apply a suitable renaming $f : \alpha \to \alpha$ to x_1 yielding x_2, t_1 and t_2 are α-equivalent. We require f to satisfy some conditions $\mathit{cond}(f)$.

- f is bijective
- f does not change free variables in the recursive components of x_1, that are not top bound, i.e.

$$\forall a \in \bigcup_{j \in \{1,2\}} \bigcup_{t \in \text{rec}_j\, x_1} \text{FVars } t \setminus \text{topBind}_{1,j}\, x_1.\ f\ a = a$$

If $\mathit{cond}(f)$ and $\text{rel}_F\ (=)\ (\text{Gr}\ f)\ [(\lambda\, t_1\ t_2 \to \text{map}_T\ f\ t_1 \equiv_\theta t_2)]^2\ x_1\ x_2$, then $t_1 \equiv_\theta t_2$

α-quotiented terms are defined as $\mathbf{T}(\alpha) := T(\alpha)/\equiv_\theta$

Operations on α-quotiened terms

We can for example define the following operations on α-quotiened terms:

- Binding-aware induction and recursion principles (easier to use, than the ones inherited from raw terms)
- Variable for variable substitution
- Term for variable substitution

Example: variable for variable substitution

We want to define a capture-avoiding variable substitution function on α-quotiened terms

$$sub : (\alpha \to \alpha) \to \mathbf{T}(\alpha) \to \mathbf{T}(\alpha)$$

Desired property of *sub*, expressed on raw terms of the form *ctor x*:

$$(topBind\ x \cap FVars(ctor\ x) = \emptyset \wedge topBind\ x \cap supp\ f = \emptyset)$$
$$\longrightarrow sub\ f\ (ctor\ x) = ctor(map_F\ f\ id\ (sub\ f)\ (sub\ f)\ x),$$

where

$$supp(f) = \{a : \alpha \mid f\ a \neq a\} \cup f\ \{a : \alpha \mid f\ a \neq a\}$$

.

The binding-aware recursor can be used to define this operation.

Thank you for your attention!