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

yet, formal reasoning about bindings remains a major hurdle, especially when complex binding patterns are involved. Among the 15 solutions reported on the POPLmark website, only three address Parts 1B and 2B of the challenge, which involve recursively defined patterns; in each case, this is done through a low-level, ad hoc technical effort that is not entirely satisfactory.

To improve the situation, we believe that the missing ingredient is a semantics-based account of the binding and substitution mechanisms, as opposed to ever more general syntactic formats. Bindings can be abstractly understood in a world of *shapes* and of *content* that fills the shapes. A binder puts together one or more variables in a suitable shape that is connected with the body through common content. Variable renaming and replacement, which give rise to the notions of alpha-equivalence (naming equivalence) and capture-free substitution, amount to replacing the content while leaving the shape unchanged. To work with such a universe of shapes and content without committing to a syntactic format, we rely on a class of functors on sets. We employ different kinds of morphisms depending on the task: functions when substituting free variables, bijections when renaming bound variables, and both bijections and relations when defining alpha-equivalence.

We develop a class of functors that can express the action of arbitrarily complex binders while supporting the construction of the key operations involving syntax with bindings—such as free variables, alpha-equivalence, and substitution—and the proof of their fundamental properties. This class of functors subsumes a large number of results for a variety of syntactic formats. Another gain is *modularity*: Complex binding patterns can be developed separately and placed in larger contexts in a manner that guarantees correct scoping and produces correct definitions of alpha-equivalence and substitution. Finally, the abstract perspective also clarifies the acquisition of fresh variables, allowing us to go beyond finitary syntax. Our theory extends the scope of techniques for reasoning about bindings to infinitely branching and non-well-founded terms.

Our work targets the Isabelle/HOL proof assistant, a popular implementation of higher-order logic (HOL), and extends Isabelle's framework of bounded natural functors, abbreviated BNFs (Section 2). We analyze examples of binders and develop the axiomatization of binder types through a process of refinement. We first focus on correctly formalizing binding variables using a suitable subclass of BNFs (Section 3). Then we analyze how complex binder types can be constructed in a uniform way (Section 4). Finally, we try to apply these ideas to define terms with bindings (Section 5): Is our abstraction "concrete" enough to support all the constructions and properties typically associated with syntax with bindings, including binding-aware datatypes? And can the constructions be performed in a modular fashion, allowing previous constructions to be reused for new ones? After undergoing a few more refinements, our binders pass the test of properly handling not only bound variables, but also free variables. This suggests that we may have identified a "sweet spot" between the assumptions and the guarantees involved in the construction of datatypes.

The binding-aware (co)datatypes would not be of much use without adequate reasoning and definitional principles that match their structure. A challenge is to develop principles that follow the spirit of Barendregt's variable convention [1984, p. 26]:

> If [the terms] $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

To this end, we generalize fresh induction in the style of nominal logic and propose new coinduction principles (Section 6). Moreover, we introduce (co)recursion principles that adhere to Barendregt's variable convention while improving on the state of the art even in the case of simple binders, and we validate the constructions by characterizing them up to isomorphism (Section 7).

A slightly restricted version of the definitions and theorems presented here have been mechanically checked in Isabelle/HOL (Section 8). In this paper, we focus on presenting the main ideas, and

refer to a technical report [Blanchette et al. 2019a] and the Isabelle formalization [Blanchette et al. 2019b] for more details, including formal proofs.

Our category-theoretic approach to bindings can be seen as a generalization of nominal logic (Section 9). Succinctly, we propose a BNF-based approach to bindings that subsumes and extends the syntactic formats in the literature, and that features capture-free substitutions as first-class citizens. By allowing infinite support, our framework supports non-well-founded and infinitely branching types that lie beyond nominal logic's reach. Our type definitions are equipped with (co)induction and (co)recursion principles implementing Barendregt's variable convention; they are expressed in a uniform way that insulates us from the complexity of bindings.

2 PRELIMINARIES

Our work has been developed in higher-order logic (HOL) and builds on bounded natural functors (BNFs), a category-theoretic approach to defining and reasoning about types in a modular fashion. HOL is a much weaker logic than classical Zermelo–Franekel set theory with the axiom of choice. The reader who prefers to think in terms of sets can ignore the syntactic aspects of HOL, mentally replacing "type" with "(nonempty) set," "type constructor" with "operator on sets," and "type variable" with either "fixed set" or "arbitrary set" as they read. The reader does not need to understand the precise HOL definition of various standard types, such as that of natural numbers and the powertype, but can assume they behave as expected. Similarly, our BNFs can be conceptualized as n-argument functors on the category of sets.

2.1 Higher-Order Logic

We consider classical higher-order logic with Hilbert choice, the axiom of infinity, and rank-1 polymorphism. HOL is based on simple type theory [Church 1940]. It is the logic of the original HOL system [Gordon and Melham 1993] and of HOL4, HOL Light, and Isabelle/HOL. In what follows, we give a brief description of HOL; the reader can consult the standard reference [Pitts 1993] or [Kunčar and Popescu 2018, Section 3] for a more compact description.

Primitive *types* are built from type variables $\alpha, \beta, \ldots$, a type *bool* of Booleans, and an infinite type *ind* using the function type constructor $\rightarrow$, i.e., by the grammar $T ::= \alpha \mid ind \mid bool \mid T \rightarrow T$. The primitive *constants* include equality $= : \alpha \rightarrow \alpha \rightarrow bool$, the Hilbert choice operator, and 0 and Suc for *ind*. Terms are built from constants and variables using typed λ-abstraction and application. The only mechanism for defining new types is *typedef*, which roughly corresponds to set comprehension in set theory: For any given type T and nonempty predicate $P : T \rightarrow bool$, we can carve out a new type $\{x : T \mid P\ x\}$ consisting of all members of T satisfying P. To lighten notation, we treat $\{x : T \mid P\ x\}$ as a first-class type expression, omitting the reference to typedef.

The type α *set* of sets over α (the powertype of α) is defined using typedef as (a copy of) $\alpha \rightarrow bool$. Other useful types, such as α *fset* for finite sets and α *mset* for multisets (bags), are also defined using typedef. The type *nat* of natural numbers, as well as many other (co)datatypes, are also derived concepts in HOL. Most HOL-based provers can automatically derive any uniformly and positively recursive ML-style datatype specified by their users, and Isabelle/HOL also supports codatatypes (Section 2.3). For example, the datatype *nat* is defined from the primitive type *ind* via typedef, by carving out the intersection of all sets that contain 0 and are closed under Suc.

A type T is called *polymorphic in* $\overline{\alpha} = (\alpha_1, \ldots, \alpha_n)$ if it contains these variables. To indicate polymorphism in $\overline{\alpha}$, we sometimes write $\overline{\alpha}\ T$ instead of T. An *instance* of a type is obtained by replacing some of its type variables with other types. For example, $(\alpha \rightarrow bool) \rightarrow \alpha$ is polymorphic in α, and $(ind \rightarrow bool) \rightarrow ind$ is one of its instances. A function is *polymorphic in* $\overline{\alpha}$ if its type is polymorphic in $\overline{\alpha}$. For example, the function Cons : $\alpha \rightarrow \alpha\ list \rightarrow \alpha\ list$ is polymorphic in α. Semantically, we think of polymorphic functions as families of functions, one for each type—for

example, the $\alpha := bool$ instance of Cons has type $bool \to bool\ list \to bool\ list$. *Formulas* are closed terms of type *bool*. Polymorphic formulas are thought of as universally quantified over their type variables. For example, $\forall x : \alpha.\ x = x$ really means $\forall \alpha.\ \forall x : \alpha.\ x = x$. To keep the discussion closer to informal mathematical practice, we sometimes use type variables as metavariables for types. For example, we may write: "Given any type α and any function $f : \alpha \to \alpha$, such and such holds."

2.2 Bounded Natural Functors

Often it is useful to think not in terms of polymorphic types, but in terms of type constructors. For example, *list* is a type constructor in one variable, whereas sum types (+) and product types (×) are binary type constructors. Most type constructors are not only operators on types but have a richer structure, that of *bounded natural functors* [Traytel et al. 2012]. Below, we write $[n]$ for $\{1, \ldots, n\}$.

Definition 1. Let $F = (F, \mathsf{map}_F, (\mathsf{set}^i_F)_{i \in [n]}, \mathsf{bd}_F)$, where

- F is an n-ary type constructor;
- $\mathsf{map}_F : (\alpha_1 \to \alpha'_1) \to \cdots \to (\alpha_n \to \alpha'_n) \to \overline{\alpha}\ F \to \overline{\alpha'}\ F$;
- $\mathsf{set}^i_F : \overline{\alpha}\ F \to \alpha_i\ set$ for $i \in [n]$;
- bd_F is an infinite cardinal number.

F's action on relations $\mathsf{rel}_F : (\alpha_1 \to \alpha'_1 \to bool) \to \cdots \to (\alpha_n \to \alpha'_n \to bool) \to \overline{\alpha}\ F \to \overline{\alpha'}\ F \to bool$ is defined by

(DefRel) $\mathsf{rel}_F\ \overline{R}\ x\ y \longleftrightarrow$
$\exists z.\ (\forall i \in [n].\ \mathsf{set}^i_F\ z \subseteq \{(a, a') \mid R_i\ a\ a'\}) \wedge \mathsf{map}_F\ [\mathsf{fst}]^n\ z = x \wedge \mathsf{map}_F\ [\mathsf{snd}]^n\ z = y$

where fst and snd are the standard first and second projection functions on the product type $\times$ and $\mathsf{map}_F\ [f]^n$ denotes the application $\mathsf{map}_F\ f\ \ldots\ f$ of map_F to n occurrences of f. F is an n-ary *bounded natural functor* if it satisfies the following properties:

(Fun) (F, map_F) is a functor (in all the n inputs)—i.e., map_F commutes with function composition and preserves the identities.

(Nat) each set^i_F is a natural transformation between the functor (F, map_F) and the powerset functor (set, image) (thinking of the latter as an n-input functor that operates on objects by taking $\overline{\alpha}$ to $\alpha_i\ set$); in other words, each set^i_F is assumed to be natural in all the n inputs.

(Cong) map_F only depends on the value of its argument functions on the elements of set^i_F—i.e., $(\forall i \in [n].\ \forall a \in \mathsf{set}^i_F\ x.\ \ f_i\ a = g_i\ a) \longrightarrow \mathsf{map}_F\ \overline{f}\ x = \mathsf{map}_F\ \overline{g}\ x$.

(Bound) The elements of set^i_F are bounded by bd_F—i.e., $\forall i \in [n].\ \forall x : \overline{\alpha}\ F.\ \ \left|\mathsf{set}^i_F\ x\right| < \mathsf{bd}_F$.

(Rel) (F, rel_F) is an n-ary relator—i.e., rel_F commutes with relation composition and preserves the equality relations.

Requiring that (F, rel_F) is a relator is equivalent to requiring that (F, map_F) preserves weak pullbacks [Rutten 1998]. It follows from the BNF axioms that the relator structure is an extension of the map function, in that mapping with a function f has the same effect as taking its graph $\mathsf{Gr}\ f$ and relating through $\mathsf{Gr}\ f$.

We regard the elements x of $\overline{\alpha}\ F$ as containers filled with content, where the content is provided by atoms in α_i. The set^i_F functions return the sets of α_i-atoms (which are bounded by bd_F). Moreover, it is useful to think of the map function and the relator in the following way:

- Applying $\mathsf{map}_F\ \overline{f}$ to x keeps x's container but updates its content as specified by $\overline{f}$, substituting $f_i\ a$ for each $a : \alpha_i$.
- For all $x : \overline{\alpha}\ F$ and $y : \overline{\beta}\ F$, $\mathsf{rel}_F\ \overline{R}\ x\ y$ if and only if x and y have the same containers and their content atoms corresponding to the same position in the container are related by R_i.

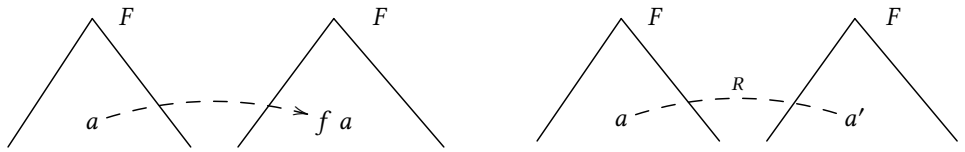

Fig. 1. $\mathsf{map}_F\ f$ (left) and $\mathsf{rel}_F\ R$ (right)

Consider a unary BNF F. For a fixed α, we represent a typical element of $x : \alpha\ F$ as depicted in Fig. 1, where the container is represented by a wedge and its content by a typical atom $a : \alpha$. The left-hand side shows how mapping $f : \alpha \to \alpha'$ amounts to replacing each a with $f\ a$. The right-hand side shows how the relator applied to $R : \alpha \to \alpha' \to \mathit{bool}$ states that each a is R-related to an a' located at the same position in the container. Consider the *list* type construct. It constitutes a unary BNF, where $\mathsf{map}_{\mathit{list}}$ is the standard map function, $\mathsf{set}_{\mathit{list}}$ collects all the elements of a list, $\mathsf{bd}_{\mathit{list}}$ is $\aleph_0$, and $\mathsf{rel}_{\mathit{list}}\ R\ \mathit{xs}\ \mathit{ys}$ states that xs and ys have the same length and are elementwise related by R.

We can also use the above intuition to explain the (DefRel) clause of Definition 1, which defines the relator in terms of the map and set operators: Given $x : \alpha\ F$ and $y : \alpha'\ F$, $\mathsf{rel}_F\ R$ holds for x and y if there exists $z : (\alpha \times \alpha')\ F$ that projects to both x and y (by mapping fst and snd, respectively) and has all its atoms $(a, b) : \alpha \times \alpha'$ related by R. This is a way of stating that x and y have the same shape (and also the same with z, thanks to their mapping connection with z) and the atoms placed at the same positions in this shape, a and a', are related by R.

This container intuition has been developed in work prior to BNFs, for various classes of functors [Abbott et al. 2005; Hoogendijk and de Moor 2000] (Section 9.2). The intuition will be crucial in allowing us to find a correct abstract notion of alpha-equivalence.

2.3 (Co)datatypes from Bounded Natural Functors

A *strong* BNF is a BNF whose map preserves not only weak pullbacks but also (ordinary) pullbacks. Strong BNFs include the basic type constructors of sum, product, and positive function space. Examples of BNFs that are not strong are the permutative (nonfree) type constructors, such as those of finite sets or bags. Both the BNFs and the strong BNFs are closed under composition and (least and greatest, possibly nonuniform) fixpoint definitions [Blanchette et al. 2017; Traytel et al. 2012]. This enables us to combine and nest BNFs arbitrarily when defining (co)datatypes.

Datatypes $\overline{\alpha}\ T$, where $\overline{\alpha}$ is a tuple of type variables of length m, written $\mathsf{len}\ \overline{\alpha} = m$, can be defined recursively from $(m+1)$-ary BNFs $(\overline{\alpha}, \tau)\ F$, by taking their least fixpoint (initial algebra): the minimal solution up to isomorphism of the recursive equation $\overline{\alpha}\ T \simeq (\overline{\alpha}, \overline{\alpha}\ T)\ F$. If we instead interpret the equation maximally—which we will indicate with the superscript ∞—we obtain the codatatype $\overline{\alpha}\ T$. In either case, the construction yields an $\overline{\alpha}$-polymorphic bijection $\mathsf{ctor} : (\overline{\alpha}, \overline{\alpha}\ T)\ F \to \overline{\alpha}\ T$.

Datatypes and codatatypes differ in their reasoning and definitional principles. For datatypes we have *structural induction*—which allows us to prove that a predicate holds on all its elements—and *recursion*—which allows us to define a function from the datatype to another type. Dually, for codatatypes we have *structural coinduction*—which allows us to prove that a binary relation is included in equality—and *corecursion*—which allows us to define functions from another type to the codatatype. Concretely, the difference can be understood in terms of well-foundedness: A datatype contains only well-founded entities, whereas a codatatype contains possibly non-well-founded ones. For example, if we take $(\alpha, \tau)\ F$ to be $\mathit{unit} + (\alpha \times \tau)$ (where *unit* is a fixed singleton type), the datatype defined by $\alpha\ T \simeq (\alpha, \alpha\ T)\ F$ is $\alpha\ \mathit{list}$, the type of (finite) lists. If instead we consider the maximal interpretation, $\alpha\ T \simeq^{\infty} (\alpha, \alpha\ T)\ F$, we obtain the codatatype of finite or infinite ("lazy")

lists, α *llist*. A substantial benefit of BNFs is that fixpoints can be freely nested. Since α *llist* is itself a BNF, it can be used in further fixpoint definitions: For $(\alpha, \tau)\, F := \alpha + \tau$ *llist*, the datatype $\alpha\ T \simeq (\alpha, \alpha\ T)\, F$ is the type of α-labeled well-founded infinitely branching rose trees.

3 TOWARDS AN ABSTRACT NOTION OF BINDER

The literature has so far focused on binding notions relying on syntactic formats. In contrast, here we ask a semantic question: Can we provide an abstract, syntax-free axiomatization of binders? We start by considering a few examples.

The paradigmatic example is the λ-calculus, in which λ-abstraction $\lambda a.\ t$ binds a single variable a in a single term t. The term t may contain several free variables. If a is one of them, adding the λ-abstraction binds it. Suppose t is the term $b\ a$ ("b applied to a"), where a and b are distinct variables. Applying the λ constructor to a and t yields $\lambda a.\ b\ a$, where a is now bound whereas b remains free. Thus, in a λ-binder we distinguish two main components: the binding variable and the body.

Other binders take into consideration a wider context than just the body. The "let" construct let $a = t_1$ in t_2 binds the variable a in the term t_2 without affecting t_1. In the expression let $a = b\ a$ in $b\ a$, the first occurrence of a is the binding occurrence, the second occurrence is free (i.e., not in the binder's scope), and the third occurrence is bound (i.e., in the binder's scope). In general, we must distinguish between the components that fall under a binder's scope and those that do not.

To further complicate matters, a single binding variable can affect multiple terms. The "let rec" construct let rec $a = t_1$ in t_2 binds the variable a simultaneously in the terms t_1 and t_2. In let rec $a = b\ a$ in $b\ a$, both the second and the third occurrences of a are bound by the first occurrence. Conversely, multiple variables can affect a single term. The expression $\lambda(a, b).\ t$ simultaneously binds the variables a and b in the term t. The binding relationship can also be many-to-many: let rec $a = t_1$ and $b = t_2$ in t simultaneously binds two variables in three terms.

Finally, the simultaneously binding variables can be organized in structures of arbitrary complexity. The "pattern let" binder in part 2B of the POPLmark challenge, pattern-let $p = t_1$ in t_2, allows binding patterns p in terms t_2, where the patterns are defined by the recursive grammar

$$p\ ::=\ a : T\ \mid\ \{l_i = p_i\}_{i=1}^{n}$$

Thus, a pattern is either a typed variable $a : T$ or, recursively, a record of labeled patterns.

Abstracting from the syntax of terms, we can think of binders as a mechanism for putting together binding variables and other entities, typically terms, which could be either inside or outside the scope of the binder. Following this view, a binder type corresponds to a type constructor $(\overline{\alpha}, \overline{\tau})\, F$, with $m = \mathsf{len}\ \overline{\alpha}$ and $n = \mathsf{len}\ \overline{\tau}$, that takes as inputs

- m types α_i of *binding variables*; and
- n types τ_i of *potential terms* that represent the context

together with a relation $\theta \subseteq [m] \times [n]$, which we call *binding dispatcher*, indicating which types of variables bind in which types of potential terms. A binder $x : (\overline{\alpha}, \overline{\tau})\, F$ can then be conceived as an arrangement of zero or more variables of each type α_i and zero or more potential terms of each type τ_i in a suitable structure. The actual binding takes place according to the binding dispatcher θ: If $(i, j) \in \theta$, all the variables of type α_i occurring in x bind in all the terms of type τ_j occurring in x. In examples, if $m = 1$ or $n = 1$, we will omit the subscript and simply write α or τ.

We use the terminology "potential terms" instead of simply "terms" to describe the inputs τ_i because they do not contain actual terms—they are simply placeholders in $(\overline{\alpha}, \overline{\tau})\, F$ indicating how terms would be treated by the binder F. The types of actual terms will be structures defined recursively as fixpoints by filling in the τ_i placeholders.

The examples above can be modeled as follows:

- For $\lambda a.\, t$, we take $m = n = 1$, $\theta = \{(1,1)\}$, and $(\alpha, \tau)\, F = \alpha \times \tau$.
- For let $a = t_1$ in t_2, we take $m = 1$, $n = 2$, $\theta = \{(1,2)\}$, and $(\alpha, \tau_1, \tau_2)\, F = \alpha \times \tau_1 \times \tau_2$.
- For let rec $a = t_1$ in t_2, we take $m = n = 1$, $\theta = \{(1,1)\}$, and $(\alpha, \tau)\, F = \alpha \times \tau \times \tau$.
- For $\lambda(a, b).\, t$, we take $m = n = 1$, $\theta = \{(1,1)\}$, and $(\alpha, \tau)\, F = \alpha \times \alpha \times \tau$.
- For let rec $a = t_1$ and $b = t_2$ in t, we take $m = n = 1$, $\theta = \{(1,1)\}$, and $(\alpha, \tau)\, F = \alpha \times \alpha \times \tau \times \tau \times \tau$.
- For pattern-let $p = t_1$ in t_2, we take $m = 1$, $n = 2$, $\theta = \{(1,2)\}$, and $(\alpha, \tau_1, \tau_2)\, F = \alpha\ \mathit{pat} \times \tau_1 \times \tau_2$, where $\alpha\ \mathit{pat}$ is the datatype defined recursively as $\alpha\ \mathit{pat} \simeq (\alpha \times \mathit{type}) + (\mathit{label}, \alpha\ \mathit{pat})\ \mathit{record}$, for *type* and *label* fixed types (as specified in the POPLmark challenge) and $(\beta_1, \beta_2)\ \mathit{record}$ the type constructor of β_1-labeled records with elements in β_2.

For the "let" binder, the type constructor $(\alpha, \tau_1, \tau_2)\, F$ must distinguish between the type of potential terms in the binder's scope, τ_2, and that of potential terms outside its scope, τ_1. This is necessary to describe the binder's structure accurately, but the actual terms corresponding to τ_1 and τ_2 will be allowed to be the same, as in $(\alpha, \tau, \tau)\, F$. Why should the binder care about potential terms that fall outside the scope of its binding variables? The answer is that this is often necessary, as pointed out by Pottier [2006]. In the "parallel let" construct let $a_1 = t_1$ and $\ldots$ and $a_n = t_n$ in t, the terms t_i are outside the scope of the variables a_i, but they must be considered as inputs for "let" to ensure that the number of terms t_i matches the number of variables a_i.

It could be argued that our proposal constitutes yet another restrictive format. However, leaving F unspecified gives considerable flexibility compared with the syntactic approach. F can incorporate arbitrarily complex binders, including the datatype $\alpha\ \mathit{pat}$ needed for the POPLmark "pattern let." It can also accommodate unforeseen situations. Capturing the "parallel let" construct above rests on the observation that the structure of binding variables can be intertwined with that of the out-of-scope potential terms, which a syntactic format would need to anticipate. By contrast, with the modular semantic approach, it suffices to choose a suitable type constructor: $(\alpha, \tau_1, \tau_2)\, F = (\alpha \times \tau_1)\ \mathit{list} \times \tau_2$, with $\theta = \{(1,2)\}$. As another example, the type schemes in Hindley–Milner type inference [Milner 1978] are assumed to have all the schematic type variables bound at the top level, but not in a particular order. A permutative type such as that of finite sets can be used: $(\alpha, \tau)\, F = \alpha\ \mathit{fset} \times \tau$, with $\theta = \{(1,1)\}$. In summary:

Proposal 1. *A binder type is a type constructor with a binding dispatcher on its inputs.*

As it stands, this proposal is not particularly impressive. For all its generality, it tells us nothing about how to construct actual terms with bindings or how to reason about them. Let us take a closer look and try to improve it. By modeling "binder types" not just as types but as type constructors, we can distinguish between the binder's structure (the shape) and the variables and potential terms that populate it (the content), following our intuition of BNFs (Section 2.2). And indeed, all the type constructors used so far would seem to be BNFs. So let us be more specific:

Proposal 2. *A binder type is a BNF with a binding dispatcher on its inputs.*

This would make our notion of binder type more versatile, given all the operations available on BNFs. In particular, we could use their map functions to perform renaming of bound variables, an essential operation for developing a theory of syntax with bindings. Moreover, complex binders could be constructed via the fixpoint operations on BNFs.

Unfortunately, there is a flaw. Full functoriality of $(\overline{\alpha}, \overline{\tau})\, F$ in the binding-variable components $\overline{\alpha}$ is problematic due to a requirement shared by many binders: *nonrepetitiveness* of the (simultaneously) binding variables. When we modeled the binder $\lambda(a, b).\, t$, which binds a and b in t, we take $(\alpha, \tau)\, F = \alpha \times \alpha \times \tau$. However, this is imprecise, because a and b must also be distinct. Similarly, $a_1, \ldots, a_n$ must be mutually distinct in let $a_1 = t_1$ and $\ldots$ and $a_n = t_n$ in t, and p may not have repeated variables in pattern-let $p = t_1$ in t_2.

This means that we must further restrict the type constructors to nonrepetitive items on the binding-variable components—for example, by taking $(\alpha, \tau)\, F$ to be $\{(a, b) : \alpha \times \alpha \mid a \neq b\} \times \tau$ instead of $\alpha \times \alpha \times \tau$. Unfortunately, the resulting type constructor is not a functor, since its map function cannot cope with noninjective functions $f : \alpha \to \alpha'$. If f identifies two variables that occur at different positions in $x : (\alpha, \tau)\, F$, then $\mathrm{map}_F\ f\ \mathrm{id}\ x$ would no longer be nonrepetitive; hence, it would not belong to $(\alpha', \tau)\, F$.

To address this issue, we refine the notion of BNF by restricting, on selected inputs, all conditions involving the map function, including the functoriality, to injective functions only. For symmetry, but also to avoid the bureaucracy of considering local inverses of functions, we take the more drastic measure of restricting the conditions to *bijective* functions only, which additionally have the same domain and codomain. We call these *endobijections*; they are also known as "permutations." All the BNF conditions (Definition 1) remain the same, except that on some of the inputs, marked as "restricted," they are further conditioned by endobijectiveness assumptions about the corresponding functions. For our type constructor $(\overline{\alpha}, \overline{\tau})\, F$, the restricted inputs will be $\overline{\alpha}$, meaning that F will behave like a functor with respect to endobijections $\overline{f} : \overline{\alpha} \to \overline{\alpha}$ and arbitrary functions $\overline{g} : \overline{\tau} \to \overline{\tau'}$. All our examples involving multiple variable bindings satisfy these weaker requirements. We call this notion *map-restricted BNF* (*MRBNF*, or $\overline{\alpha}$*-MRBNF*).

Proposal 3. *A binder type is a map-restricted BNF with a binding dispatcher on its inputs.*

MRBNFs remain general while offering a sound mechanism for renaming bound variables. To validate this proposal, we ask two questions, which will be answered in the next sections: How can nonrepetitive MRBNFs be constructed from possibly repetitive ones? How can MRBNFs be used to define and reason about actual terms with bindings and their basic operators?

4 CONSTRUCTING NONREPETITIVE MAP-RESTRICTED BNFS

To construct arbitrarily complex BNFs, we can start with the basic BNFs and repeatedly apply composition, least fixpoint (datatype), and greatest fixpoint (codatatype). Any BNF also constitutes a map-restricted BNF, and it is in principle possible to lift the map-restricted arguments through fixpoints on the nonrestricted type arguments. However, nonrepetitiveness is not closed under fixpoints. Thus, if $(\alpha, \tau)\, F$ is a nonrepetitive α-MRBNF, the (least or greatest) fixpoint $\alpha\ T$ specified as $(\alpha, \alpha\ T)\, F \simeq \alpha\ T$ will be an α-MRBNF, but not necessarily a nonrepetitive one. For example, $(\alpha, \tau)\, F = \mathit{unit} + (\alpha \times \tau)$ is a nonrepetitive α-MRBNF (because α atoms cannot occur multiple times in members of $(\alpha, \tau)\, F$), but its least and greatest fixpoints are $\alpha\ \mathit{list}$ and $\alpha\ \mathit{llist}$, the types of list and lazy lists, and these are repetitive α-MRBNFs because they may contain duplicate elements.

This means that complex nonrepetitive MRBNFs cannot be built recursively from simpler components. But there is an alternative: We can employ the fixpoint constructions on BNFs, and as a last step carve out nonrepetitive MRBNFs from BNFs, by taking the subset of items whose atoms of selected type arguments are nonrepetitive. For example, from the BNF $\alpha\ \mathit{list}$, we construct the MRBNF of nonrepetitive lists: $\{\mathit{xs} : \alpha\ \mathit{list} \mid \mathrm{nonrep}_{\mathit{list}}\ \mathit{xs}\}$. Similarly, from the "pattern let" BNF $\alpha\ \mathit{pat}$ (built recursively from the BNF $\alpha \times \mathit{type} + (\mathit{label}, \beta)\ \mathit{record}$), we construct the MRBNF of nonrepetitive patterns: $\{\mathit{xs} : \alpha\ \mathit{pat} \mid \mathrm{nonrep}_{\mathit{pat}}\ \mathit{xs}\}$. In both examples, we have a clear intuition for what it means to be a nonrepetitive member of the given BNF: A list xs is nonrepetitive, written $\mathrm{nonrep}_{\mathit{list}}\ \mathit{xs}$, if no α atom occurs more than once in it; and similarly for the members of $p : \alpha\ \mathit{pat}$, which are essentially trees whose leaf nodes are labeled with α atoms.

Can we express nonrepetitiveness generally for any BNF? A first idea is to rely on the cardinality of sets of atoms. For the $\alpha\ \mathit{list}$ BNF, the nonrepetitive items are those lists $as = [a_1, \ldots, a_n]$ containing precisely n distinct elements $a_1, \ldots, a_n$—or, equivalently, having a maximal cardinality of atoms, $|\mathrm{set}_{\mathit{list}}\ x|$, among the lists of a given length. This idea can be generalized to arbitrary

BNFs $\alpha\ F$ by observing that the length of a list fully characterizes its shape. We say that two members x, x' of $\alpha\ F$ have the same shape, written $\mathsf{sameShape}_F\ x\ x'$, if $\mathsf{rel}_F\ \top\ x\ x'$ holds, where $\top : \alpha \to \alpha \to \mathit{bool}$ is the vacuously true relation that ignores the content. Recall from Section 2.2 that the main intuition behind a BNF relator rel_F is that $\mathsf{rel}_F\ R\ x\ x'$ holds if and only if x and x' have the same shape and their atoms are positionwise related by R. The second condition is trivially satisfied for $R := \top$. For lists and lazy lists, sameShape means "same length," and for various kinds of trees it means that the two trees become identical if we erase their labels.

We could define $\mathsf{nonrep}_F\ x$ to mean that, for all x' such that $\mathsf{sameShape}_F\ x\ x'$, $|\mathsf{set}_F\ x'| \leq |\mathsf{set}_F\ x|$. This works for finitary BNFs such as lists and finitely branching well-founded trees, but fails for infinitary ones. For example, a lazy list $as = [1, 1, 2, 2, \ldots] : \mathit{nat\ stream}$ has $|\mathsf{set}_{\mathit{llist}}\ as|$ of maximal cardinality, and yet it is repetitive. We need a more abstract approach. An essential property of the nonrepetitive lists $as = [a_1, \ldots, a_n]$ is their ability to *pattern-match* any other list $as' = [a'_1, \ldots, a'_n]$ of the same length n; and the pattern-matching process yields the function f that sends each a_i to a'_i (and leaves the shape unchanged), where f achieves the overall effect that it *maps as* to *as'*.

In general, for $x : \alpha\ F$, we define $\mathsf{nonrep}_F\ x$ so that for all x' such that $\mathsf{sameShape}_F\ x\ x'$, there exists a function f that maps x to x', meaning that $x' = \mathsf{map}_F\ f\ x$. This works for lists, lazy lists, trees, and any other combination of (co)datatypes where each atom has a fixed position in the shape—i.e., strong BNFs. We can define the corresponding nonrepetitive MRBNF:

Theorem 2. If $\alpha\ F$ is a strong BNF and nonrep_F is nonempty, then $\alpha\ G = \{x : \alpha\ F \mid \mathsf{nonrep}_F\ x\}$, in conjunction with the corresponding restrictions of map_F, set_F, rel_F, and bd_F, forms an MRBNF.

This construction works for any n-ary BNF $\overline{\alpha}\ F$, which can be restricted to nonrepetitive members with respect to any of its strong inputs α_i, and more generally to any $\overline{\alpha}$-MRBNF $(\overline{\alpha}, \overline{\tau})\ F$, which can be further restricted with respect to any of its unrestricted strong inputs τ_i.

We introduce the notation $(\overline{\alpha}, \overline{\tau})\ F\ @\ \tau_i$ to indicate such further restricted nonrepetitive MRBNFs. For example, we write $\alpha\ \mathit{list}\ @\ \alpha$ for the α-MRBNF of nonrepetitive lists over α, and $(\alpha \times \beta)\ \mathit{list}\ @\ \alpha$ for the α-MRBNF of lists of pairs in $\alpha \times \beta$ that do not have repeated occurrences of the first component. Thus, given $a, a' : \alpha$ with $a \neq a'$ and $b, b' : \beta$ with $b \neq b'$, the type $(\alpha \times \beta)\ \mathit{list}\ @\ \alpha$ contains the list $[(a, b), (a', b)]$ but not the list $[(a, b), (a, b')]$.

For BNFs that are not strong, such as finite sets and multisets, the nonrepetitiveness construction tends to give counterintuitive results; for example, no finite set but the empty one is nonrepetitive. Nevertheless, these structures are useful as they are, without any nonrepetitiveness restriction. For example, Hindley–Milner type schemes bind finite sets of variables.

In summary, our nonrepetitiveness construction removes all the possible variable duplication introduced by fixpoint definitions. The construction reflects informal practice. For example, the POPLmark recursive "let" patterns are defined inductively, and then it is stated that the variables and the labels should not be repetitive. We capture this informal practice under a uniform concept, applicable to a large class of (binders modeled as) functors.

5 DEFINING TERMS WITH BINDINGS VIA MAP-RESTRICTED BNFS

So far, we have modeled binders as $\overline{\alpha}$-MRBNFs $(\overline{\alpha}, \overline{\tau})\ F$, with $m = \mathsf{len}\ \overline{\alpha}$, $n = \mathsf{len}\ \overline{\tau}$, together with a binding dispatcher $\theta \subseteq [m] \times [n]$. We think of each α_i as a type of variables, of each τ_j as a type of potential terms, and of $(i, j) \in \theta$ as indicating that α_i variables are binding in τ_j terms.

Before we can define actual terms, we must prepare for a dual phenomenon to the binding of variables: The terms must be allowed to have *free* variables in the first place, before these can be bound. Thus, in addition to binding mechanisms, we need mechanisms to inject free variables into potential terms. This can be achieved by upgrading F: Instead of $(\overline{\alpha}, \overline{\tau})\ F$, we work with $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F$, where we consider an additional vector of inputs $\overline{\beta}$ representing the types of (injected) free variables.

It is natural to consider the same types of variables as possibly free and possibly bound. Hence, we will assume $\mathsf{len}\ \overline{\beta} = \mathsf{len}\ \overline{\alpha} = m$ and use $(\overline{\alpha}, \overline{\alpha}, \overline{\tau})\, F$ when defining actual terms. Nevertheless, it is important to allow F to distinguish between the two—this distinction will affect the central notions of free variable and alpha-equivalence.

Actual terms can be defined by means of a datatype construction framed by F. For simplicity, we will define a single type of terms in which all the types of variables α_i can be bound, which means assuming that all potential term types τ_i are equal. This is achieved by taking the following datatype $\overline{\alpha}\ T$ of F-framed terms with variables in $\overline{\alpha}$:

$$\overline{\alpha}\ T \ \simeq\ (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ T]^n)\, F$$

where $[\overline{\alpha}\ T]^n$ denotes the tuple consisting of n occurrences of $\overline{\alpha}\ T$. The fully general case, of multiple (mutually recursive) term types, is a straightforward but technical generalization.

Example 3. Consider the syntax of the λ-calculus, where the collection $\alpha\ T$ of terms t with variables in α are defined by the grammar

$$t \ ::=\ \mathsf{Var}\, a \ \mid\ \lambda a.\, t \ \mid\ t\, t$$

A term is either a variable, an abstraction, or an application. This is supported by taking $m = 1$, $n = 2$, $\theta = \{(1, 1)\}$, and

$$(\beta, \alpha, \tau_1, \tau_2)\, F = \beta + (\alpha \times \tau_1) + (\tau_2 \times \tau_2)$$

The resulting $\alpha\ T$ satisfies the recursive equation

$$\alpha\ T \ \simeq\ \alpha + (\alpha \times \alpha\ T) + (\alpha\ T \times \alpha\ T)$$

Not visible in this equation is how F distinguishes (1) between the free-variable type β and the binding-variable type α and (2) between two different types of potential terms, τ_1 and τ_2. The first distinction ensures that the occurrence of α as the first summand stands for an injection of free variables, whereas the first occurrence of α in the second summand stands for binding variables. The second distinction ensures, in conjunction with θ, that α's binding powers extend to the occurrence of $\alpha\ T$ in the second summand but not to the two occurrences in the third summand. This additional information is needed for the proper treatment of the bindings.

The functor F both binds variables and injects free variables. Despite this dual role, we will call F a binder type. Multiple binding or free-variable injecting operators can be handled simultaneously by defining F appropriately.

Example 4. Consider the extension of the λ-calculus syntax with "parallel let" binders:

$$t \ ::=\ \cdots \ \mid\ \mathsf{let}\ a_1 = t_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ a_n = t_n\ \mathsf{in}\ t$$

We can add a further summand, $((\alpha \times \tau_2)\ \mathit{list} \times \tau_1)\ @\ \alpha$, to the previous definition of $(\beta, \alpha, \tau_1, \tau_2)\, F$. The choice of the type variables in $(\alpha \times \tau_2)\ \mathit{list} \times \tau_1$, in conjunction with θ's relating α with τ_1 but not with τ_2, indicates that the term t, but not the terms t_i, is in the scope of the binding variables a_i.

To summarize, we have extended the MRBNF F with a further vector of inputs, $\overline{\beta}$. The new functor $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F$ has the following inputs:

- $\overline{\beta}$ are types of free variables;
- $\overline{\alpha}$ are types of binding variables;
- $\overline{\tau}$ are types of potential terms, which are made into actual terms when defining the datatype $\overline{\alpha}\ T$ as $\overline{\alpha}\ T \simeq (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ T]^n)\, F$.

We have assumed F to be an unrestricted functor on $\overline{\tau}$, and to be a functor on $\overline{\alpha}$ with respect to endobijections. (We qualify as "unrestricted" a functor whose morphism component operates on arbitrary functions as opposed to restricted types of functions.) But how should F behave on $\overline{\beta}$? A natural answer would be to require unrestricted functoriality, because the nonrepetitiveness condition that compelled us to restrict F's behavior on binding variable inputs seems unnecessary here: There is no apparent need to avoid repeated occurrences of free variables. In fact, the central operation of substitution introduces repetitions—e.g., by substituting a' for a in a term that already contains a free occurrence of a'. For now, we will assume unrestricted functoriality on $\overline{\beta}$.

Proposal 4. *A binder type is a map-restricted BNF that*

- *distinguishes between free-variable, binding-variable, and potential term inputs, and*
- *puts the map restriction on the binding-variable inputs only,*

together with a binding dispatcher between the binding-variable inputs and the potential term inputs.

Let us see if this is sufficient to support fundamental constructions on terms. A small running example, exhibiting enough binding diversity, will keep us company.

Example 5. Consider a λ-calculus variant in which abstractions simultaneously bind two variables in two terms, given by the grammar

$$t \ ::= \ \mathsf{Var}\ a \ \mid \ \lambda(a, b).\,(t_1, t_2)$$

with the usual requirement that the variables a and b are distinct. We can take $\theta = \{(1, 1)\}$ and

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

We write Inl and Inr for the left and right injections of the components into sums types: $\mathsf{Inl} : \beta \to (\beta, \alpha, \tau)\,F$ and $\mathsf{Inr} : ((\alpha \times \alpha)\ @\ \alpha) \times \tau \times \tau \to (\beta, \alpha, \tau)\,F$.

5.1 Free Variables

Any element $t : \overline{\alpha}\ T$ can be written as $\mathsf{ctor}\ x$, where $x : (\overline{\alpha}, : \overline{\alpha}, : [\overline{\alpha}\ T]^n)\,F$ has three kinds of atoms:

- the (i-)*top-free variables*, $\mathsf{topFree}_i\ x : \alpha_i$, are elements of $\mathsf{set}^F_i\ x$ for $i \in [m]$ representing the free variables injected by the topmost constructor of t;
- the (i-)*top-binding variables*, $\mathsf{topBind}_i\ x : \alpha_i$, are elements of $\mathsf{set}^F_{m+i}\ x$ for $i \in [m]$ representing the binding variables introduced by the topmost constructor of t;
- the (j-)*recursive components*, $\mathsf{rec}_j\ x : \overline{\alpha}\ T$, are elements of $\mathsf{set}^F_{2m+j}\ x$ for $j \in [n]$.

To refer precisely to the scope of bindings in light of the binding dispatcher θ, for each $i \in [m]$ and $j \in [n]$ we define $\mathsf{topBind}_{i,j}\ x$ to be $\mathsf{topBind}_i\ x$ if $(i, j) \in \theta$ and $\varnothing$ otherwise. We can think of $\mathsf{topBind}_{i,j}\ x$ as the top-binding variables that are actually binding in all of the $\mathsf{rec}_j\ x$ components, simultaneously. Since $\mathsf{topBind}_{i,j}$ incorporates the information provided by θ, the latter will be left implicit in our forthcoming constructions.

Equipped with these notations, we can define a free variable of a term to be either a top-free variable or, recursively, a free variable of some recursive component that is not among the relevant top-binding variables. Formally, $\mathsf{FVars}_i\ t$ for $i \in [m]$ is defined inductively by the following rules:

$$\frac{a \in \mathsf{topFree}_i\ x}{a \in \mathsf{FVars}_i\ (\mathsf{ctor}\ x)} \qquad \frac{t \in \mathsf{rec}_j\ x \quad a \in \mathsf{FVars}_i\ t \setminus \mathsf{topBind}_{i,j}\ x}{a \in \mathsf{FVars}_i\ (\mathsf{ctor}\ x)}$$

In the context of our running Example 5, let us assume from now on that $a, b, c, d, \ldots$ are mutually distinct variables. Consider the term $t = \mathsf{Var}\ c$. It can be written as $\mathsf{ctor}\ x$, where $x = \mathsf{Inl}\ c$. Therefore, $\mathsf{topFree}\ x = \{c\}$, $\mathsf{topBind}\ x = \varnothing$, and $\mathsf{rec}\ x = \varnothing$. (We omit the subscripts since $m = n = 1$.) Thus, t has c as its single top-free variable, has no top-binding variables, and has no recursive components.

Moreover, c is the only free variable in t: Applying the first rule in the definition of FVars, we infer $c \in \mathsf{FVars}\ (\mathsf{ctor}\ x)$ from $c \in \mathsf{topFree}\ x$.

Now consider the term $t = \lambda(a, b).\ (\mathsf{Var}\ a, \mathsf{Var}\ c)$. It can be written as $t = \mathsf{ctor}\ x$, where $x = \mathsf{Inr}\ ((a, b), (\mathsf{Var}\ a, \mathsf{Var}\ c))$. We have $\mathsf{topFree}\ x = \varnothing$, $\mathsf{topBind}\ x = \{a, b\}$, and $\mathsf{rec}\ x = \{\mathsf{Var}\ a, \mathsf{Var}\ c\}$. In other words, t has no top-free variables, has a and b as top-binding variables, and has $\mathsf{Var}\ a$ and $\mathsf{Var}\ c$ as recursive components. Moreover, t has c as its single free variable: Applying the second rule in the definition of FVars, we infer $c \in \mathsf{FVars}\ (\mathsf{ctor}\ x)$ from $\mathsf{Var}\ c \in \mathsf{rec}\ x$ and $c \in \mathsf{FVars}\ (\mathsf{Var}\ c) \setminus \mathsf{topBind}\ x = \{c\} \setminus \{a, b\} = \{c\}$ using $(1, 1) \in \theta$.

5.2 Alpha-Equivalence

To express alpha-equivalence, we first need to define the notion of renaming the variables of a term via m bijections $\overline{f}$. This can be achieved using by the map function of $\overline{\alpha}\ T$, defined recursively as

$$\mathsf{map}_T\ \overline{f}\ (\mathsf{ctor}\ x)\ =\ \mathsf{ctor}\ (\mathsf{map}_F\ \overline{f}\ \overline{f}\ [\mathsf{map}_T\ \overline{f}]^n\ x)$$

Thus, $\mathsf{map}_T\ \overline{f}$ applies $\overline{f}$ to the top-binding and top-free variables of any term $\mathsf{ctor}\ x$, and calls itself recursively for the recursive components. The overall effect is the application of $\overline{f}$ to all the variables (free or not) of a term.

Intuitively, two terms t_1 and t_2 should be alpha-equivalent if they are the same up to a renaming of their bound variables. More precisely, the situation is as follows (Fig. 2):

- Their top-free variables (marked in the figure as a_1 and a_2) are positionwise equal.
- The top-binding variables of one (marked as a_1') are positionwise renamed into the top-binding variables of the other (marked as a_2'), e.g., by a bijection f_i.
- The results of correspondingly (i.e., via $\overline{f}$) renaming the recursive components of t_1 are positionwise alpha-equivalent to the recursive components of t_2. In symbols, we will express this fact as $\mathsf{map}_T\ \overline{f}\ t_1 \equiv t_2$.

Relators can be used to express positionwise correspondences such as the above (Section 2.2). Formally, we define the (infix) alpha-equivalence relation $\equiv : \alpha\ T \to \alpha\ T \to \mathit{bool}$ inductively by

$$\frac{\mathsf{rel}_F\ [(=)]^m\ (\mathsf{Gr}\ f_1)\ \ldots\ (\mathsf{Gr}\ f_m)\ [(\lambda t_1, t_2.\ \mathsf{map}_T\ \overline{f}\ t_1 \equiv t_2)]^n\ x_1\ x_2 \quad \mathsf{cond}_1\ f_1 \quad \cdots \quad \mathsf{cond}_m\ f_m}{\mathsf{ctor}\ x_1 \equiv \mathsf{ctor}\ x_2}$$

The first hypothesis is the inductive one. It employs the relator rel_F to express how the three kinds of atoms of x_1 and x_2 must be positionwise related: by equality for the top-free variables, by the graph of the m renaming functions f_i for the top-binding variables, and by alpha-equivalence after renaming with $\overline{f}$ for the recursive components. The second hypothesis is a condition on the f_i's. Clearly, $f_i : \alpha_i \to \alpha_i$ must be a bijection (written $\mathsf{bij} : (\alpha \to \alpha) \to \mathit{bool}$), to avoid collapsing top-binding variables. Moreover, f_i should not be allowed to change the free variables of the recursive components t_1 that are not captured by the top-binding variables. We thus take $\mathsf{cond}_i\ f_i$ to be

$$\mathsf{bij}\ f_i\ \wedge\ \forall a \in \big(\bigcup_{j \in [n]} (\bigcup_{t_1 \in \mathsf{rec}_j\ x_1} \mathsf{FVars}\ t_1) \setminus \mathsf{topBind}_{i,j}\ x_1\big).\ f_i\ a = a$$

Returning to Example 5, we note that $\mathsf{Var}\ a \equiv \mathsf{Var}\ a$ for every a. This is shown by applying the definitional clause of $\equiv$ with the identity for f. The first hypothesis can be immediately verified: Since $\mathsf{Var}\ a$ has no top-binding variables or recursive components, only the condition concerning the top-free variables needs to be checked: $a = a$.

Now consider the terms $t_1 = \lambda(a, b).\ (\mathsf{Var}\ a, \mathsf{Var}\ c)$ and $t_2 = \lambda(b, a).\ (\mathsf{Var}\ b, \mathsf{Var}\ c)$, which can be written as $\mathsf{ctor}\ x_1$ and $\mathsf{ctor}\ x_2$, where $x_1 = \mathsf{Inr}((a, b), (\mathsf{Var}\ a, \mathsf{Var}\ c))$ and $x_2 = \mathsf{Inr}((b, a), (\mathsf{Var}\ b, \mathsf{Var}\ c))$. We can prove their alpha-equivalence by taking f to swap a and b (i.e., send a to b and b to a) and leaving all the other variables unchanged. Verifying the first hypothesis of $\equiv$'s definitional

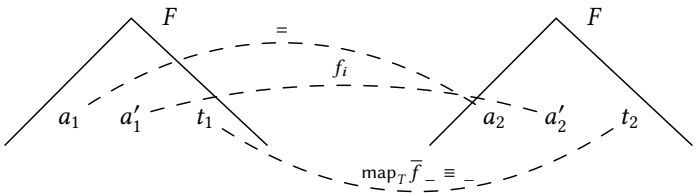

Fig. 2. Alpha-equivalence

rule amounts to the following: Concerning positionwise equality of the top-free variables, there is nothing to check (since t_1 and t_2 have none); concerning the top-binding variables, we must check that $f\ a = b$ and $f\ b = a$; concerning the recursive components, we must check that $\mathsf{map}_T\,f\,(\mathsf{Var}\ a) \equiv \mathsf{Var}\ b$ and $\mathsf{map}_T\,f\,(\mathsf{Var}\ c) \equiv \mathsf{Var}\ c$. Applying the definition of map_T, the last equivalences become $\mathsf{Var}\ (f\ a) \equiv \mathsf{Var}\ b$—i.e., $\mathsf{Var}\ b \equiv \mathsf{Var}\ b$ and $\mathsf{Var}\ c \equiv \mathsf{Var}\ c$. Finally, verifying the second hypothesis, cond f, amounts to checking that f is bijective and that f is the identity on all variables in $(\bigcup_{t_1 \in \{\mathsf{Var}\ a, \mathsf{Var}\ c\}} \mathsf{FVars}\ t_1) \setminus \{a, b\} = \{a, c\} \setminus \{a, b\} = \{c\}$—i.e., f sends c to c.

There are other approaches to define alpha-equivalence. We could pose stricter conditions on the functions f_i, allowing them to change only the top-binding variables, but no other (nonfree) variables occurring in the components. In the running example, if the left term is $\lambda(a, b).\ (\mathsf{Var}\ c,\ \lambda(c, d).\ (\mathsf{Var}\ c, \mathsf{Var}\ d))$, then both approaches allow f to change a and b, and forbid it to change c. Our definition additionally allows f to change d. Another alternative would consist in a symmetric formulation: Rather than renaming variables of the left term only, we could rename the variables of both terms to a third term, whose binding variables are all distinct from those of the first two. All these variants have different virtues in terms of the ease or elegance of proving various basic properties, but they produce the same concept of alpha-equivalence.

For any MRBNF F, we can prove the following crucial properties of alpha-equivalence. All these results follow by either rule induction on the definition of ≡ or structural induction on $\overline{\alpha}\ T$.

Theorem 6. Alpha-equivalence is an equivalence and is compatible with

- the term constructor, in that $\mathsf{rel}_F\ [(=)]^m\ [(=)]^m\ [(\equiv)]^n\ x_1\ x_2$ implies $\mathsf{ctor}\ x_1 \equiv \mathsf{ctor}\ x_2$ for all $x_1, x_2 : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ T]^n)\ F$;
- the free-variable operators, in that $t_1 \equiv t_2$ implies $\mathsf{FVars}_i\ t_1 = \mathsf{FVars}_i\ t_2$ for all $i \in [m]$;
- the map function of T, in that, if $f_i : \alpha \to \alpha$ for $i \in [m]$ are (endo)bijections, then $t_1 \equiv t_2$ implies $\mathsf{map}_T\ \overline{f}\ t_1 \equiv \mathsf{map}_T\ \overline{f}\ t_2$.

5.3 Alpha-Quotiented Terms

Exploiting Theorem 6, we can define the quotient $\overline{\alpha}\ \boldsymbol{T} = (\overline{\alpha}\ T)/{\equiv}$, and lift the relevant functions to $\overline{\alpha}\ \boldsymbol{T}$. Using overloaded notation, we obtain the constructor $\mathsf{ctor} : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\ F \to \overline{\alpha}\ \boldsymbol{T}$ and the operators $\mathsf{FVars} : \overline{\alpha}\ \boldsymbol{T} \to \alpha\ \mathit{set}$ and $\mathsf{map}_{\boldsymbol{T}} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$.

Whereas on $\overline{\alpha}\ T$ the constructor is bijective, on $\overline{\alpha}\ \boldsymbol{T}$ it is only surjective. Quotienting allows us to obtain equal results even if we bind different variables in different terms. In our running example, as $\alpha\ \boldsymbol{T}$ terms, $\lambda(a, b).\ (\mathsf{Var}\ a, \mathsf{Var}\ c)$ and $\lambda(b, a).\ (\mathsf{Var}\ b, \mathsf{Var}\ c)$ are actually equal, since $\mathsf{ctor}\ x_1 = \mathsf{ctor}\ x_2$, where $x_1 = \mathsf{Inr}\ ((a, b), (\mathsf{Var}\ a, \mathsf{Var}\ c))$ and $x_2 = \mathsf{Inr}\ ((b, a), (\mathsf{Var}\ b, \mathsf{Var}\ c))$; but $x_1 \neq x_2$.

Nevertheless, the quotient type enjoys injectivity up to a renaming, which follows from the definition of α, its compatibility with ctor and map_T, and the properties of relators.

Proposition 7. Given $x_1, x_2 : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\ F$, we have $\mathsf{ctor}\ x_1 = \mathsf{ctor}\ x_2$ if and only if there exist functions $f_i : \alpha_i \to \alpha_i$ satisfying $\mathsf{cond}_i\ f_i$ for $i \in [m]$ such that $x_2 = \mathsf{map}_F\ [\mathsf{id}]^m\ \overline{f}\ [\mathsf{map}_{\boldsymbol{T}}\ \overline{f}]^n\ x_1$.

In summary, $\overline{\alpha}\ \boldsymbol{T}$ is defined as a fixpoint framed by F followed by a quotienting construction with respect to alpha-equivalence determined by the binding dispatcher θ, which for $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F$ states what $\overline{\alpha}$ binds in $\overline{\tau}$. We will use the suggestive notation

$$\alpha\ \boldsymbol{T} \simeq_\theta (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\, F$$

to express the definition of this binding-aware datatype, where the θ subscript emphasizes that we have an isomorphism up to the alpha-equivalence notion determined by θ.

The presence of the operators ctor, FVars, and $\mathsf{map}_{\boldsymbol{T}}$ on quotiented terms offers them a large degree of independence from the underlying terms. Indeed, one of our main design goals is to develop an abstraction layer for reasoning about the type $\overline{\alpha}\ \boldsymbol{T}$ that allows us to forget about $\overline{\alpha}\ T$.

Convention 8. From now on, we will call the members of $\overline{\alpha}\ \boldsymbol{T}$ *terms* and the members of the underlying type $\overline{\alpha}\ T$ *raw terms*.

5.4 Infinitely Branching Terms

Our constructions do not commit to the finite branching of terms, thus capturing situations required by some calculi [Milner 1989] and logics [Hennessy and Milner 1980; Keisler 1971].

Example 9. A simplified version of the syntax for processes in the calculus of communicating systems (CCS) [Milner 1989] is the following, where c ranges over a fixed type C of channels, e over a type $\alpha\ E$ of arithmetic expressions with variables in α, and J over subsets of a fixed, possibly infinite type I of indices:

$$p ::= c\,(a).\,p \mid \overline{c}\,e.\,p \mid \textstyle\sum_{i \in J} p_i$$

Above, $c(a)\,.\,p$ is an input-prefix process with the binding variable a modeling the receiving of data on channel c, and $\overline{c}\,e.\,p$ is an output-prefixed process that starts by sending the value of expression e on c. The sum constructor models nondeterministic choice from a variety of possible continuations $(p_i)_{i \in J}$. The terminating process 0 is defined as an empty sum.

In our framework, $\alpha\ E$ is a standard datatype BNF and the type $\alpha\ \boldsymbol{T}$ of process terms can be defined as the binding-aware datatype given by $m = 1$, $n = 2$, $\theta = \{(1, 1)\}$, and

$$(\beta, \alpha, \tau_1, \tau_2)\, F = (C \times \alpha \times \tau_1) + (C \times \beta\ E \times \tau_2) + (I \to (\tau_2 + \mathit{unit}))$$

where we have modeled functions from subsets of I to τ_2 as functions from I to $\tau_2 + \mathit{unit}$. Hence, as desired, $\alpha\ \boldsymbol{T}$ satisfies the recursive equation

$$\alpha\ \boldsymbol{T} \simeq_\theta (C \times \alpha \times \alpha\ \boldsymbol{T}) + (C \times \alpha\ E \times \alpha\ \boldsymbol{T}) + (I \to (\alpha\ \boldsymbol{T} + \mathit{unit}))$$

where the notion of alpha-equivalence induced by θ asks that in the first summand the α variables bind in their neighboring terms.

5.5 Substitution

An important operation we want to support on terms $\overline{\alpha}\ \boldsymbol{T}$ is capture-avoiding substitution. With our current infrastructure, we would hope to be able to define *simultaneous substitution of variables for variables*, $\mathsf{sub} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$. It would take m functions $f_i : \alpha_i \to \alpha_i$ and a term t and return a term obtained by substituting in t, in a capture-avoiding fashion, all its free variables a with $f_i\ a$. Substitution with injective functions f_i is customarily called "renaming." Moreover, unary substitution is a particular case of simultaneous substitution, defined as $t[a/b] = \mathsf{sub}\ f_{a,b}\ t$, where $f_{a,b}$ sends b to a and all other variables to themselves.

A candidate for sub that suggests itself is $\mathsf{map}_{\boldsymbol{T}}$, which $\overline{\alpha}\ \boldsymbol{T}$ inherits from $\overline{\alpha}\ T$. However, this operator is not suitable, since it works only with bijections f_i. The fundamental desired property of

sub concerns its recursive behavior on terms of the form ctor x:

$$(\forall i \in [m].\ \mathsf{topBind}_i\ x \cap \mathsf{FVars}_i\ (\mathsf{ctor}\ x) = \varnothing \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing) \\ \longrightarrow\ \mathsf{sub}\ \overline{f}\ (\mathsf{ctor}\ x) = \mathsf{ctor}\ (\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\mathsf{sub}\ \overline{f}]^n\ x) \qquad (*)$$

where $\mathsf{supp}\ f_i$, the *support* of f_i, is defined as the union of $A_i = \{a : \alpha_i \mid f_i\ a \neq a\}$ and $\mathsf{image}\ f_i\ A_i$ (A_i's image through f_i). Thus, $\mathsf{sub}\ \overline{f}$ would distribute over the term constructor when neither the term's free variables nor the variables affected by f_i overlap with the top-binding variables.

The first of these two conditions, $\mathsf{topBind}_i\ x \cap \mathsf{FVars}_i\ (\mathsf{ctor}\ x) = \varnothing$, which we will abbreviate to $\mathsf{nonClash}_i\ x$, is an instance of Barendregt's variable convention. All our proof and definition principles observe it; we never consider variables that appear both bound and free in the same term. The condition is vacuously true for syntaxes in which no constructor simultaneously binds variables and introduces free variables.

To satisfy the two conditions, it must be possible to replace any binding variables in x that belong to the offending sets $\mathsf{FVars}_i\ (\mathsf{ctor}\ x)$ or $\mathsf{supp}\ f_i$ with variables from outside these sets, resulting in x'. This replacement would be immaterial as far as the input term ctor x is concerned: Alpha-equivalence being equality on $\alpha\ \boldsymbol{T}$, we would have ctor x' = ctor x. By this argument, it would be legitimate to assume the premise of $(*)$ whenever we apply substitution. However, there is the issue that $\mathsf{FVars}_i\ (\mathsf{ctor}\ x) \cup \mathsf{supp}\ f_i$ may be too large; indeed, it may even exhaust the entire type α_i. We need a mechanism to ensure that enough fresh variables are available.

5.6 Fresh Variable Acquisition

An advantage of our functorial setting is that the collection of variables $\overline{\alpha}$ is not a priori fixed. Since the functor F that underlies $\overline{\alpha}\ T$ is a BNF, we can prove $\forall i \in [m].\ |\mathsf{FVars}_i\ t| < \mathsf{bd}_F$ for all *raw terms* t, hence also for all *terms* t, where bd_F is the bound of F (Section 2.2). To ensure that $|\mathsf{FVars}_i\ t| < |\alpha_i|$, it suffices to instantiate α_i with a type with a cardinality $\geq \mathsf{bd}_F$. We can therefore hope to prove the existence of a function $\mathsf{sub} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$ satisfying $(*)$ if $\mathsf{bd}_F < |\alpha_i|$ and $|\mathsf{supp}\ f_i| < |\alpha_i|$. The proof will be given in Section 7.1.

Here is an illustration of the above phenomenon, in a case that goes beyond finite support. Using the notations of Example 9, let $J = I = \mathit{nat}$ and let t be the process term $c\,(a_0).\ \sum_{i \in \mathit{nat}}\ \overline{c}\,a_i.\ 0$, where the a_i's are all distinct variables. Then what is the term $\mathsf{sub}\ f\ t$, where f is the function that sends a_0 to a_0 and any a_{i+1} to a_i? Clearly, $\mathsf{sub}\ f\ t$ should start with $c\,(a)$ for some variable a. However, a cannot be an a_i, since a_i must be free in $\mathsf{sub}\ f\ t$ (due to a_{i+1} being free in t). But what if the a_i's exhaust all the available variables? To avoid this scenario, we ask that the support of f be smaller than the set of available variables, which is true for example (1) if the support is finite and there are infinitely many variables, or (2) if the support is countable and there are uncountably many variables. In the first case, f is not deemed suitable for substitution. In the second case, there exists a fresh variable x_π with the help of which we can express $\mathsf{tsub}\ f\ t$ as $c\,(a_\pi).\ \sum_{i \in \mathit{nat}}\ \overline{c}\,a'_i.\ 0$, where $a'_0 = a_\pi$ and $a'_{i+1} = a_i$. By keeping the type $\alpha\ \boldsymbol{T}$ of terms polymorphic in the type α of variables, we can avoid committing to a specific scenario. This contributes to modularity: When using $\alpha\ \boldsymbol{T}$ as part of a larger (co)datatype (perhaps defined as a $\boldsymbol{T}$-nested fixed point), $\boldsymbol{T}$ will be able to export collections of variables of any required size. (See also Section 5.9.)

The above solution seems to require needlessly many variables when F is finitary—i.e., $\mathsf{bd}_F = \aleph_0$—which is the case with all finitely branching datatypes. With our approach, we would need α to be uncountable, even though countably infinitely many variables would suffice. It could be argued that variable countability is theoretically unimportant, and indeed some textbooks only assume an *infinite* supply of variables. But in practice the situation is different. For example, when writing a code generator for operations on finitary syntax, it helps if variables come in a countable supply. Therefore, it is worth salvaging countability if we can. It turns out that we can do this with a little

insight from the theory of cardinals—noting that $|\mathsf{FVars}_i\ t| < |\alpha_i|$ for all $i \in [m]$ and $t : \overline{\alpha}\ \boldsymbol{T}$ can be achieved using the nonstrict inequality $\mathrm{bd}_F \leq |\alpha_i|$ if $|\alpha_i|$ is a regular cardinal.

Theorem 10. There exists a (polymorphic) function $\mathsf{sub} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$ satisfying $(*)$ for all α_i and f_i if $|\alpha_i|$ is regular, $\mathrm{bd}_F \leq |\alpha_i|$, and $|\mathsf{supp}\ f_i| < |\alpha_i|$.

This solution is applicable for any MRBNF F: Since there exist arbitrarily large regular cardinals, for any bd_F we can choose suitable types α_i—for example, we can choose α_i whose cardinality is the successor cardinal of bd_F. Moreover, the solution gracefully caters for the finitary case: Since $\aleph_0$ is regular, for a countable bound bd_F we can choose countable types α_i.

5.7 Term-for-Variable Substitution

So far, we have only discussed variable-for-variable substitutions. Often it is necessary to perform a term-for-variable substitution, in a capture-avoiding fashion. In the λ-calculus, we could substitute $\lambda c.\ \mathsf{Var}\ a$ for b in $\lambda a.\ (\mathsf{Var}\ a)\ (\mathsf{Var}\ b)$, yielding $\lambda a'.\ (\mathsf{Var}\ a')\ (\lambda c.\ \mathsf{Var}\ a)$ (after a renaming which does not affect alpha-equivalence). However, not all syntaxes with bindings allow substituting terms for variables. Process terms in the π-calculus [Milner 1999] contain channel variables (names), which can be substituted by other channel variables but not by processes.

So when is term-for-variable substitution possible? A key difference is that the λ-calculus, unlike the π-calculus, can embed single variables into terms. This can be achieved either explicitly via an operator (e.g., Var) or implicitly by stating that variables are terms.

We can express such situations abstractly in our framework, by requiring that the framing MRBNF $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F$ accommodate such embeddings. We fix $I \subseteq [m]$ and assume injective natural transformations $\eta_i : \beta_i \to (\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F$ for $i \in I$ such that $\mathsf{set}^F_i\ (\eta_i\ a) = \{a\}$. Moreover, we assume that η_i is the only source of variables in F, by requiring that $\mathsf{set}^F_i\ x = \varnothing$ for every x that is not in the image of η_i. The injections of variables into terms, $\mathsf{Var}_i : \alpha_i \to \overline{\alpha}\ \boldsymbol{T}$, are defined as $\mathsf{Var}_i = \mathsf{ctor} \circ \eta_i$. For the syntax of our running Example 5, where $(\beta, \alpha, \tau)\ F = \beta + ((\alpha \times \alpha)\ @\ \alpha) \times \tau \times \tau$, we have that $\eta : \beta \to (\beta, \alpha, \tau)\ F$ is the injection of the leftmost summand.

We can now define simultaneous term-for-variable substitution similarly to variable-for-variable substitution, parameterized by functions $f_i : \alpha_i \to \overline{\alpha}\ \boldsymbol{T}$ of suitable small support:

$$\mathsf{tsub}\ \overline{f}^I\ (\mathsf{ctor}\ x) = \begin{cases} f_i\ a & \text{if } x \text{ has the form } \eta_i\ a \\ \mathsf{ctor}\ (\mathsf{map}_F\ [\mathsf{id}]^m\ [\mathsf{id}]^m\ [\mathsf{tsub}\ \overline{f}^I]^n\ x) & \text{otherwise} \end{cases} \qquad (**)$$

provided that $\forall i \in I.\ \mathsf{nonClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing$. Above, $\mathsf{supp}\ f_i$ is the union of $A_i = \{a : \alpha_i \mid f_i\ a \neq \mathsf{Var}\ a\}$ and $\mathsf{image}\ (\mathsf{FVars}_i \circ f_i)\ A_i$ (A_i's image through $\mathsf{FVars}_i \circ f_i$), and $\overline{f}^I$ denotes the tuple $(f_i)_{i \in I}$.

For a term $t = \mathsf{ctor}\ x$, saying that x has the form $\eta_i\ a$ is the same as saying that t has the form $\mathsf{Var}_i\ a$—hence the first case in the above equality is the base case of a variable term $\mathsf{Var}_i\ a$.

The existence of an operator tsub exhibiting such recursive behavior can be established by playing a similar cardinality game as we did for sub:

Theorem 11. There exists a (polymorphic) function $\mathsf{tsub} : \prod_{i \in I} (\alpha_i \to \overline{\alpha}\ \boldsymbol{T}) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$ satisfying $(**)$ for all α_i and f_i if $|\alpha_i|$ is regular, $\mathrm{bd}_F \leq |\alpha_i|$, and $|\mathsf{supp}\ f_i| < |\alpha_i|$.

5.8 Non-Well-Founded Terms

We have developed the theory of well-founded terms framed by an abstract binder type F using a binding dispatcher θ. An analogous development results in a theory for possibly non-well-founded terms, yielding non-well-founded terms modulo the alpha-equivalence induced by θ:

$$\overline{\alpha}\ \boldsymbol{T} \simeq^{\infty}_{\theta} (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\ F$$

To this end, raw terms are defined as a greatest fixpoint: $\overline{\alpha}\ T \simeq^{\infty} (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ T]^n)\ F$. Then alpha-equivalence $\equiv : \overline{\alpha}\ T \to \overline{\alpha}\ T \to \mathit{bool}$ is defined by the same rules as in Section 5.2, but employing a coinductive (greatest-fixpoint) interpretation. On the other hand, the free-variable operator is still defined inductively, by the same rules as in Section 5.1.

To see why alpha-equivalence becomes coinductive whereas free variables stay inductive, imagine coinductive terms as infinite trees: If two terms are alpha-equivalent, this cannot be determined by a finite number of the applications of $\equiv$; by contrast, if a variable is free in a term, it must be located somewhere at a finite depth, so a finite number of rule applications should suffice to find it.

This asymmetry between the inductive and the coinductive case would appear to stand in the way of a duality principle that would enable us to reuse, or at least copy, the proofs above to cover non-well-founded terms. Fortunately, there is a way to restore the symmetry. On well-founded terms, $\equiv$ could have been equivalently defined coinductively. This is because the fixpoint operator $\mathsf{Op}_{\equiv} : (\overline{\alpha}\ T \to \overline{\alpha}\ T \to \mathit{bool}) \to (\overline{\alpha}\ T \to \overline{\alpha}\ T \to \mathit{bool})$ underlying the definition of $\equiv$ has a unique fixpoint: $(\equiv) = \mathsf{lfp}\ \mathsf{Op}_{\equiv} = \mathsf{gfp}\ \mathsf{Op}_{\equiv}$. In addition, the recursive definition of map_T on well-founded terms in Section 5.2 has an identical formulation for non-well-founded terms, although it has a different, corecursive justification.

As a result, many properties concerning the constructor, alpha-equivalence, free variables, the map function, and their combination on raw terms, including Theorem 6 (which justifies the construction of $\overline{\alpha}\ \boldsymbol{T}$), can be proved in exactly the same way for possibly non-well-founded terms. All the theorems stated in Sections 5.1 to 5.7 hold for non-well-founded terms as well, with identical formulations. In particular, our solution to allow infinite support also applies to non-well-founded terms, which is crucial given that infinite terms rarely have finite support.

Example 12. In Example 3's setting, i.e., with the MRBNF and binding dispatcher for the syntax of λ-calculus, by switching from the least to the greatest fixpoint we obtain that $\alpha\ \boldsymbol{T}$ consists of all possibly non-well-founded λ-terms, known as Böhm trees [Barendregt 1984; Kennaway et al. 1997].

5.9 Modularity Considerations

Starting with a binding dispatcher θ and an $\overline{\alpha}$-MRBNF $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F$, we have constructed the binding-aware datatype (or codatatype) $\overline{\alpha}\ \boldsymbol{T}$ as $\overline{\alpha}\ \boldsymbol{T} \simeq_{\theta}^{(\infty)} (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\ F$. It enjoys the following property.

Theorem 13. $\overline{\alpha}\ \boldsymbol{T}$ is an $\overline{\alpha}$-MRBNF with map function $\mathsf{map}_{\boldsymbol{T}}$ and set functions FVars_i.

This suggests that our framework is modular in the sense that we can employ $\boldsymbol{T}$ in further constructions of binding-aware types. This is indeed possible if we want to use the variables $\overline{\alpha}$ that parameterize $\overline{\alpha}\ \boldsymbol{T}$ as binding variables. For example, if $\mathsf{len}\ \overline{\alpha} = 1$ and we take $(\beta, \alpha, \tau)\ F'$ to be $\beta + \alpha\ \boldsymbol{T} \times \tau$, then F' is an α-MRBNF, which can in turn be used to build further binding-aware datatypes $\alpha\ \boldsymbol{T}'$ as $\alpha\ \boldsymbol{T}' \simeq_{\theta} (\alpha, \alpha, \alpha\ \boldsymbol{T}')\ F'$.

However, $\boldsymbol{T}$ cannot also export its variables as free variables. For example, if again $\mathsf{len}\ \overline{\alpha} = 1$ and we take $(\beta, \alpha, \tau)\ F'$ to be $\beta + \alpha \times \beta\ \boldsymbol{T} \times \tau$, then F' is not an α-MRBNF; it is only a (β, α)-MRBNF, since it is defined using $\beta\ \boldsymbol{T}$, which imposes a map-restriction on β as well. In particular, F' cannot be employed in fixpoints $\alpha\ \boldsymbol{T}' \simeq_{\theta} (\alpha, \alpha, \alpha\ \boldsymbol{T}')\ F'$, since they require unrestricted functoriality of $(\beta, \alpha, \tau)\ F'$ on β. Unfortunately, this second scenario seems to be the most useful. The next example illustrates it. It considers a syntactic category of types that allows binding type variables, while participating as annotations in a syntactic category of terms that also allows binding type variables.

Example 14. Consider the syntax of System F types

$$\sigma ::= \mathsf{TyVar}\ a \mid \forall a.\ \sigma \mid \sigma \to \sigma$$

assumed to be quotiented by the alpha-equivalence standardly induced by the ∀-binders. In our framework, this is modeled as $\alpha\ \boldsymbol{T} \simeq_\theta (\alpha, \alpha, \alpha\ \boldsymbol{T}, \alpha\ \boldsymbol{T})\ F$, where $\theta = \{(1,1)\}$ and

$$(\beta, \alpha, \tau_1, \tau_2)\ F\ =\ \beta + (\alpha \times \tau_1) + (\tau_2 \times \tau_2)$$

Now consider the syntax of System F terms, writing a' for term variables:

$$t\ ::=\ \mathsf{Var}\ a' \mid \Lambda a.\ t \mid \lambda a' : \sigma.\ t \mid t\ \sigma \mid t\ t$$

This should be expressed as $\overline{\alpha}\ \boldsymbol{T'} \simeq_\theta (\overline{\alpha}, \overline{\alpha}, [\alpha\ \boldsymbol{T'}]^3)\ F'$, where $\theta = \{(1,1),(2,2)\}$ and

$$(\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F'\ =\ \beta_2 + (\alpha_1 \times \tau_1) + (\alpha_2 \times \beta_1\ \boldsymbol{T} \times \tau_2) + (\tau_3 \times \beta_1\ \boldsymbol{T}) + (\tau_3 \times \tau_3)$$

with $\mathsf{len}\ \overline{\beta} = \mathsf{len}\ \overline{\alpha} = 2$ and $\mathsf{len}\ \overline{\tau} = 3$. Indeed, this would give the overall fixpoint equation

$$\overline{\alpha}\ \boldsymbol{T'}\ \simeq_\theta\ \alpha_2 + (\alpha_1 \times \overline{\alpha}\ \boldsymbol{T'}) + (\alpha_2 \times \alpha_1\ \boldsymbol{T} \times \overline{\alpha}\ \boldsymbol{T'}) + (\overline{\alpha}\ \boldsymbol{T'} \times \overline{\alpha}\ \boldsymbol{T}) + \overline{\alpha}\ \boldsymbol{T'} \times \overline{\alpha}\ \boldsymbol{T'}$$

In this scheme, α_1 stores the System F type variables, and α_2 stores the System F term variables. As usual, this isomorphism is considered up to the alpha-equivalence induced by θ, which tells us that in the second summand α_1 binds in its neighboring $\overline{\alpha}\ \boldsymbol{T'}$, and in the third summand α_2 binds in its neighboring $\overline{\alpha}\ \boldsymbol{T'}$. Note that System F type variables (represented by α_1) appear as binding in the second summand and as free (as part of $\alpha_1\ \boldsymbol{T}$) in the third summand. However, the definition of $\overline{\alpha}\ \boldsymbol{T'}$ is not possible; due to the presence of $\beta_1\ \boldsymbol{T}$ as a component, $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F'$ is not an $\overline{\alpha}$-restricted MRBNF, but is additionally map-restricted on β_1.

The above problem would disappear if $\boldsymbol{T}$ were an unrestricted functor (with respect to arbitrary functions). However, the map function $\mathsf{map}_{\boldsymbol{T}}$'s restriction to endobijections is quite fundamental: Its definition is based on the low-level map_T on raw terms, which preserves alpha-equivalence only if applied to endoinjections or endobijections. (This phenomenon is also the reason for nominal logic's focus on bijective renaming.)

On the other hand, besides $\mathsf{map}_{\boldsymbol{T}} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$, on $\overline{\alpha}\ \boldsymbol{T}$, we can also rely on the capture-avoiding substitution operator $\mathsf{sub} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$. The latter has functorial behavior with respect to functions $f_i : \alpha_i \to \alpha_i$ that are not endobijections, but suffers from a different kind of limitation: It requires that f has *small support* (of cardinality less than $|\alpha|$). Thus, we do have a partial preservation of functoriality that goes beyond endobijections: On $\overline{\alpha}$, the framing F was an unrestricted functor, while the emerging datatype is a functor only with respect to small-support endofunctions.

At this point, it is worth asking whether unrestricted functoriality of $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F$ on its free-variable inputs $\overline{\beta}$ is really necessary for the constructions leading to $\overline{\alpha}\ T$ and its properties. It turns out that the answer is no. It is enough to assume functoriality with respect to small-support endofunctions to recover everything we developed, at the cost of minor changes to the definitions to assume that all the functions involved have small support; in particular, we must add this condition to the $\mathsf{cond}_i\ f_i$ hypothesis in the definition of alpha-equivalence. This leads us to our final proposal:

Proposal 5. *A binder type is a map-restricted BNF that*

- *distinguishes between free-variable, binding-variable and potential term inputs,*
- *puts a small-support endobijection map restriction on the binding-variable inputs, and*
- *puts a small-support endofunction map restriction on the free-variable inputs,*

together with a binding dispatcher between the binding-variable inputs and the term inputs.

Thus, we will require that $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F$ be a functor with respect to small-support endofunctions on $\overline{\beta}$, with respect to small-support endobijections on $\overline{\alpha}$ and arbitrary functions on $\overline{\tau}$. Unrestricted functoriality on $\overline{\tau}$ is necessary to solve the fixpoint equations that define the (co)datatypes. To

clearly indicate this refined classification of its inputs, we will call $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F$ a $\overline{\beta}$-free $\overline{\alpha}$-binding MRBNF, where we omit "$\overline{\beta}$-free" or "$\overline{\alpha}$-binding" if the corresponding vector $\overline{\beta}$ or $\overline{\alpha}$ is empty.

This final notion of MRBNF, whose full definition is given in our technical report [Blanchette et al. 2019a], achieves the desired modularity, in the sense that the free variables of terms are really a free MRBNF component:

Theorem 15. $\overline{\alpha}\ \boldsymbol{T}$ is an $\overline{\alpha}$-free MRBNF with map function sub and set functions FVars_i.

6 BINDING-AWARE (CO)INDUCTION PROOF PRINCIPLES

In this and the next section, we provide an abstraction layer, consisting of reasoning and definitional principles, that insulates the user from raw terms. The present section is dedicated to reasoning about terms, whether well-founded or not, taking their binding structure into consideration. In what follows, we will assume that α_i is such that $|\alpha_i|$ is regular and $\mathsf{bd}_F \leq |\alpha_i|$.

6.1 Induction

Let $\overline{\alpha}\ \boldsymbol{T}$ be the type of well-founded terms, as introduced in Section 5.3. Let us also fix a polymorphic type $\overline{\alpha}\ P$, of entities we will call *parameters*. For proving a property such as $\forall t : \overline{\alpha}\ T.\ \forall p : \overline{\alpha}\ P.\ \varphi\ t\ p$, we have at our disposal the standard structural induction principle inherited by the quotient $\overline{\alpha}\ \boldsymbol{T}$ from the free datatype $\overline{\alpha}\ T$ of raw terms: It suffices to prove that, for each term ctor x, the predicate $\lambda t.\ \forall p : \overline{\alpha}\ P.\ \varphi\ t\ p$ holds provided that it holds for each of the term's recursive components. However, we can be more ambitious. The following *fresh structural induction* (*FSI*) is a binding-aware improvement inspired by the nominal logic principle of Urban and Tasson [2005], which in turn is a formally rigorous incarnation of Barendregt's variable convention:

Theorem 16 (FSI). Let $\mathsf{PFVars}_i : \overline{\alpha}\ P \to \alpha_i$ *set* with $\forall p : \overline{\alpha}\ P.\ |\mathsf{PFVars}_i\ p| < |\alpha_i|$. Given a predicate $\varphi : \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ P \to$ *bool*, to prove $\forall t : \overline{\alpha}\ \boldsymbol{T}.\ \forall p : \overline{\alpha}\ P.\ \varphi\ t\ p$, it suffices to prove

$$\forall x : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\, F.\ (\forall j \in [n].\ \forall t \in \mathsf{rec}_j\ x.\ \forall p : \overline{\alpha}\ P.\ \varphi\ t\ p)$$
$$\longrightarrow (\forall p : \overline{\alpha}\ P.\ (\forall i \in [m].\ \mathsf{nonClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{PFVars}_i\ p = \varnothing) \longrightarrow \varphi\ (\mathsf{ctor}\ x)\ p)$$

Above, we highlighted the two differences with standard structural induction: We assume that parameters p come with sets of variables $\mathsf{PFVars}_i\ p$ which are smaller than α_i. This weakens what we must prove in the induction step for a given term ctor x, by allowing us to further assume that x does not clash and that its top-binding variables are fresh for the parameter's variables.

The (FSI) principle is especially useful when the parameters are themselves terms, variables, or functions on them, which is often the case. An example is the distributivity over composition of sub:

Proposition 17. We have $\mathsf{sub}\ (g_1 \circ f_1)\ \ldots\ (g_m \circ f_m) = \mathsf{sub}\ \overline{g} \circ \mathsf{sub}\ \overline{f}$ for all $f_i, g_i : \alpha_i \to \alpha_i$ such that $|\mathsf{supp}\ f_i| < |\alpha|$ and $|\mathsf{supp}\ g_i| < |\alpha|$ for all $i \in [m]$.

This property would be difficult to prove by standard induction, since the support of the functions $\overline{f}$ and $\overline{g}$ may capture bound variables. With (FSI), by taking $(\overline{f}, \overline{g})$ as parameters, we can rule out capture and apply sub's recursive law $(*)$ directly.

6.2 Coinduction

Next, let $\overline{\alpha}\ \boldsymbol{T}$ be the type of non-well-founded terms, as introduced in Section 5.8. Concerning binding-aware proof principles for $\overline{\alpha}\ \boldsymbol{T}$, we encounter a discrepancy from the inductive case. The standard structural coinduction principle imported from raw terms would allow us to prove that a binary relation on $\overline{\alpha}\ \boldsymbol{T}$ is included in the equality if it is an F-bisimulation (i.e., if it is preserved by F's relator). Using the ideas discussed in the previous subsection, we can prove a parameter-based fresh variation of this principle, where we again emphasize the binding-specific enhancements.

Theorem 18 (FSC). Let $\overline{\alpha}\ P$ and PFVars_i be as in Theorem 16. Given a binary relation $\varphi : \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ P \to \mathit{bool}$, to prove $\forall t_1, t_2 : \overline{\alpha}\ \boldsymbol{T}.\ \forall p : \overline{\alpha}\ P.\ \varphi\ t_1\ t_2\ p \longrightarrow t_1 = t_2$, it suffices to prove

$$\begin{aligned}&\forall x_1, x_2 : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\ F.\ (\forall p : \overline{\alpha}\ P.\ \varphi\ (\mathsf{ctor}\ x_1)\ (\mathsf{ctor}\ x_2)\ p\\ &\quad \wedge (\forall i \in [m].\ \mathsf{nonClash}_i\ x_1 \wedge \mathsf{nonClash}_i\ x_2 \wedge (\mathsf{topBind}_i\ x_1 \cup \mathsf{topBind}_i\ x_2) \cap \mathsf{PFVars}_i\ p = \varnothing))\\ &\quad \longrightarrow \mathsf{rel}_F\ [(=)]^m\ [(=)]^m\ [\lambda t_1\ t_2.\ \forall p : \overline{\alpha}\ P.\ \varphi\ t_1\ t_2\ p]^n\ x_1\ x_2\end{aligned}$$

However, this proof principle is not as useful as its inductive counterpart. Consider the task of proving Proposition 17 for non-well-founded terms. Let us attempt to prove it using (FSC). We again take the parameters to be tuples $(\overline{f}, \overline{g})$ of endofunctions of small support and $\mathsf{PFVars}_i\ (\overline{f}, \overline{g})$ to be $\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i$. We define $\varphi\ t_1\ t_2\ (\overline{f}, \overline{g})$ as $\exists t.\ t_1 = \mathsf{sub}\ (g_1 \circ f_1) \ldots (g_m \circ f_m)\, t \wedge t_2 = (\mathsf{sub}\ \overline{g} \circ \mathsf{sub}\ \overline{f})\, t$. Then, it suffices to verify (FSC)'s hypothesis. We may assume that, for all endofunctions of small support f_i and g_i, (1) $\mathsf{ctor}\ x_1 = \mathsf{sub}\ (g_1 \circ f_1) \ldots (g_m \circ f_m)\ t$ and $\mathsf{ctor}\ x_2 = (\mathsf{sub}\ \overline{g} \circ \mathsf{sub}\overline{f})\ t$, and (2) $(\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i) \cap (\mathsf{topBind}_i\ x_1 \cup \mathsf{topBind}_i\ x_2) = \varnothing$ for all $i \in [m]$. We must prove (3) $\mathsf{rel}_F\ [(=)]^m\ [(=)]^m\ [\lambda t_1\ t_2.\ \forall(\overline{f}, \overline{g}).\ \varphi\ t_1\ t_2\ (\overline{f}, \overline{g})]^m\ x_1\ x_2$. To this end, assume t has the form $\mathsf{ctor}\ x$, where thanks to the availability of enough fresh variables we may assume $(\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i) \cap \mathsf{topBind}_i\ x = \varnothing$ for all $i \in [m]$. This allows us to push sub under the constructor in the equalities (1), obtaining (4) $\mathsf{ctor}\ x_1 = \mathsf{ctor}\ x_1'$ and $\mathsf{ctor}\ x_2 = \mathsf{ctor}\ x_2'$, where

$$\begin{aligned}x_1' &= \mathsf{map}_F\ (g_1 \circ f_1) \ldots (g_m \circ f_m)\ [\mathsf{id}]^m\ [\mathsf{sub}\,(g_1 \circ f_1) \ldots (g_m \circ f_m)]^n\ x\\ x_2' &= \mathsf{map}_F\ (g_1 \circ f_1) \ldots (g_m \circ f_m)\ [\mathsf{id}]^m\ [\mathsf{sub}\,\overline{g} \circ \mathsf{sub}\,\overline{f}]^n\ x\end{aligned}$$

At this point, we are stuck. To prove (3), we seem to need (5) $x_1 = x_1'$ and $x_2 = x_2'$, which do not follow from the equalities (4), and the freshness assumption (2) does not help. Indeed, we could use (2) in conjunction with a suitable choice of x to prove one of the equalities (5), but not both.

The problem above is a certain synchronization requirement between the top-binding variables of x_1 and x_2, which is not accounted for by the freshness hypothesis. To accommodate such a synchronization, we prove a different enhancement of structural coinduction, (ESC). Instead of explicitly avoiding clashes with parameters, (ESC) enables the terms themselves to avoid any clashes, and also to synchronize their decompositions via ctor, as long as this does not change their identity:

Theorem 19 (ESC). Given a binary relation $\varphi : \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T} \to \mathit{bool}$, to prove $\forall t_1, t_2 : \overline{\alpha}\ \boldsymbol{T}.\ \varphi\ t_1\ t_2 \longrightarrow t_1 = t_2$, it suffices to prove

$$\begin{aligned}&\forall x_1, x_2 : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\ F.\ \varphi\ (\mathsf{ctor}\ x_1)\ (\mathsf{ctor}\ x_2)\\ &\quad \longrightarrow (\exists x_1', x_2'.\ \mathsf{ctor}\ x_1 = \mathsf{ctor}\ x_1' \wedge \mathsf{ctor}\ x_2 = \mathsf{ctor}\ x_2' \wedge \mathsf{rel}_F\ [(=)]^m\ [(=)]^m\ [\lambda t_1\ t_2.\ \varphi\ t_1\ t_2]^n\ x_1'\ x_2')\end{aligned}$$

(ESC) is more general than (FSC) and can easily be used to prove it. It solves the problem in our concrete example with substitution (by allowing us to dynamically switch from x_1 and x_2 to x_1' and x_2') and similar problems when proving equational theorems on non-well-founded terms.

7 BINDING-AWARE (CO)RECURSIVE DEFINITION PRINCIPLES

Two important aspects have not been formally addressed so far. The first concerns Theorems 10 and 11, stating the existence of substitution operators. While we have shown how the need for sufficiently many fresh variables can be fulfilled, we have not accounted for the possibility of defining the substitutions as well-behaved functions on (quotiented) terms. The second aspect concerns a standard litmus test for abstract data types: the unique characterization of a construction up to isomorphism. In contrast to raw terms, which are known to form initial objects in categories of algebras for BNFs [Traytel et al. 2012], the status of terms is currently less abstract, since they rely on alpha-equivalence. Can we also characterize the term algebras as initial objects?

The answer is yes if we can provide suitable (co)recursion definitional principles, (*co*)*recursors*, for the types of terms. The main difficulty in developing such a recursor is that since terms do

not form a free datatype, we cannot define functions on terms by simply listing some constructor-based recursive clauses. Instead, the recursor must be aware of the nonfreeness introduced by bindings, and in fact must take advantage of this nonfreeness to incorporate Barendregt's variable convention. And a similar (dual) problem holds for the corecursor. The key to address these problems is to identify suitable abstract algebraic structures that satisfy term-like properties, to be used as (co)domains for (co)recursive definitions.

We will present a simple version of the (co)recursors, which are more commonly called "(co)iterators." Our technical report [Blanchette et al. 2019a] covers the straightforward extension to full-fledged (co)recursors. Below, we implicitly assume that $|\alpha_i|$ is regular and $\mathrm{bd}_F \leq |\alpha_i|$. In addition, unless otherwise stated, f_i and g_i range over (endo)bijections of type $\alpha_i \to \alpha_i$ that have small support: $|\mathsf{supp}\ f_i| < |\alpha_i|$ and $|\mathsf{supp}\ g_i| < |\alpha_i|$.

Definition 20. A *term-like structure* is a triple $\mathcal{D} = (\overline{\alpha}\ D, \overline{\mathsf{DFVars}}, \mathsf{Dmap})$, where

- $\overline{\alpha}\ D$ is a polymorphic type;
- $\overline{\mathsf{DFVars}}$ is a tuple of functions $\mathsf{DFVars}_i : \overline{\alpha}\ D \to \alpha_i\ \mathit{set}$ for $i \in [m]$;
- $\mathsf{Dmap} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ D \to \overline{\alpha}\ D$

are such that the following properties hold:

- $\mathsf{Dmap}\ [\mathsf{id}]^m = \mathsf{id}$;
- $\mathsf{Dmap}\ (g_1 \circ f_1)\ \ldots\ (g_m \circ f_m) = \mathsf{Dmap}\ \overline{g} \circ \mathsf{Dmap}\ \overline{f}$;
- $(\forall i \in [m].\ \forall a \in \mathsf{DFVars}_i\ d.\ f_i\ a = a) \longrightarrow \mathsf{Dmap}\ \overline{f}\ d = d$;
- $a \in \mathsf{DFVars}_i\ (\mathsf{Dmap}\ \overline{f}\ d) \longleftrightarrow f_i^{-1}\ a \in \mathsf{DFVars}_i\ d$.

Term-like structures imitate to a degree the type of terms. Indeed, $(\overline{\alpha}\ \boldsymbol{T}, \overline{\mathsf{FVars}}, \mathsf{map}_{\boldsymbol{T}})$ forms the archetypal term-like structure.

7.1 Binding-Aware Recursor

Let $\overline{\alpha}\ \boldsymbol{T}$ be the type of well-founded terms (Section 5.3). Recall from Section 6.1 that fresh induction relies on parameters, assumed to be equipped with small-cardinality free-variable-like operators. To discuss recursion, we need parameters to have map functions as well.

Definition 21. A *parameter structure* is a term-like structure $\mathcal{P} = (\overline{\alpha}\ P, \overline{\mathsf{FVars}}, \mathsf{Pmap})$ such that $\forall p : \overline{\alpha}\ P.\ |\mathsf{PFVars}_i\ p| < |\alpha_i|$.

The codomains of our recursive definitions, called *models*, must be even more similar to the type of terms than term-like structures. Namely, they must also have a constructor-like operator.

Definition 22. Given a parameter structure $\mathcal{P}$, a *$\mathcal{P}$-model* is a quadruple $\mathcal{U} = (\overline{\alpha}\ U, \overline{\mathsf{UFVars}}, \mathsf{Umap}, \mathsf{Uctor})$, where

- $(\overline{\alpha}\ U, \overline{\mathsf{UFVars}}, \mathsf{Umap})$ is a term-like structure;
- $\mathsf{Uctor} : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ P \to \overline{\alpha}\ U]^n)\ F \to \overline{\alpha}\ P \to \overline{\alpha}\ U$

such that the following properties hold:

(MC) $\mathsf{Umap}\ \overline{f}\ (\mathsf{Uctor}\ y\ p) = \mathsf{Uctor}\ (\mathsf{map}_F\ \overline{f}\ \overline{f}\ [\mathsf{Umap}\ \overline{f}]^n\ y)\ (\mathsf{Pmap}\ \overline{f}\ p)$

(VC) $(\forall i \in [m].\ \mathsf{topBind}_i\ y \cap \mathsf{PFVars}_i\ p = \varnothing) \wedge$
$(\forall i \in [m].\ \forall j \in [n].\ \forall pu \in \mathsf{rec}_j\ y.\ \forall p.\ \mathsf{UFVars}_i\ (pu\ p) \setminus \mathsf{topBind}_{i,j}\ y \subseteq \mathsf{PFVars}_i\ p)$
$\longrightarrow\ \forall i \in [m].\ \mathsf{UFVars}_i\ (\mathsf{Uctor}\ y\ p) \subseteq \mathsf{topFree}\ y \cup \mathsf{PFVars}_i\ p$

Above, we use similar concepts for models as for terms, such as $\mathsf{topBind}_i$ and rec_j, applied to members y of $(\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ P \to \overline{\alpha}\ U]^n)\ F$. They are defined in the same way and follow the same intuition as for terms, with Uctor playing the role of ctor. The recursion theorem states the existence and uniqueness of a "recursively defined" function from terms to any model:

Theorem 23. Given a parameter structure $\mathcal{P}$ and a $\mathcal{P}$-model $\mathcal{U}$, there exists a unique function $H : \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ P \to \overline{\alpha}\ U$ that preserves the constructor, mapping, and free-variable operators:

(C) $(\forall i \in [m].\ \mathsf{nonClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{PFVars}_i\ p = \varnothing) \longrightarrow$
$H\ (\mathsf{ctor}\ x)\ p = \mathsf{Uctor}\ (\mathsf{map}_F\ [\mathsf{id}]^{2m}\ [H]^n\ x)\ p$
(M) $H\ (\mathsf{map}_{\boldsymbol{T}}\ \overline{f}\ t)\ p = \mathsf{Umap}\ \overline{f}\ (H\ t\ (\mathsf{Pmap}\ \overline{f}^{-1}\ p))$
(V) $\forall i \in [m].\ \mathsf{UFVars}_i\ (H\ t\ p) \subseteq \mathsf{FVars}_i\ t \cup \mathsf{PFVars}_i\ p$

This theorem captures the following contract: To define a function H from $\overline{\alpha}\ \boldsymbol{T}$ to $\overline{\alpha}\ U$, it suffices to organize $\overline{\alpha}\ U$ as a $\mathcal{P}$-model for some parameter structure $\mathcal{P}$. In other words, it suffices to define on $\overline{\alpha}\ U$ some $\mathcal{P}$-model operators and check that they satisfy the required properties. In exchange, we obtain the function H, which is also guaranteed to preserve the operators.

This function depends on both terms and parameters. Intuitively, H recurses over terms while the binding variables are assumed to avoid the parameters' variables. Indeed, the theorem's clause (C) specifies the behavior of H on terms of the form ctor x not for an arbitrary x, but for a (nonclashing) x whose top-binding variables do not overlap with those of a given parameter p. This is the recursive-definition incarnation of Barendregt's convention, in the same way as the parameter trick of fresh induction (Theorem 16) is its inductive-proof incarnation.

The two additional model axioms are also generalizations of term properties. They describe the interaction between the constructor-like operator and the other operators. (MC) states that the map function commutes with the constructor (for endobijections $\overline{f}$ of small support). (VC) is more subtle. If we ignore its first premise, (VC) states an implication that generalizes and weakens the following property of the term constructor's free variables:

$$\forall i \in [m].\ \mathsf{FVars}_i\ (\mathsf{ctor}\ x)\ =\ \mathsf{topFree}\ y \cup \bigcup_{j \in [n]} \bigcup_{t \in \mathsf{rec}_j\ x} \mathsf{FVars}_i\ t \setminus \mathsf{topBind}_{i,j}\ x$$

The weakening consists of turning the above equality, which has the form $\forall i \in [m].\ L_i = R_i \cup R'_i$, into an inclusion $\forall i \in [m].\ L_i \subseteq R_i \cup R'_i$ and further weakening the latter into an "inclusion modulo parameters," $\forall i \in [m].\ R'_i \subseteq \mathsf{PFVars}_i\ p \longrightarrow L_i \subseteq R_i \cup \mathsf{PFVars}_i\ p$, which is equivalent to

$$\begin{aligned}&(\forall i \in [m].\ \forall j \in [n].\ \forall t \in \mathsf{rec}_j\ x.\ \mathsf{FVars}_i\ t \setminus \mathsf{topBind}_{i,j}\ x \subseteq \mathsf{PFVars}_i\ p)\\ &\longrightarrow\ \forall i \in [m].\ \mathsf{FVars}_i\ (\mathsf{ctor}\ x) \subseteq \mathsf{topFree}\ y \cup \mathsf{PFVars}_i\ p\end{aligned}$$

(VC) is the model version of this last property, mutatis mutandis (e.g., replacing ctor and FVars_i with Uctor and UFVars_i), together with the additional weakening brought by its first premise: The top-binding variables in Uctor $y\ p$ are fresh for the parameters. Given that weaker model axiomatizations lead to more expressive recursors, our recursor improves on the state of the art (Section 9).

To define the variable-for-variable substitution sub, we define $\mathcal{P}$ by taking $\overline{\alpha}\ P$ to consist of all tuples of small-support endofunctions $\overline{f}$, $\mathsf{PFVars}_i\ \overline{f} = \mathsf{supp}\ f_i$ and $\mathsf{Pmap}\ \overline{g}\ \overline{f} = \overline{g} \circ \overline{f} \circ \overline{g}^{-1}$. We define the $\mathcal{P}$-model $\mathcal{U}$ by taking $\overline{\alpha}\ U = \overline{\alpha}\ \boldsymbol{T}$, $\mathsf{UFVars}_i = \mathsf{FVars}_i$, $\mathsf{Umap} = \mathsf{map}_{\boldsymbol{T}}$ and $\mathsf{Uctor}\ y\ \overline{f} = \mathsf{ctor}\ (\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\lambda pu.\ pu\ \overline{f}]^n\ y)$. To apply Theorem 23, we must check its hypotheses, which amount to standard identities on terms. We obtain a function $\mathsf{sub} : \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ P \to \overline{\alpha}\ \boldsymbol{T}$ satisfying three clauses, among which

(C) $(\forall i \in [m].\ \mathsf{nonClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing)$
$\longrightarrow\ \mathsf{sub}\ (\mathsf{ctor}\ x)\ \overline{f} = \mathsf{ctor}\ (\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\lambda pu.\ pu\ \overline{f}]^n\ (\mathsf{map}_F\ [\mathsf{id}]^{2m}\ [\mathsf{sub}]^n\ x))$

By (restricted) functoriality, $\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\lambda pu.\ pu\ \overline{f}]^n\ (\mathsf{map}_F\ [\mathsf{id}]^{2m}\ [\mathsf{sub}]^n\ x) = \mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\lambda t.\ \mathsf{sub}\ t\ \overline{f}]^n\ x$, making (C) equivalent to Section 5.5's clause $(*)$, hence proving the desired behavior for substitution (Theorem 10)—after flipping the arguments of sub, to turn it into a function of type $\overline{\alpha}\ P \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$. Term-for-variable substitution (Section 5.7) can be defined similarly.

To characterize terms as an abstract data type, let $\bot$ be the parameter structure where the carrier type is a singleton, the map function is trivial, and PFVars_i returns $\varnothing$ for all $i \in [m]$. We obtain the following, as an immediate consequence of Theorem 23:

Corollary 24. $(\overline{\alpha}\ \boldsymbol{T}, \overline{\mathsf{FVars}}, \mathsf{map}_{\boldsymbol{T}}, \mathsf{ctor})$ is the initial $\bot$-model (where a model morphism is a function that preserves all the operators).

To summarize, the generalization of natural term properties has led us to the axiomatization of models and to an associated recursor. The axiomatization factors in parameters, which are useful for enforcing Barendregt's convention; in particular, they allow a uniform recursive definition of substitution. If we ignore parameters, our recursor exhibits the term model as initial, which yields an up to isomorphism characterization in a standard way (via Lambek's lemma).

7.2 Binding-Aware Corecursor

Next, we take $\overline{\alpha}\ \boldsymbol{T}$ to be the type of non-well-founded terms. Traditionally, a corecursor is based on an identification of our collection of interest as a *final coalgebra* for a suitable functor. The problem here is that, unlike raw terms, terms do not form a standard coalgebra for F. Indeed, since ctor is not injective, there is no destructor operation $\mathsf{dtor} : \overline{\alpha}\ \boldsymbol{T} \to (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\ F$.

Yet something akin to a coalgebaic structure can still be obtained if we leave some room for nondeterminism. Namely, we define a nondeterministic destructor $\mathsf{dtor} : \overline{\alpha}\ \boldsymbol{T} \to ((\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\ F)\ \mathit{set}$ as $\mathsf{dtor}\ t = \{x \mid t = \mathsf{ctor}\ x\}$. Crucially, this destructor is still *deterministic up to a renaming* of the top-binding variables. Indeed, Proposition 7 ensures that, for any $x, x' \in \mathsf{dtor}\ t$, we have $x' = \mathsf{map}_F\ [\mathsf{id}]^m\ \overline{f}\ [\mathsf{map}_{\boldsymbol{T}}\ \overline{f}]^n\ x$ for some small-support endobijections $\overline{f}$ subject to some suitable conditions. This suggests the following axiomatization of corecursive models:

Definition 25. A *comodel* is a quadruple $\mathcal{U} = (\overline{\alpha}\ U, \overline{\mathsf{UFVars}}, \mathsf{Umap}, \mathsf{Udtor})$, where

- $(\overline{\alpha}\ U, \overline{\mathsf{UFVars}}, \mathsf{Umap})$ is a term-like structure;
- $\mathsf{Udtor} : \overline{\alpha}\ U \to ((\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ U]^n)\ F)\ \mathit{set}$

such that the following properties hold:

(Dne) $\mathsf{Udtor}\ u \neq \varnothing$

(DRen) $y, y' \in \mathsf{Udtor}\ u \longrightarrow \exists \overline{f}.\ (\forall i \in [m].\ \mathsf{bij}\ f_i \wedge |\mathsf{supp}\ f_i| < |\alpha_i| \wedge$
$(\forall a \in (\bigcup_{j \in [n]} (\bigcup_{u \in \mathsf{rec}_j\ y} \mathsf{UFVars}_i\ u) \setminus \mathsf{topBind}_{i,j}\ y).\ f_i\ a = a)) \wedge$
$y' = \mathsf{map}_F\ [\mathsf{id}]^m\ \overline{f}\ [\mathsf{Umap}\ \overline{f}]^n\ y$

(MD) $\mathsf{Udtor}\ (\mathsf{Umap}\ \overline{f}\ u) \subseteq \mathsf{image}\ (\mathsf{map}_F\ \overline{f}\ \overline{f}\ [\mathsf{Umap}\ \overline{f}]^n)\ (\mathsf{Udtor}\ u)$

(VD) $y \in \mathsf{Udtor}\ u \longrightarrow$
$\forall i \in [m].\ \mathsf{topFree}\ y \cup \bigcup_{j \in [n]} (\bigcup_{u' \in \mathsf{rec}_j\ y} \mathsf{UFVars}_i\ u') \setminus \mathsf{topBind}_{i,j}\ y \subseteq \mathsf{UFVars}_i\ u$

Thus, comodels exhibit the term-like structure of models; but instead of a constructor-like operator, they are equipped with a destructor-like operator Udtor that returns nonempty sets (Dne) and is deterministic modulo a renaming (DRen), thereby generalizing properties of the term destructor. Moreover, (MD) generalizes the term property $\mathsf{dtor}\ (\mathsf{map}_{\boldsymbol{T}}\ \overline{f}\ t) \subseteq \mathsf{image}\ (\mathsf{map}_F\ \overline{f}\ \overline{f}\ [\mathsf{map}_{\boldsymbol{T}}\ \overline{f}]^n)\ (\mathsf{dtor}\ t)$, which after expanding the definition of dtor from ctor becomes $\mathsf{map}_{\boldsymbol{T}}\ \overline{f}\ t = \mathsf{ctor}\ x' \longrightarrow \exists x.\ t = \mathsf{ctor}\ x \wedge x' = \mathsf{map}_F\ \overline{f}\ \overline{f}\ [\mathsf{map}_{\boldsymbol{T}}\ \overline{f}]^n\ x$. Since this property is (implicitly) quantified universally over the small-support endobijections $\overline{f}$, by mapping with the inverses $\overline{g} = \overline{f}^{-1}$ of these functions and using the restricted functoriality of $\mathsf{map}_{\boldsymbol{T}}$ and map_F we can rewrite it into

$\mathsf{map}_{\boldsymbol{T}}\ \overline{g}\ (\mathsf{map}_{\boldsymbol{T}}\ \overline{f}\ t) = \mathsf{map}_{\boldsymbol{T}}\ \overline{g}\ (\mathsf{ctor}\ x') \longrightarrow$
$\exists x.\ t = \mathsf{ctor}\ x \wedge \mathsf{map}_F\ \overline{g}\ \overline{g}\ [\mathsf{map}_{\boldsymbol{T}}\ \overline{g}]^n\ x' = \mathsf{map}_F\ \overline{g}\ \overline{g}\ [\mathsf{map}_{\boldsymbol{T}}\ \overline{g}]^n\ (\mathsf{map}_F\ \overline{f}\ \overline{f}\ [\mathsf{map}_{\boldsymbol{T}}\ \overline{f}]^n\ x)$

then into $t = \mathsf{map}_{\boldsymbol{T}}\ \overline{g}\ (\mathsf{ctor}\ x') \longrightarrow \exists x.\ t = \mathsf{ctor}\ x \wedge \mathsf{map}_F\ \overline{g}\ \overline{g}\ [\mathsf{map}_{\boldsymbol{T}}\ \overline{g}]^n\ x' = x$ and finally into $\mathsf{map}_{\boldsymbol{T}}\ \overline{g}\ (\mathsf{ctor}\ x') = \mathsf{ctor}\ (\mathsf{map}_F\ \overline{g}\ \overline{g}\ [\mathsf{map}_{\boldsymbol{T}}\ \overline{g}]^n\ x')$. This last property is the one that inspired the model axiom (MC), which shows the conceptual duality between (MC) and (MD): They generalize the same term property, but one from a constructor and the other from a destructor point of view. The property (VD) is also in a dual relationship with the corresponding model axiom (VC). Both can be traced back to the term property $\forall i \in [m].\ \mathsf{FVars}_i\ (\mathsf{ctor}\ x) = \mathsf{topFree}\ x \cup \bigcup_{j\in[n]} (\bigcup_{t\in \mathsf{rec}_j\ x} \mathsf{FVars}_i\ t) \setminus \mathsf{topBind}_{i,j}\ x$, which is weakened by (VC) and (VD) into inclusions of opposite polarities. And indeed, comodels achieve the dual of what models achieve:

Theorem 26. Given a comodel $\boldsymbol{\mathcal{U}}$, there exists a unique function $H : \overline{\alpha}\ U \to \overline{\alpha}\ \boldsymbol{T}$ that preserves the destructor, mapping and free-variable operators, in the following sense:

(D) $\mathsf{map}_F\ [\mathsf{id}]^{2m}\ [H]^n\ (\mathsf{Udtor}\ d) \subseteq \mathsf{dtor}\ (H\ d)$

(M) $H\ (\mathsf{Umap}\ \overline{f}\ u) = \mathsf{map}_{\boldsymbol{T}}\ \overline{f}\ (H\ u)$

(V) $\forall i \in [m].\ \mathsf{UFVars}_i\ (H\ t) \subseteq \mathsf{FVars}_i\ t$

Note that clause (D) can be rewritten into $y \in \mathsf{Udtor}\ d \longrightarrow \mathsf{map}_F\ [\mathsf{id}]^{2m}\ [H]^n\ y \in \mathsf{dtor}\ (H\ d)$ and, expanding the definition of dtor from ctor, further into

(D′) $y \in \mathsf{Udtor}\ u \longrightarrow H\ u = \mathsf{ctor}\ (\mathsf{map}_F\ [\mathsf{id}]^{2m}\ [H]^n\ y)$

which shows the corecursive behavior of H in a more operational fashion: To build a (possibly infinite) term starting with the input d, H can choose any $y \in \mathsf{Udtor}\ d$ and then delve into y after "producing" a ctor. Thanks to the comodel axioms, notably, (DRen), the choice of y is irrelevant.

Corollary 27. $(\overline{\alpha}\ \boldsymbol{T}, \overline{\mathsf{FVars}}, \mathsf{map}_{\boldsymbol{T}}, \mathsf{dtor})$ is the final comodel.

Unlike models, our comodels do not have parameters. This is because, in the corecursive case, any freshness assumptions can be easily incorporated in the choice of the destructor-like operator. (This mirrors the situation of binding-aware coinduction, which also departs from binding-aware induction precisely on the topic of explicit parameters.) The corecursive definition of substitution is a good illustration of this phenomenon. We define the comodel $\boldsymbol{\mathcal{U}}$ by taking $\overline{\alpha}\ U$ to consists of all pairs $(t, \overline{f})$ with t term and $\overline{f}$ tuple of small-support endofunctions, $\mathsf{UVar}_i\ (t, \overline{f}) = \mathsf{FVars}_i\ t \cup \mathsf{supp}\ f_i$, $\mathsf{Umap}\ \overline{g}\ (t, \overline{f}) = (\mathsf{map}_{\boldsymbol{T}}\ \overline{g}\ t, \overline{g} \circ \overline{f} \circ \overline{g}^{-1})$, and $\mathsf{Udtor}\ (t, \overline{f}) = \{\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [(\lambda t'.\ (t', \overline{f}))]^n\ x \mid x \in \mathsf{dtor}\ t \wedge \mathsf{nonClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing\}$. We have highlighted how we insulate, among all possible ways to choose $x \in \mathsf{dtor}\ t$, those x's that avoid capture, as required by the desired clause $(*)$ for substitution—which is the same as for well-founded terms. This is a general trick for replacing the explicit use of parameters. After checking that this is indeed a comodel, Theorem 26 offers the function $\mathsf{sub} : \overline{\alpha}\ U \to \overline{\alpha}\ \boldsymbol{T}$ that satisfies three clauses, among which

(D′) $x \in \mathsf{dtor}\ t \wedge (\forall i \in [m].\ \mathsf{nonClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing)$
$\longrightarrow \mathsf{sub}\ (\overline{f}, t) = \mathsf{ctor}\ (\mathsf{map}_F\ [\mathsf{id}]^{2m}\ [\mathsf{sub}]^n\ (\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [(\lambda t'.\ (t', \overline{f}))]^n\ x))$

This is equivalent to $(*)$ by the functoriality of map_F and the equivalence $x \in \mathsf{dtor} \longleftrightarrow t = \mathsf{ctor}\ x$.

8 ISABELLE FORMALIZATION AND IMPLEMENTATION

All our results have been formalized in Isabelle/HOL, in a slightly less general case than presented in this paper [Blanchette et al. 2019b]. While the formalization is expressed abstractly in terms of arbitrary type constructors and constants (such as F and map_F), it is concrete in that it fixes arities for the type constructors. Thus, for the fixpoint constructions, instead of $\theta \subseteq [m] \times [n]$ and $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F$ where $\mathsf{len}\ \overline{\beta} = \mathsf{len}\ \overline{\alpha} = m$ and $\mathsf{len}\ \overline{\tau} = n$, we work with a fixed $(\beta_1, \alpha_1, \tau_1, \tau_2)\ F$, taking $m = n = 1$ and $\theta = \{(1, 2)\}$.

There is no way to avoid this while working in the Isabelle/HOL user space, given that in HOL we cannot consider type constructors depending on varying numbers of type variables. On the

positive side, having the fixed-arity case fully worked out gives us strong confidence that our results are correct. Moreover, by doing a lot of copy-pasting, we can easily adapt our formalization to any given m, n, and θ. On the negative side, our framework is not yet usable in the way a definitional package such as Isabelle Nominal [Urban and Tasson 2005] and the BNF-based (co)datatype package [Blanchette et al. 2014] is. Besides the above issue with arities, another missing component is a mechanism for hiding the category theory under some user-friendly notation—for example, splitting the single abstract constructor ctor into multiple user-named constructors. As a first step, we have implemented in Standard ML an axiomatic tool that automates the process of instantiating an Isabelle formalization parameterized by some type constructors and polymorphic constants with user-specified instances. This tool is not necessary for developing our theory, but it is helpful for avoiding proof duplication—for example, when instantiating the arbitrary (co)models used by our (co)recursion principles with the specific ones needed to define substitution.

The main benefit of our approach is the semantic treatment of binders, which allows us to combine arbitrarily complex binders with (co)datatypes. But the functorial approach is not a cure for the combinatorial complexity associated with many-sortedness: multiple types of variables bound in multiple types of terms. To cope with these, we need to consider multiple arguments for our functors, and mutually recursive (co)datatypes. This complicates the implementation.

9 RELATED WORK

We will articulate our discussion of related work around three main paradigms in reasoning about bindings: the "nameful" paradigm, the nameless paradigm, and higher-order abstract syntax. The main distinction between these lies in the way they consider binders and associated concepts such as substitution. We will continue using the same notational style as above, writing α for a type of variables and $\alpha\ \boldsymbol{T}$ for the type of terms over these variables. Concretely, $\alpha\ \boldsymbol{T}$ will consist of λ-calculus terms equipped with a binding operator λ.

In what we call the *nameful paradigm*, binding variables are passed explicitly to the binding operator (so that λ has type $\alpha \to \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$), and terms are usually equated modulo alpha-equivalence. The nameful paradigm is followed by most of the informal, pen-and-paper developments of logic, λ-calculi, and programming languages. In his famous monograph on the λ-calculus, Barendregt [1984] has developed a systematic (yet informal) methodology for nameful reasoning, which is summarized by his variable convention. A rigorous account of this paradigm is offered by nominal logic [Gabbay and Pitts 2002; Pitts 2003, 2006]. In contrast, with the *nameless paradigm* originating with the work of de Bruijn [1972], the bindings are indicated through nameless pointers to binding positions in a term. The λ constructor has type $\alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$ or a variation of it, and there is a convention on what element of α is being bound, such as de Bruijn indices and de Bruijn levels [Lescanne and Rouyer-Degli 1995]. A type-safe variation of the above is achieved by enhancing the source type of λ to $(\alpha + \mathit{unit})\ \boldsymbol{T}$, with the understanding that the binding positions are the occurrences of $* : \mathit{unit}$. The third paradigm, *higher-order abstract syntax* (HOAS), is based on the reduction of all binders to the binder of a fixed λ-calculus provided by the metalogic. It relies on higher-order types for the binding operators; for example, the type of λ would be $(\alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}) \to \alpha\ \boldsymbol{T}$ or, for *weak HOAS*, $(\alpha \to \alpha\ \boldsymbol{T}) \to \alpha\ \boldsymbol{T}$. As a technique for representing and implementing logics and programming languages, HOAS originated in the early 1980s with Martin-Löf's work [Nordström et al. 1990, Chapter 3], although its ideas can be traced back to Huet and Lang [1978] and even Church [1940]. The paradigm gained traction in the formal methods community with the works of Harper et al. [1987], Pfenning and Elliott [1988], and Paulson [1989]. HOAS has been subsequently extended with sophisticated definition and (meta)reasoning capabilities (e.g., Pfenning and Schürmann [1999], Despeyroux et al. [1995], Chlipala [2008], Felty and Pientka [2010]), including (structural) recursion in a functional setting [Ferreira and Pientka 2017; Pientka 2010; Schürmann et al. 2001].

Some approaches in the literature combine two paradigms [Aydemir et al. 2008; Charguéraud 2012; Felty and Momigliano 2012; Popescu et al. 2010]. Moreover, it should be emphasized that the three paradigms do not differ fundamentally in the employed datatype of terms. Regardless of whether terms are defined by quotienting raw terms to alpha-equivalence, defined as a free datatype using nameless pointers, or encoded using a metalevel binder, they yield essentially the same concept. (In HOAS approaches, this fact is often reinforced by an *adequacy* proof.) The distinction is also not so much about what constructors are available to build terms; for example, the nameful constructors are easily definable from the nameless constructors and vice versa. The deeper difference lies in their respective *reasoning and definitional styles*, which are optimized for the default types of the binding constructors. Consider a recursively defined function f. If its λ clause has the form $f\ (\lambda a.\ t) = \ldots$, the burden is to maintain compatibility with alpha-equivalence. On the other hand, if it has the form $f\ (\lambda\, t) = \ldots$, the difficulty is to shift the pointers properly [Berghofer and Urban 2007]. From a proof assistant perspective, in either case the challenge is to provide convenient abstractions to support the chosen style.

Essentially a functorial generalization of Barendregt's variable convention, our work belongs squarely to the nameful paradigm. It is inspired by, and extends, nominal logic (Section 9.1). Its generality and modularity is owed to category theory, which has a rich tradition of functorial representation of binding structures (Section 9.2). A distinguishing feature of our work is an abstract axiomatization of binders, which differs from the syntactic approaches described in the literature (Section 9.3). We also consider infinitely branching and non-well-founded terms; these are seldom considered in conjunction with bindings (Section 9.4). Finally, as a contribution to proof assistant infrastructure for syntax with bindings, our work joins a large body of prior art (Section 9.5).

9.1 Comparison with Nominal Logic

There is considerable overlap between our framework and nominal logic, which is itself a syntax-free axiomatization of term-like entities that can contain variables, called atoms. We can establish a precise correspondence between a specific fragment of our framework and nominal logic. Let $\overline{\alpha}\ F$ be an $\overline{\alpha}$-binding MRBNF (whose inputs are all binding inputs) that is finitary (i.e., $\mathrm{bd}_F = \aleph_0$), and fix $\overline{\alpha}$ to some countable types. Then $\overline{\alpha}\ F$ is a (multi-atom) nominal set with $\overline{\alpha}$ as its sets of atoms. Moreover, our endobijections $f_i : \alpha_i \to \alpha_i$ of small support coincide with what nominal logic calls permutations of finite support, and the map_T function is the same as the nominal permutation action. This is no coincidence: Our MRBNF restriction to small-support functions, as well as our fresh induction proof principle, are inspired by the nominal approach.

Nevertheless, there are important differences. First, we employ functors for modeling the presence of variables, instead of atom-enriched sets. We exploit a mechanism that is already present in the logical foundation: the dependence of type constructors on type variables. Moreover, unlike nominal sets, which are assigned fixed collections of atoms, the inputs to our functors are parameters that can be instantiated in various ways. We exploit this flexibility to remove the finite support restriction and to accept terms that are infinitely branching, that have infinite depth, or both. To accommodate such large entities, we only need to instantiate type variables with suitably large types.

Another difference concerns the amount of theory (structure and properties) that is built into the framework as opposed to developed in an ad hoc fashion. Unlike nominal sets, whose atoms can only be manipulated via bijections, our functors distinguish between binding variables (manipulated via bijections) and free variables (manipulated via possibly nonbijective functions). These functors allow us to apply not only swappings or permutations but arbitrary substitutions.

With our approach, the reasoning and definitional principles associated with binding-aware (co)datatypes are also more abstract and uniform than with nominal logic. For any syntactic framework, the difficulty of providing such an infrastructure increases with the complexity of the

supported binders. For example, Nominal Isabelle includes simple binders supported by induction [Urban and Tasson 2005] and recursion [Urban and Berghofer 2006], whereas Isabelle Nominal2 provides complex binders but only induction. By contrast, our induction and recursion principles operate generically for arbitrary MRBNFs, regardless of the binders' complexity. For finitary syntax, our (co)induction and (co)recursion principles are as expressive as those of nominal logic, via the correspondence between MRBNFs and nominal sets described above; any predicate that is provable or function that is definable using one approach is also provable or definable with the other.

Binding-aware recursion is technically more complex than induction, because we must produce a function that is well defined on alpha-quotiented terms. Here, the state of the art on high expressiveness is the nominal recursor [Pitts 2006] (implemented in Isabelle/HOL [Urban and Berghofer 2006], Coq [Aydemir et al. 2007], and Agda [Copello et al. 2018]) and its essential variations due to Norrish [2004] (in HOL4) and Gheri and Popescu [2017] (in Isabelle/HOL). Our recursor improves on all these recursors, by combining their respective strengths: It supports Norrish's flexible (dynamic) parameters and Gheri and Popescu's improved Horn-style axiomatization, while circumventing the nominal recursor's limitation that freshness must be definable from the permutation action. Our technical report [Blanchette et al. 2019a] formally proves these claims for the λ-calculus.

9.2 Category-Theoretic Approaches

Category theory is a source of highly general and modular representations of datatypes. Our work on endowing Isabelle/HOL with modular (co)datatypes, initiated in 2012 with the introduction of bounded natural functors (BNFs), draws heavily from category theory. As discussed in Traytel et al. [2012, Section III.C], BNFs have been designed as a HOL-friendly class of functors after analyzing various classes from the literature, including set-based functors [Aczel and Mendler 1989], accessible functors [Makkai and Paré 1990], datafunctors [Hensel and Jacobs 1997], container types [Hoogendijk and de Moor 2000], and containers [Abbott et al. 2005]. The last two have inspired the "shape and content" intuition described in Section 2.2. Like Hoogendijk and de Moor's *container types*, BNFs are a class of functors specified extensionally by some additional structure and properties; shapes and contents are handled implicitly, through the properties of functors and relators. By contrast, Abbot et al.'s *containers* are described by a format that explicitly refers to shapes with positions to be filled with content; for the category of sets, containers are a subclass of what we call strong BNFs. Quotient containers [Abbott et al. 2004], introduced for capturing permutative datatypes, form a proper generalization of analytic functors [Joyal 1986] and are a subclass of BNFs. Both containers and BNFs enjoy nice closure properties, which is crucial for modularity. Closure properties have not been studied for container types, which seem to form a superclass of strong BNFs. The BNF natural transformation to sets is also similar to the notion of support in Pierce's [2002] account of (co)inductive types and with the urelement relation employed by Abel and Altenkirch [1999] in their predicative proof of strong normalization for λ-calculus with (co)inductive types.

Among category-theoretic work that addresses bindings specifically, there is a substantial body of work based on presheaves, which are (co- or contravariant) functors $P : \mathcal{C} \to \mathcal{S}et$, where $\mathcal{C}$ is often taken to be (a skeleton of) the category of finite sets having as morphisms arbitrary functions, injections, or bijections. The last case gives the species [Joyal 1981], which have been used in conjunction with analytic functors in the study of combinatorial properties of labeled structures [Bergeron et al. 1997], where bijections rename labels. Our MRBNFs employ a blend of arbitrary functions and bijections, where bijections rename only binding variables.

Representations of syntax with bindings using presheaves [Fiore et al. 1999; Hofmann 1999] follow the nameless paradigm. Terms with bindings form the initial object in a category of algebras over a presheaf topos. For example, in Fiore et al. [1999], which is explicitly inspired by de Bruijn levels, the binding constructor λ is a natural transformation between the context-extended presheaf

of terms and the presheaf of terms. Concretely, this means that λ is a family $(\lambda_n)_{n \in nat}$ such that $\lambda_n : [n+1]\ \boldsymbol{T} \to [n]\ \boldsymbol{T}$, where $[n]\ \boldsymbol{T}$ can be construed as the collection of all terms with free variables among $x_0, \ldots, x_{n-1}$ (the first n variables), the intuition being that λ_n binds the $(n+1)$st variable x_n. A closely related approach is taken by Bird and Paterson [1999] and Altenkirch and Reus [1999]. They use nested datatypes to extend the context, yielding $\lambda : (\alpha + unit)\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$. The same construction is available for Joyal's species using the differentiation operator, as noted by Yorgey [2014, p. 119].

Generalizations of the early presheaf representations have been developed using the (related) concepts of dependent polynomial functors [Gambino and Hyland 2003], generalized species [Fiore et al. 2008], polynomial functors over grupoids [Kock 2012], and indexed containers [Altenkirch et al. 2015; Morris et al. 2009]. Indexed containers feature a stronger type-theoretic view that complements the category-theoretic view. They are extensions of the containers developed with the goal of extending the specification language beyond standard datatypes, to capture bindings, constraints, nested recursion, and dependency on data. Like our MRBNFs, they have been shown to admit least and greatest fixpoints (datatypes and codatatypes) and powerful corresponding (co)induction principles [Ghani et al. 2013].

Recent work [Ahrens et al. 2018] on the univalent foundation of mathematics (having roots in earlier category-theoretic advances [Ghani et al. 2006; Hirschowitz and Maggesi 2012]) defines a general notion of signature for syntax with bindings and identifies criteria for the existence of initial models, again following the nameless paradigm. Like our work, it emphasizes modularity and studies the existence of substitution operators.

Despite being generalizations and type- and scope-safe enhancements of de Bruijn's construction, presheaf-based and related representations reach beyond the nameless paradigm. By virtue of working in a suitable presheaf topos where context extension is isomorphic to the function space from the presheaf of variables, some presheaf representations yield a weak-HOAS-like view [Hofmann 1999]. In addition, the notion of presheaf itself is not committed to a nameless paradigm. In fact, the (quintessentially nameful) category of nominal sets is equivalent to a category of sheaves (pullback-preserving presheaves), namely the Schanuel topos [Johnstone 1983]. However, the nominal induction and recursion principles, which we generalize in this paper, have a different flavor from the principles stemming from presheaf-based representations: The inductive and recursive clauses refer to bound variables, and Barendregt's variable convention is observed. Gabbay and Pitts [2002, Section 7] and Staton [2007, Chapter 7] explain this distinction, also emphasizing the Schanuel topos's classical logic nature.

9.3 Complex Bindings

The literature on specification mechanisms for syntax with bindings offers a wide range of syntactic formats of various levels of sophistication, including those underlying CαMl [Pottier 2006], Ott [Sewell et al. 2010], Unbound [Weirich et al. 2011], Isabelle Nominal2 [Urban and Kaliszyk 2012], and Needle & Knot [Keuchel et al. 2016]. By contrast, we axiomatize binders semantically, via a class of functors. To our knowledge, our approach is the first in which category theory is used not only for constructing (co)datatypes with bindings but also for axiomatizing complex binders.

Scope graphs [van Antwerpen et al. 2016] are a language-independent framework for specifying bindings. This research focuses on the integration of bindings with compilers and static analyzers.

9.4 Non-Well-Founded Terms with Bindings

Infinitely branching and non-well-founded nameless λ-terms have been studied in the context of infinitary higher-order rewriting and proof theory [Joachimski 2001]. Moreover, corecursion and coinduction techniques for non-well-founded terms with bindings have been developed for presheaves [Matthes and Uustalu 2004] and also fall within the scope of the work on indexed

containers and fibrational coinduction [Ghani et al. 2013]. Kurz et al. have studied nameful corecursion for Böhm trees [2012] and more generally for nominal codatatypes associated with a binding signature [2013]. They also showed how definitions from the theory of the infinitary λ-calculus [Kennaway et al. 1997], including substitution and various normal forms, can be rigorously cast as corecursive definitions. The main difference from our work is that they stay in a nominal setting by restricting the non-well-founded terms to have finitely many free variables, whereas we avoid this restriction by resorting to transfinite cardinals. Going beyond finite support has also been proposed by Gabbay [2007] as an infinitary extension of nonstandard set-theoretic foundations for nominal logic [Gabbay and Pitts 1999, 2002].

9.5 Proof Assistant Representations of Bindings

Much of the work we have mentioned so far has been formalized or implemented in proof assistants or logical frameworks, which has become a scientific norm for syntax with bindings. HOAS is supported by dedicated frameworks (e.g., Abella [Baelde et al. 2014], Beluga [Pientka 2010], Delphin [Poswolsky and Schürmann 2009] and Twelf [Pfenning and Schürmann 1999]) and is also implemented in general-purpose proof assistants such as Coq [Chlipala 2008; Despeyroux et al. 1995] and Isabelle/HOL [Gunter et al. 2009].

Indexed containers are particularly suitable for dependent type theories and have been formalized in Agda and deployed for generic programming in the dependently typed language Epigram [McBride 2004]. An expressive type theory such as Agda's or Coq's caters for generic developments using universes populated by codes for types. Besides containers, universe-based developments include the works of Keuchel and Jeuring [2012] (using HOAS and nameless representations), Keuchel et al. [2016], Licata and Harper [2009], Allais et al. [2018], Lee et al. [2012] (using nameless representations), and Copello et al. [2018] (using a nameful representation). Some of these support formats for complex binder patterns and feature a substantial amount of infrastructure lemmas and formalized case studies. The work of Copello et al. is most similar to ours in that it provides nameful, Barendregt-style definition and reasoning principles. Their terms, like ours, are constructed from raw terms via an alpha-equivalence relation operating on elements of functors, but the construction has a more concrete flavor, because their functors are generated by a grammar whereas ours are described abstractly by properties. All these works have been performed in a constructive setting, in contrast to ours, which relies on classical logic. It is not clear if our approach can be adapted to constructive type theory; a prerequisite would be the availability of (co)datatypes based explicitly on functors, as those recently designed for Cedille [Firsov and Stump 2018] and Lean.

An alternative to generic programming via universes is the definitional package approach, which is available in proof assistants based on a weaker logic such as HOL. Definitional packages produce the datatypes and associated theorems dynamically for each user-specified syntax. This is the approach taken for Isabelle Nominal but also for Coq's Autosubst tool [Schäfer et al. 2015]. We plan to use our MRBNF formalization as the basis of a definitional package that will extend Isabelle's BNF-based (co)datatypes [Blanchette et al. 2014]. We are looking forward to taking on POPLmark [Aydemir et al. 2005] and its follow-up challenges [Abel et al. 2017; Felty et al. 2015b].

Acknowledgments. The paper and its formalization have benefited from detailed comments by six reviewers and two artifact evaluators at POPL and four reviewers at ICFP, whose wit and dedication we highly appreciate. It also benefited from Mark Summerfield's many suggestions. Blanchette received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant No. 713999, Matryoshka). Gheri was partially supported by the German Federal Ministry of Education and Research (BMBF) and by the Hessen State Ministry for Higher Education, Research, and the Arts (HMWK) within CRISP. Popescu received funding from UK's Engineering and Physical Sciences Research Council (EPSRC) via grant EP/N019547/1, Verification of Web-based Systems (VOWS). The authors are listed alphabetically.

REFERENCES

Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing Strictly Positive Types. *Theor. Comput. Sci.* 342, 1 (2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

Michael Gordon Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. 2004. Constructing Polymorphic Programs with Quotient Types. In *Mathematics of Program Construction (MPC) 2004*, Dexter Kozen and Carron Shankland (Eds.). LNCS, Vol. 3125. Springer, 2–15. https://doi.org/10.1007/978-3-540-27764-4_2

Andreas Abel and Thorsten Altenkirch. 1999. A Predicative Strong Normalisation Proof for a λ-Calculus with Interleaving Inductive Types. In *Types for Proofs and Programs (TYPES) 1999*, Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan M. Smith (Eds.). LNCS, Vol. 1956. Springer, 21–40. https://doi.org/10.1007/3-540-44557-9_2

Andreas Abel, Alberto Momigliano, and Brigitte Pientka. 2017. POPLMark Reloaded. In *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP) 2017*, Marino Miculan and Florian Rabe (Eds.). https://lfmtp.org/workshops/2017/inc/papers/paper_8_abel.pdf

Peter Aczel and Nax Paul Mendler. 1989. A Final Coalgebra Theorem. In *Category Theory and Computer Science 1989*, David H. Pitt, David E. Rydeheard, Peter Dybjer, Andrew M. Pitts, and Axel Poigné (Eds.). LNCS, Vol. 389. Springer, 357–365. https://doi.org/10.1007/BFb0018361

Benedikt Ahrens, André Hirschowitz, Ambroise Lafont, and Marco Maggesi. 2018. High-Level Signatures and Initial Semantics. In *Computer Science Logic (CSL) 2018*, Dan R. Ghica and Achim Jung (Eds.). LIPIcs, Vol. 119. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 4:1–4:22. https://doi.org/10.4230/LIPIcs.CSL.2018.4

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *Proc. ACM Program. Lang.* 2, International Conference on Functional Programming (ICFP) (2018), 90:1–90:30. http://doi.acm.org/10.1145/3236785

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed Containers. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S095679681500009X

Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms using Generalized Inductive Types. In *Computer Science Logic (CSL) 1999*, Jörg Flum and Mario Rodríguez-Artalejo (Eds.). LNCS, Vol. 1683. Springer, 453–468. https://doi.org/10.1007/3-540-48168-0_32

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics (TPHOLs) 2005*, Joe Hurd and Thomas F. Melham (Eds.). LNCS, Vol. 3603. Springer, 50–65. https://doi.org/10.1007/11541868_4

Brian E. Aydemir, Aaron Bohannon, and Stephanie Weirich. 2007. Nominal Reasoning Techniques in Coq (Extended Abstract). *Electr. Notes Theor. Comput. Sci.* 174, 5 (2007), 69–77. https://doi.org/10.1016/j.entcs.2007.01.028

Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Principles of Programming Languages (POPL) 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 3–15. https://doi.org/10.1145/1328438.1328443

David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. 2014. Abella: A System for Reasoning about Relational Specifications. *J. Formalized Reasoning* 7, 2 (2014), 1–89. https://doi.org/10.6092/issn.1972-5787/4650

Henk P. Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic, Vol. 40. Elsevier.

François Bergeron, Gilbert Labelle, and Pierre Leroux. 1997. *Combinatorial Species and Tree-Like Structures*. Number 67 in Encyclopedia of Mathematics and Its Applications. Cambridge University Press. Translated by Margaret Readdy.

Stefan Berghofer and Christian Urban. 2007. A Head-to-Head Comparison of de Bruijn Indices and Names. *Electr. Notes Theor. Comput. Sci.* 174, 5 (2007), 53–67. https://doi.org/10.1016/j.entcs.2007.01.018

Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.* 9, 1 (1999), 77–91. https://doi.org/10.1017/S0956796899003366

Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. 2019a. Bindings as Bounded Natural Functors (Extended Version). https://github.com/dtraytel/Bindings-as-BNFs/blob/master/bindings_extended.pdf.

Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. 2019b. Formalization associated with this paper. https://github.com/dtraytel/Bindings-as-BNFs.

Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. 2014. Truly Modular (Co)datatypes for Isabelle/HOL. In *Interactive Theorem Proving (ITP) 2014*, Gerwin Klein and Ruben Gamboa (Eds.). LNCS, Vol. 8558. Springer, 93–110. https://doi.org/10.1007/978-3-319-08970-6_7

Jasmin Christian Blanchette, Fabian Meier, Andrei Popescu, and Dmitriy Traytel. 2017. Foundational Nonuniform (Co)datatypes for Higher-Order Logic. In *Logic in Computer Science (LICS) 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005071

Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reasoning* 49, 3 (2012), 363–408. https://doi.org/10.1007/s10817-011-9225-2

Adam Chlipala. 2008. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *International Conference on Functional Programming (ICFP) 2008*, James Hook and Peter Thiemann (Eds.). ACM, 143–156. https://doi.org/10.1145/1411204.1411226

Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *J. Symb. Log.* 5, 2 (1940), 56–68. https://doi.org/10.2307/2266170

Ernesto Copello, Nora Szasz, and Álvaro Tasistro. 2018. Formalisation in Constructive Type Theory of Barendregt's Variable Convention for Generic Structures with Binders. In *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP) 2018*, Frédéric Blanqui and Giselle Reis (Eds.). EPTCS, Vol. 274. 11–26. https://doi.org/10.4204/EPTCS.274.2

N. G. de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church–Rosser Theorem. *Indag. Math* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. 1995. Higher-Order Abstract Syntax in Coq. In *Typed Lambda Calculi and Applications (TLCA) 1995*, Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin (Eds.). LNCS, Vol. 902. Springer, 124–138. https://doi.org/10.1007/BFb0014049

Amy P. Felty and Alberto Momigliano. 2012. Hybrid: A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax. *J. Autom. Reasoning* 48, 1 (2012), 43–105. https://doi.org/10.1007/s10817-010-9194-x

Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. 2015a. The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 2—a Survey. *J. Autom. Reasoning* 55, 4 (2015), 307–372. https://doi.org/10.1007/s10817-015-9327-3

Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. 2015b. An Open Challenge Problem Repository for Systems Supporting Binders. In *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP) 2015*, Iliano Cervesato and Kaustuv Chaudhuri (Eds.). EPTCS, Vol. 185. 18–32. https://doi.org/10.4204/EPTCS.185.2

Amy P. Felty and Brigitte Pientka. 2010. Reasoning with Higher-Order Abstract Syntax and Contexts: A Comparison. In *Interactive Theorem Proving (ITP) 2010*, Matt Kaufmann and Lawrence C. Paulson (Eds.). LNCS, Vol. 6172. Springer, 227–242. https://doi.org/10.1007/978-3-642-14052-5_17

Francisco Ferreira and Brigitte Pientka. 2017. Programs Using Syntax with First-Class Binders. In *European Symposium on Programming (ESOP) 2017*, Hongseok Yang (Ed.). LNCS, Vol. 10201. Springer, 504–529. https://doi.org/10.1007/978-3-662-54434-1_19

M. Fiore, N. Gambino, M. Hyland, and G. Winskel. 2008. The Cartesian Closed Bicategory of Generalised Species of Structures. *J. London Math. Soc.* 77, 1 (2008), 203–220. https://doi.org/10.1112/jlms/jdm096

Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding. In *Logic in Computer Science (LICS) 1999*. IEEE Computer Society, 193–202. https://doi.org/10.1109/LICS.1999.782615

Denis Firsov and Aaron Stump. 2018. Generic Derivation of Induction for Impredicative Encodings in Cedille. In *Certified Programs and Proofs (CPP) 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 215–227. http://doi.acm.org/10.1145/3167087

Murdoch Gabbay. 2007. A General Mathematics of Names. *Inf. Comput.* 205, 7 (2007), 982–1011. https://doi.org/10.1016/j.ic.2006.10.010

Murdoch Gabbay and Andrew M. Pitts. 1999. A New Approach to Abstract Syntax Involving Binders. In *Logic in Computer Science (LICS) 1999*. IEEE Computer Society, 214–224. https://doi.org/10.1109/LICS.1999.782617

Murdoch Gabbay and Andrew M. Pitts. 2002. A New Approach to Abstract Syntax with Variable Binding. *Formal Asp. Comput.* 13, 3-5 (2002), 341–363. https://doi.org/10.1007/s001650200016

Nicola Gambino and Martin Hyland. 2003. Wellfounded Trees and Dependent Polynomial Functors. In *Types for Proofs and Programs (TYPES) 2003*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). LNCS, Vol. 3085. Springer, 210–225. https://doi.org/10.1007/978-3-540-24849-1_14

Neil Ghani, Patricia Johann, and Clément Fumex. 2013. Indexed Induction and Coinduction, Fibrationally. *Logical Methods in Computer Science* 9, 3 (2013). https://doi.org/10.2168/LMCS-9(3:6)2013

Neil Ghani, Tarmo Uustalu, and Makoto Hamana. 2006. Explicit Substitutions and Higher-Order Syntax. *Higher-Order and Symbolic Computation* 19, 2-3 (2006), 263–282. https://doi.org/10.1007/s10990-006-8748-4

Lorenzo Gheri and Andrei Popescu. 2017. A Formalized General Theory of Syntax with Bindings. In *Interactive Theorem Proving (ITP) 2017*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.). LNCS, Vol. 10499. Springer, 241–261. https://doi.org/10.1007/978-3-319-66107-0_16

M. J. C. Gordon and T. F. Melham (Eds.). 1993. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.

Elsa L. Gunter, Christopher J. Osborn, and Andrei Popescu. 2009. Theory Support for Weak Higher Order Abstract Syntax in Isabelle/HOL. In *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP) 2009*, James Cheney and Amy P. Felty (Eds.). ACM, 12–20. https://doi.org/10.1145/1577824.1577827

Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1987. A Framework for Defining Logics. In *Logic in Computer Science (LICS) 1987*. IEEE Computer Society, 194–204. https://doi.org/10.1145/138027.138060

Matthew Hennessy and Robin Milner. 1980. On Observing Nondeterminism and Concurrency. In *International Colloquium on Automata, Languages, and Programming (ICALP) 1980*, J. W. de Bakker and Jan van Leeuwen (Eds.). LNCS, Vol. 85. Springer, 299–309. https://doi.org/10.1007/3-540-10003-2_79

Ulrich Hensel and Bart Jacobs. 1997. Proof Principles for Datatypes with Iterated Recursion. In *Category Theory and Computer Science 1997*, Eugenio Moggi and Giuseppe Rosolini (Eds.). LNCS, Vol. 1290. Springer, 220–241. https://doi.org/10.1007/BFb0026991

André Hirschowitz and Marco Maggesi. 2012. Initial Semantics for Strengthened Signatures. In *Fixed Points in Computer Science (FICS) 2012*, Dale Miller and Zoltán Ésik (Eds.). EPTCS, Vol. 77. 31–38. https://doi.org/10.4204/EPTCS.77.5

Martin Hofmann. 1999. Semantical Analysis of Higher-Order Abstract Syntax. In *Logic in Computer Science (LICS) 1999*. IEEE Computer Society, 204–213. https://doi.org/10.1109/LICS.1999.782616

Paul F. Hoogendijk and Oege de Moor. 2000. Container Types Categorically. *J. Funct. Program.* 10, 2 (2000), 191–225. https://doi.org/10.1017/S0956796899003640

Gérard P. Huet and Bernard Lang. 1978. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Inf.* 11 (1978), 31–55. https://doi.org/10.1007/BF00264598

Felix Joachimski. 2001. *Reduction Properties of* IIIE*-Systems*. Ph.D. Dissertation. LMU München.

P. T. Johnstone. 1983. Quotients of Decidable Objects in a Topos. *Math. Proc. Cambridge Philosophical Society* 93 (1983), 409–419. https://doi.org/10.1017/S0305004100060734

André Joyal. 1981. Une théorie combinatoire des séries formelles. *Adv. Math.* 42 (1981), 1–82. https://doi.org/10.1016/0001-8708(81)90052-9

André Joyal. 1986. Foncteurs analytiques et espèces de structures. In *Combinatoire Énumérative*. LNM, Vol. 1234. Springer, 126–159. https://doi.org/10.1007/BFb0072514

Jonas Kaiser, Brigitte Pientka, and Gert Smolka. 2017. Relating System F and $\lambda 2$: A Case Study in Coq, Abella and Beluga. In *Formal Structures for Computation and Deduction (FSCD) 2017*, Dale Miller (Ed.). LIPIcs, Vol. 84. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 21:1–21:19. https://doi.org/10.4230/LIPIcs.FSCD.2017.21

H. Jerome Keisler. 1971. *Model Theory for Infinitary Logic: Logic with Countable Conjunctions and Finite Quantifiers*. Studies in Logic and the Foundations of Mathematics, Vol. 62. North Holland.

Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. 1997. Infinitary Lambda Calculus. *Theor. Comput. Sci.* 175, 1 (1997), 93–125. https://doi.org/10.1016/S0304-3975(96)00171-5

Steven Keuchel and Johan Jeuring. 2012. Generic Conversions of Abstract Syntax Representations. In *Workshop on Generic Programming 2012*, Andres Löh and Ronald Garcia (Eds.). ACM, 57–68. https://doi.org/10.1145/2364394.2364403

Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. 2016. Needle & Knot: Binder Boilerplate Tied Up. In *European Symposium on Programming (ESOP) 2016*, Peter Thiemann (Ed.). LNCS, Vol. 9632. Springer, 419–445. https://doi.org/10.1007/978-3-662-49498-1_17

Joachim Kock. 2012. Data Types with Symmetries and Polynomial Functors over Groupoids. *Electr. Notes Theor. Comput. Sci.* 286 (2012), 351–365. https://doi.org/10.1016/j.entcs.2013.01.001

Ondřej Kunčar and Andrei Popescu. 2018. Safety and Conservativity of Definitions in HOL and Isabelle/HOL. *Proc. ACM Program. Lang.* 2, Principles of Programming Languages (POPL) (2018), 24:1–24:26. https://doi.org/10.1145/3158112

Alexander Kurz, Daniela Petrisan, Paula Severi, and Fer-Jan de Vries. 2012. An Alpha-Corecursion Principle for the Infinitary Lambda Calculus. In *Coalgebraic Methods in Computer Science (CMCS) 2012*, Dirk Pattinson and Lutz Schröder (Eds.). LNCS, Vol. 7399. Springer, 130–149. https://doi.org/10.1007/978-3-642-32784-1_8

Alexander Kurz, Daniela Petrisan, Paula Severi, and Fer-Jan de Vries. 2013. Nominal Coalgebraic Data Types with Applications to Lambda Calculus. *Logical Methods in Computer Science* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:20)2013

Gyesik Lee, Bruno C. d. S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. 2012. GMeta: A Generic Formal Metatheory Framework for First-Order Representations. In *European Symposium on Programming (ESOP) 2012*, Helmut Seidl (Ed.). LNCS, Vol. 7211. Springer, 436–455. https://doi.org/10.1007/978-3-642-28869-2_22

Pierre Lescanne and Jocelyne Rouyer-Degli. 1995. Explicit Substitutions with de Bruijn's Levels. In *Rewriting Techniques and Applications (RTA) 1995*, Jieh Hsiang (Ed.). LNCS, Vol. 914. Springer, 294–308. https://doi.org/10.1007/3-540-59200-8_65

Daniel R. Licata and Robert Harper. 2009. A Universe of Binding and Computation. In *International Conference on Functional Programming (ICFP) 2009*, Graham Hutton and Andrew P. Tolmach (Eds.). ACM, 123–134. https://doi.org/10.1145/1596550.1596571

Michael Makkai and Robert Paré. 1990. *Accessible Categories: The Foundations of Categorical Model Theory*. Contemporary Mathematics, Vol. 104. American Mathematical Society.

Ralph Matthes and Tarmo Uustalu. 2004. Substitution in Non-Wellfounded Syntax with Variable Binding. *Theor. Comput. Sci.* 327, 1-2 (2004), 155–174. https://doi.org/10.1016/j.tcs.2004.07.025

Conor McBride. 2004. Epigram: Practical Programming with Dependent Types. In *Advanced Functional Programming 2004*, Varmo Vene and Tarmo Uustalu (Eds.). LNCS, Vol. 3622. Springer, 130–170. https://doi.org/10.1007/11546382_3

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Robin Milner. 1989. *Communication and Concurrency*. Prentice Hall.

Robin Milner. 1999. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press.

Peter Morris, Thorsten Altenkirch, and Neil Ghani. 2009. A Universe of Strictly Positive Families. *Int. J. Found. Comput. Sci.* 20, 1 (2009), 83–107. https://doi.org/10.1142/S0129054109006462

Bengt Nordström, Kent Petersson, and Jan M. Smith. 1990. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press.

Michael Norrish. 2004. Recursive Function Definition for Types with Binders. In *Theorem Proving in Higher Order Logics (TPHOLs) 2004*, Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan (Eds.). LNCS, Vol. 3223. Springer, 241–256. https://doi.org/10.1007/978-3-540-30142-4_18

Lawrence C. Paulson. 1989. The Foundation of a Generic Theorem Prover. *J. Autom. Reasoning* 5, 3 (1989), 363–397. https://doi.org/10.1007/BF00248324

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Programming Language Design and Implementation (PLDI) 1988*, Richard L. Wexelblat (Ed.). ACM, 199–208. https://doi.org/10.1145/53990.54010

Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf—A Meta-Logical Framework for Deductive Systems. In *Conference on Automated Deduction (CADE) 1999*, Harald Ganzinger (Ed.). LNCS, Vol. 1632. Springer, 202–206. https://doi.org/10.1007/3-540-48660-7_14

Brigitte Pientka. 2010. Beluga: Programming with Dependent Types, Contextual Data, and Contexts. In *Functional and Logic Programming (FLOPS) 2010*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.). LNCS, Vol. 6009. Springer, 1–12. https://doi.org/10.1007/978-3-642-12251-4_1

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

A. Pitts. 1993. In *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, Chapter The HOL Logic, 191–232.

Andrew M. Pitts. 2003. Nominal Logic, a First Order Theory of Names and Binding. *Inf. Comput.* 186, 2 (2003), 165–193. https://doi.org/10.1016/S0890-5401(03)00138-X

Andrew M. Pitts. 2006. Alpha-Structural Recursion and Induction. *J. ACM* 53, 3 (2006), 459–506. https://doi.org/10.1145/1147954.1147961

Andrei Popescu, Elsa L. Gunter, and Christopher J. Osborn. 2010. Strong Normalization for System F by HOAS on Top of FOAS. In *Logic in Computer Science (LICS) 2010*. IEEE Computer Society, 31–40. https://doi.org/10.1109/LICS.2010.48

Adam Poswolsky and Carsten Schürmann. 2009. System Description: Delphin—A Functional Programming Language for Deductive Systems. *Electr. Notes Theor. Comput. Sci.* 228 (2009), 113–120. https://doi.org/10.1016/j.entcs.2008.12.120

François Pottier. 2006. An Overview of Cαml. *Electr. Notes Theor. Comput. Sci.* 148, 2 (2006), 27–52. https://doi.org/10.1016/j.entcs.2005.11.039

Jan J. M. M. Rutten. 1998. Relators and Metric Bisimulations. *Electr. Notes Theor. Comput. Sci.* 11 (1998), 252–258. https://doi.org/10.1016/S1571-0661(04)00063-5

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving (ITP) 2015*, Christian Urban and Xingyuan Zhang (Eds.). LNCS, Vol. 9236. Springer, 359–374. https://doi.org/10.1007/978-3-319-22102-1_24

Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. 2001. Primitive Recursion for Higher-Order Abstract Syntax. *Theor. Comput. Sci.* 266, 1-2 (2001), 1–57. https://doi.org/10.1016/S0304-3975(00)00418-7

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2010. Ott: Effective Tool Support for the Working Semanticist. *J. Funct. Program.* 20, 1 (2010), 71–122. https://doi.org/10.1017/S0956796809990293

Sam Staton. 2007. *Name-Passing Process Calculi: Operational Models and Structural Operational Semantics*. Technical Report UCAM-CL-TR-688. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-688.pdf

Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. 2012. Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In *Logic in Computer Science (LICS) 2012*. IEEE Computer Society, 596–605. https://doi.org/10.1109/LICS.2012.75

Christian Urban and Stefan Berghofer. 2006. A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL. In *International Joint Conference on Automated Reasoning (IJCAR) 2006*, Ulrich Furbach and Natarajan Shankar (Eds.). LNCS, Vol. 4130. Springer, 498–512. https://doi.org/10.1007/11814771_41

Christian Urban, Stefan Berghofer, and Michael Norrish. 2007. Barendregt's Variable Convention in Rule Inductions. In *Conference on Automated Deduction (CADE) 2007*, Frank Pfenning (Ed.). LNCS, Vol. 4603. Springer, 35–50. https:

//doi.org/10.1007/978-3-540-73595-3_4

Christian Urban and Cezary Kaliszyk. 2012. General Bindings and Alpha-Equivalence in Nominal Isabelle. *Logical Methods in Computer Science* 8, 2 (2012). https://doi.org/10.2168/LMCS-8(2:14)2012

Christian Urban and Christine Tasson. 2005. Nominal Techniques in Isabelle/HOL. In *Conference on Automated Deduction (CADE) 2005*, Robert Nieuwenhuis (Ed.). LNCS, Vol. 3632. Springer, 38–53. https://doi.org/10.1007/11532231_4

Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A Constraint Language for Static Semantic Analysis Based on Scope Graphs. In *Partial Evaluation and Program Manipulation (PEPM) 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 49–60. https://doi.org/10.1145/2847538.2847543

Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. 2011. Binders Unbound. In *International Conference on Functional Programming (ICFP) 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 333–345. https://doi.org/10.1145/2034773.2034818

Brent Abraham Yorgey. 2014. *Combinatorial Species and Labelled Structures.* Ph.D. Dissertation. University of Pennsylvania.