Bindings Are 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; Interactive proof systems**;

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 Are Functors. *Proc. ACM Program. Lang.* 0, 0, Article 0 (July 2019), 39 pages.

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., Chlipala [2008]; Felty et al. [2015]; Kaiser et al. [2017]; Pitts [2006]; 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 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.

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.

2019. 2475-1421/2019/7-ART0 $15.00
https://doi.org/

To improve the situation, we believe that the missing ingredient is a semantics-based understanding of the binding and substitution mechanisms, as opposed to ever more general syntactic formats. In this paper, we aim at identifying the fundamental laws that guide binding and substitution and expressing them in a syntax-free manner. 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 alpha-equivalence (naming equivalence) and capture-free substitution, amount to replacing 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 *functors* on sets. We employ different notions of morphisms depending on the task: arbitrary functions when substituting free variables, bijections when renaming bound variables, 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 gracefully 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). The current work can be regarded as the journey of transforming BNFs into binding-aware entities. 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 have identified a "sweet spot" between the assumptions and the guarantees involved in the construction of datatypes.

We equip our binding-aware (co)datatypes with a powerful arsenal of reasoning and definitional principles that follow the (co)datatypes' structure. We generalize fresh induction in the style of nominal logic and propose new coinduction principles (Section 6). Moreover, we provide (co)recursion principles that improve 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).

This work improves on previous abstract frameworks such as nominal logic and other category-theoretic approaches in terms of generality, flexibility, and modularity (Section 9). 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 the supplementary material (appendix and formalization) for more formal details, including proofs.

Succinctly, the main contributions of this paper are as follows:

- We propose a "bindings as functors" approach that subsumes and extends the syntactic formats in the literature, and that features capture-free substitutions as fist-class citizens.

- By allowing infinite support, our framework supports non-well-founded and infinitely branching types that lie beyond nominal logic's reach.
- All our type definitions are equipped with (co)induction and (co)recursion principles, which are expressed in a uniform, modular way that insulates from the complexity of bindings.

Although our work is grounded in HOL, in principle any logical foundation that is rich enough to model the BNFs can be used. The framework could be adapted to proof assistants beyond Isabelle/HOL and the HOL family of systems, and would be particularly suitable to systems that already offer functor-based (co)datatypes, such as Cedille [Firsov and Stump 2018].

2 PRELIMINARIES

Our work relies on HOL and on BNFs, a category-theoretic approach to defining and reasoning about types in a modular way.

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.

Primitive *types* are built from type variables α, β, ..., a type *bool* of Booleans, and an infinite type *ind* using the function type constructor; for example, $(bool \to \alpha) \to ind$ is a type. Unlike in dependent type theory, all types are inhabited (nonempty). Primitive *constants* are equality $= : \alpha \to \alpha \to bool$, the Hilbert choice operator, and 0 and Suc for *ind*. Terms are built from constants and variables by means of typed λ-abstraction and application.

A *polymorphic type* is a type T that contains type variables. If T is polymorphic with variables $\overline{\alpha} = (\alpha_1, \ldots, \alpha_n)$, we sometimes write $\overline{\alpha}\ T$ instead of T. An *instance* of a polymorphic type is obtained by replacing some of its type variables with other types. For example, $(\alpha \to bool) \to \alpha$ is a polymorphic type, and $(ind \to bool) \to ind$ is an instance of it. A *polymorphic function* is a function that has a polymorphic type—for example, $\mathrm{Cons} : \alpha \to \alpha\ list \to \alpha\ list$. 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 notation simple, we use type variables in two different ways: (1) as part of polymorphic types; and (2) as arbitrary but fixed types. For example, we can write (1) "α *list* is a polymorphic type" and (2) "given any type α and any function $f : \alpha \to \alpha$, such and such holds." These conventions allow us to express the concepts in a more "semantic" style, closer to standard mathematical notation. The reader is free to think of types as nonempty sets.

Unlike dependent type theory, HOL does not have (co)datatypes as primitives. The only primitive for defining new types in HOL is the *typedef* mechanism: which roughly corresponds to set comprehension in set theory: For any given type $\overline{\alpha}\ T$ and nonempty predicate $P : \overline{\alpha}\ T \to bool$, we can carve out a new type $\{x : \overline{\alpha}\ T \mid P\ x\}$ consisting of all members of $\overline{\alpha}\ T$ satisfying P. (Co)datatypes are supported via derived specification mechanisms.

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 ($\times$) 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].

We write $[n]$ for $\{1, \ldots, n\}$ and $\alpha\ set$ for the powertype of α, consisting of sets of elements of α.

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

- F is an n-ary type constructor;
- $\mathsf{map}_F : (\alpha_1 \to \alpha_1') \to \cdots \to (\alpha_n \to \alpha_n') \to \overline{\alpha}\, F \to \overline{\alpha'}\, F$;
- $\mathsf{set}_F^i : \overline{\alpha}\, F \to \alpha_i\ set$ 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) $\mathsf{rel}_F\ \overline{R}\ x\ y \longleftrightarrow$
$\exists z.\ (\forall i \in [n].\ \mathsf{set}_F^i\ z \subseteq \{(a, a') \mid R_i\ a\ b\})\ \wedge\ \mathsf{map}_F\ [\mathsf{fst}]^n\ z = x \wedge \mathsf{map}_F\ [\mathsf{snd}]^n\ z = y$

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

(Fun) (F, map_F) is an n-ary functor—i.e., map_F commutes with function composition and preserves the identities

(Nat) each set_F^i is a natural transformation between the functor (F, map_F) and the powerset functor (set, image)

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

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

(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_F^i 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.

Consider a unary BNF F. For a fixed α, we represent a typical element of $x : \alpha\, F$ as depicted in Fig. 1, where we indicate the container as a triangle and its content via a typical atom $a : \alpha$. The left-hand side of the figure 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 $R : \alpha \to \alpha' \to bool$ states that each a is R-related to an a' located at the same position in the container.

As an example, *list* is a unary BNF, where map_{list} is the standard map function, set_{list} collects all the elements of a list, and bd_{list} is $\aleph_0$. Moreover, $\mathsf{rel}_{list}\ R\ xs\ ys$ states that xs and ys have the same length and are elementwise related by R.

2.3 (Co)datatypes from Bounded Natural Functors

A *strong* BNF is a BNF whose map preserves not only weak pullbacks but also strong pullbacks. Strong BNFs include the basic type constructors of sum, product, and positive function space.

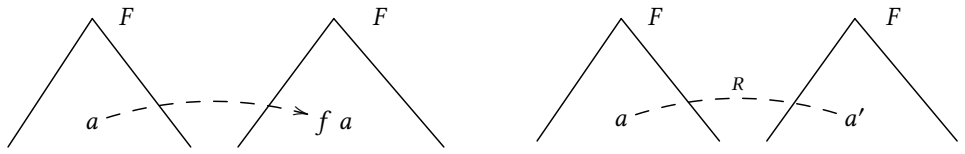


Fig. 1. $\mathsf{map}_F\ f$ (left) and $\mathsf{rel}_F\ R$ (right)

Examples of BNFs that are not strong are the permutative (nonfree) type constructors, such as the finite powertype denoted α *fset* and the type of finite multisets (bags). Both the BNFs and the strong BNFs are closed under composition and (least and greatest) fixpoint definitions [Traytel et al. 2012]. This enables us to mix 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 instead we interpret the equation maximally—which we will indicate with the superscript ∞—it yields the codatatype $\overline{\alpha}\ T$. In either case, the construction yields an $\overline{\alpha}$-polymorphic bijection $\mathsf{ctor} : (\overline{\alpha}, \overline{\alpha}\ T)\ F \to \overline{\alpha}\ T$.

Abstractly, the difference between datatypes and codatatypes lies in their reasoning and definitional principles. For datatypes we have *structural induction*—which allows us to prove a that 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 as $\alpha\ T \simeq (\alpha, \alpha\ T)\ F$ is α *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 in a modular fashion. Because α *llist* is itself a BNF, it can be used in further fixpoint definitions: Taking $(\alpha, \tau)\ F$ to be $\alpha + \tau\ \mathit{llist}$, the datatype $\alpha\ T$ defined as $\alpha\ T \simeq (\alpha, \alpha\ T)\ F$ is the type of α-labeled well-founded infinitely branching rose trees. (Appendix A gives more details.)

3 TOWARDS AN ABSTRACT NOTION OF BINDER

The literature so far has 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?

3.1 Examples of Binders

We first try to extract the essence of binders from examples. The paradigmatic example is the λ-calculus, in which λ-abstraction binds a single variable a in a single term t, obtaining $\lambda a.\ t$. The term t may contain several free variables. If a is one of them, adding the λ-abstraction binds it. Suppose t is $b\ a$ ("b applied to a"), where a and b are distinct variables. After applying the λ constructor to a and t, we obtain $\lambda a.\ b\ a$, 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 (the term where this variable is to be bound).

Other binders take into consideration a wider context than just the body. The "let" construct $\mathsf{let}\ a = t_1\ \mathsf{in}\ t_2$ binds the variable a in the term t_2 without affecting t_1. In the expression $\mathsf{let}\ a = b\ a\ \mathsf{in}\ b\ a$ 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: $\lambda(a, b).\ t$ 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 binds simultaneously two variables (a and b) in three terms (t_1, t_2, and t).

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.

3.2 Abstract Binder Types

Binders distinguish between binding variables and the other entities, typically terms, which could be either inside or outside the scope of the binder. Thus, we can think of a binder type as 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 different types α_i of *binding variables*;
- n different 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 thought of 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.

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.

Section 3.1's examples are modeled as follows (writing α and τ instead of α_1 and τ_1 if $m = 1$):

- 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, $(\alpha, \tau_1, \tau_2)\,F$, needs to 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 $(\alpha, \tau, \tau)\,F$. One may wonder why the binder should care about potential terms that fall outside the scope of its binding variables. The answer is that this could lead to severe lack of precision, as argued 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 α *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 explicitly. By contrast, with our 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, this is our first proposal:

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

As it stands, the 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 look closer at our proposal and try to improve it. By modeling "binder types" not as mere types but as type constructors, we can distinguish between the binder's structure and the variables and potential terms that populate it—that is, between shape and content. This follows our intuition of BNFs (Section 2.2). And indeed, all the type constructors we used in examples of binders seem to be BNFs. So we can 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, this proposal does not work: 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 simultaneously binds a and b in t, we took $(\alpha, \tau)\, F$ to be $\alpha \times \alpha \times \tau$. However, this is imprecise, because we also need a and b to 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 reasons of symmetry, we take the more drastic measure of restricting to *bijective* functions only, which additionally have the same domain and codomain—we call these endobijections. In other words, all the conditions of the BNF definition (Definition 1) remain the same, except that on some of the inputs (which are 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 $\overline{\alpha}$, which means 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, including $(\alpha, \tau)\, F = \{(a, b) : \alpha \times \alpha \mid a \neq b\} \times \tau$, belong to this category. We call this refined 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 Sections 4 and 5: (1) How can nonrepetitive MRBNFs be constructed from possibly repetitive ones? (2) How can MRBNFs be used to define and reason about actual terms with bindings and their fundamental operators?

4 CONSTRUCTING NONREPETITIVE MAP-RESTRICTED BNFS

For BNFs, the question of constructibility has a most satisfactory answer: 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 usual types of list and lazy lists—which are repetitive α-MRBNFs because lists, whether lazy or otherwise, may contain repeated elements.

The absence of good fixpoint behavior implies that complex nonrepetitive MRBNFs cannot be built recursively from simpler components. But we can take an alternative route for building nonrepetitive MRBNFs. We can employ the fixpoint constructions on BNFs, and as a last step we 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 $\alpha\ \mathit{list}$ (built recursively from the BNF $\mathit{unit} + \alpha \times \beta$), we construct the MRBNF of nonrepetitive lists, $\{xs : \alpha\ \mathit{list} \mid \mathsf{nonrep}_{\mathit{list}}\ xs\}$. 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, $\{xs : \alpha\ \mathit{pat} \mid \mathsf{nonrep}_{\mathit{pat}}\ xs\}$. 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 $\mathsf{nonrep}_{\mathit{list}}\ xs$, if no α-atom occurs more than once in it; and similarly for the members of $p : \alpha\ \mathit{pat}$, which are essentially trees with α-labeled leaves.

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, $|\mathsf{set}_{\mathit{list}}\ x|$, 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 is essentially its "shape." So we can define the fact that two members x, x' of $\alpha\ F$ have the same shape, written $\mathsf{sameShape}_F\ x\ x'$, to mean that $\mathsf{rel}_F\ \top\ x\ x'$ holds, where $\top : \alpha \to \alpha \to \mathit{bool}$ is the vacuously true relation that ignores the content. Indeed, 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 (1) x and x' have the same shape and (2) their atoms are positionwise related by R. Condition (2) 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 content (e.g., the labels on their nodes or their branches).

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 that exploits the functorial

structure. 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', so that $x' = \mathsf{map}_F\, f\, x$. This gives the correct result for lists, lazy lists, trees, and in general for any combination of (co)datatypes where each atom has a fixed position in the shape—i.e., for 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, e.g., no finite set but the empty one is nonrepetitive. However, these structures do not require any nonrepetitiveness revision, being useful as they are. For example, Hindley–Milner type schemes bind finite sets of variables.

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 an $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F$, 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 kinds of inputs.

Actual terms can be defined by means of a datatype construction framed by F. For simplicity, we will define a single type of terms where 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 (identical) 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 in our abstract scheme 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

- between the free-variable type β and the binding-variable type α—a distinction that 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; and
- between two different types of potential terms, τ_1 and τ_2—a distinction that ensures, via 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, indicate 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 a full functor on $\overline{\tau}$, and to be a functor on $\overline{\alpha}$ with respect to endobijections. But how should it behave on $\overline{\beta}$? A natural answer would be to require full 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' while a was already free in the term. So for now, we will assume full 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.

Next, we will see if this refined proposal is apt for supporting some fundamental constructions on terms. A small running example, exhibiting plenty of binding diversity, will keep us company:

Example 5. Consider a variation of the λ-calculus syntax where 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 ctor x, where $x : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ T]^n)\ F$ has three kinds of atoms.

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

To refer more 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 either $\mathsf{topBind}_i\ x$ if $(i, j) \in \theta$ or $\varnothing$ otherwise. (Since $\mathsf{topBind}_{i,j}$ incorporates the information provided by θ, the latter will be left implicit in our forthcoming constructions.) 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.

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. First, consider the term $t = \mathsf{Var}\ c$. It can be written as ctor x, where $x = \mathsf{Inl}\ c$. Therefore, $\mathsf{topFree}\ x = \{c\}$, $\mathsf{topBind}\ x = \varnothing$, and $\mathsf{rec}\ x = \varnothing$. (We omit the indices 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, t has c as its single free variable: 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 $x = \mathsf{Inr}\ ((a, b), (\mathsf{Var}\ a, \mathsf{Var}\ c))$. Therefore, $\mathsf{topFree}\ x = \varnothing$, $\mathsf{topBind}\ x = \{a, b\}$, and $\mathsf{rec}\ x = \{\mathsf{Var}\ a,\ \mathsf{Var}\ c\}$. Thus, t has no top-free variables, has a and b as top-binding variables, and has $\mathsf{Var}\ a$, $\mathsf{Var}\ c$ 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 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 $\mathsf{map}_T\ \overline{f}\ t_1 \equiv t_2$.

As discussed in Section 2.2, the relators can elegantly express positionwise correspondences as required above. Formally, we define the (infix-applied) alpha-equivalence relation $\equiv : \alpha\ T \to \alpha\ T \to \mathit{bool}$ inductively by the following clause:

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

The clause's 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

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

Returning to the context of Example 5, we first 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 these to be alpha-equivalent by taking f to swap a and b (i.e., send a to b and b to a) and leave all the other variables, including c, unchanged. Verifying the first hypothesis of $\equiv$'s definitional clause 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, $\mathsf{cond}\,(f)$, amounts to checking that f is bijective and that f is the identity on all variables in the set $(\bigcup_{t_1 \in \{\mathsf{Var}\ a,\ \mathsf{Var}\ c\}} \mathsf{FVars}\ t_1) \setminus \{a, b\} = \{a, c\} \setminus \{a, b\} = \{c\}$—i.e., f sends c to c.

The approach above is one of the several possible choices to define alpha-equivalence. We could pose stricter conditions on the f_i, allowing them to change *only* the top-binding variables, whereas we also allow it to change some other (nonfree) variables occurring in the components. In the context of 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 solutions allow f to change a and b, and do not allow 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 of introducing alpha-equivalence have different virtues in terms of the ease or elegance of proving various basic properties, but in the end produce the same concept.

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 $\equiv$ or structural induction on $\overline{\alpha}\ T$.

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

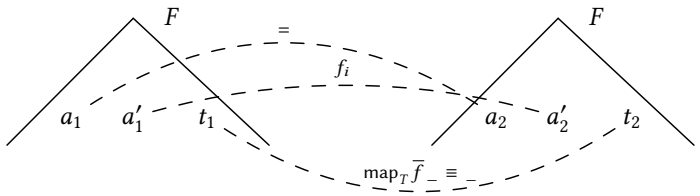


Fig. 2. Alpha-equivalence

- 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 $\mathsf{map}_{\boldsymbol{T}} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$.

Note that, whereas on $\overline{\alpha}\ T$ the constructor is bijective, on $\overline{\alpha}\ \boldsymbol{T}$ it is only surjective. Its injectivity fails due to quotienting, which allows us to bind different variables in different terms but obtain equal results. In our running example, the terms $\lambda(a, b).\,(\mathsf{Var}\ a, \mathsf{Var}\ c)$ and $\lambda(b, a).\,(\mathsf{Var}\ b, \mathsf{Var}\ c)$ are *equal* (in $\alpha\ \boldsymbol{T}$), which means that $\mathsf{ctor}\ x_1 = \mathsf{ctor}\ x_2$, 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 $x_1 \neq x_2$, since $\mathsf{Var}\ a \neq \mathsf{Var}\ b$.

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 $x_1, x_2 : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\,F$, the following are equivalent:

- $\mathsf{ctor}\ x_1 = \mathsf{ctor}\ x_2$;
- there exist functions $f_i : \alpha_i \to \alpha_i$ satisfying $\mathsf{cond}_i\,(f_i)$ for $i \in [m]$ such that $x_2 = \mathsf{map}_F\ [\mathsf{id}]^m\ \overline{f}\ [\mathsf{map}_{\boldsymbol{T}}\,\overline{f}]^n\ x_1$.

Thus, $\overline{\alpha}\ \boldsymbol{T}$ was defined by fixpoint construction framed by F followed by a quotienting construction to a notion of 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 notation

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

to summarize the definition of this binding-aware datatype, where the θ subscript emphasizes that we have an isomorphism up to the alpha-equivalence 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 $\overline{\alpha}\ T$.

Terminology. From now on, we will adhere to the following convention: 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 Substitution

An important operation we want to support on terms $\overline{\alpha}\ \boldsymbol{T}$ is capture-avoiding substitution. With our current infrastructure, we should 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 should 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 only works with bijections f_i. The fundamental desired property of sub concerns its recursive behavior on terms of the form ctor x:

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

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, $\mathsf{sub}\ \overline{f}$ should distribute over the term constructor when neither the term's free variables nor the variables touched 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 as $\mathsf{noClash}_i\ x$, follows a general hygiene principle that is independent of the notion of substitution (and which will be observed by all our proof and definition principles): 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 $\alpha\ \boldsymbol{T}$, ctor x' and ctor x would be equal. By this argument, it would be legitimate to take the premise of ($*$) as true 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 way to ensure that enough fresh variables are available.

5.5 Acquiring Enough Fresh Variables

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 $|\mathsf{supp}\ f_i| < |\alpha_i|$. However, this solution seems to require needlessly many variables when F is finitary—i.e., $\mathsf{bd}_F = \aleph_0$ (the countable cardinal). This is the case for 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 unimportant. Indeed, some textbooks assume only "an infinite supply of variables," without mentioning countability. But countability becomes important when we consider practical aspects such as executability. Therefore, it is worth salvaging countability if we can. It turns out that we can do that with a little insight from the theory of cardinals. We note that the crucial property 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 $\mathsf{bd}_F \leq |\alpha_i|$ if $|\alpha_i|$ is a regular cardinal.

Theorem 8. 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, $\mathsf{bd}_F \leq |\alpha_i|$, 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 α_i's.

5.6 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 (after a renaming which does not affect alpha-equivalence) $\lambda a'.\,(\mathsf{Var}\ a')\,(\lambda c.\,\mathsf{Var}\ a)$. However, not all syntaxes with bindings allow substituting terms for variables. Process terms in the π-calculus [Milner 2001] 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 observation is that, unlike the π-calculus, the λ-calculus can embed single variables into terms. This is 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 $\mathsf{set}_i^F\ (\eta_i\ a) = \{a\}$. Moreover, we assume that η_i is the only source of variables in F, by requiring that $\mathsf{set}_i^F\ x = \varnothing$ 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.

Now we can 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 $\forall i \in I.\ \mathsf{noClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing$. 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 9. There exists a (polymorphic) function $\mathsf{tsub} : (\prod_{i \in I} (\alpha_i \to \overline{\alpha}\ \boldsymbol{T})) \to \overline{\alpha}\ \boldsymbol{T}$ satisfying $(**)$ for all α_i and f_i in case $|\alpha_i|$ is regular, $\mathrm{bd}_F \leq |\alpha_i|$ and $|\mathsf{supp}\ f_i| < |\alpha_i|$.

5.7 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:

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

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 inductive–coinductive asymmetry seems to stand in the way of a duality principle, which would allow 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 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. All the theorems shown in Sections 5.1 to 5.6 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.

5.8 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 $\overline{\alpha}\ \boldsymbol{T} \simeq_\theta (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\ F$. It enjoys the following property:

Theorem 10. $\overline{\alpha}\ \boldsymbol{T}$ is an $\overline{\alpha}$-MRBNF with map function map_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. And indeed, this is 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 $\alpha\ \boldsymbol{T}' \simeq_\theta (\alpha, \alpha, \alpha\ \boldsymbol{T}')\ F'$, which requires full 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 11. 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 $\forall$ 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 vanish if $\boldsymbol{T}$ were a full functor (with respect to arbitrary functions). However, the map function's $\mathsf{map}_{\boldsymbol{T}}$ 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 well known in nominal logic, and is a main reason for this logic's focus on the swapping operator: Swapping a and b respects alpha-equivalence (e.g., starting with $\lambda a.\, a\, b \equiv \lambda c.\, c\, b$ we obtain $\lambda b.\, b\, a \equiv \lambda c.\, c\, a$), whereas substituting a for b (in a capturing fashion) does not (e.g., starting with the same terms as above we obtain $\lambda a.\, a\, a \not\equiv \lambda c.\, c\, a$).

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 $\mathsf{sub} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$. This one 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 a full functor, while the emerging datatype is only a functor with respect to small-support endofunctions.

At this point, it is worth asking whether full functoriality of $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F$ on its free-variable inputs $\overline{\beta}$ was 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, while performing minor changes to the definitions. Assuming that all the functions involved have small support, in particular, adding 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 with respect to arbitrary functions on $\overline{\tau}$. Full 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 Appendix B) achieves useful modularity, in the sense that the free variables of terms are really a free MRBNF component:

Theorem 12. $\overline{\alpha}\ \boldsymbol{T}$ is an $\overline{\alpha}$-free MRBNF with map function sub and set functions FVars_i.

In the next two sections, we complete our stated goal of offering the terms with bindings an abstraction layer, consisting of reasoning and definitional principles, that insulates them completely from the low-level details of raw terms.

6 BINDING-AWARE (CO)INDUCTION PROOF PRINCIPLES

This section is dedicated to reasoning about terms, in their well-founded and non-well-founded incarnations, taking their binding structure into consideration. In what follows, we will implicitly assume that α_i is such that $|\alpha_i|$ is regular and $\mathsf{bd}_F \leq |\alpha_i|$.

6.1 Induction

Next, we take $\overline{\alpha}\ \boldsymbol{T}$ to be type of (well-founded) terms (as in Section 5.3). Let us 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 for it provided that holds for all its 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]:

Theorem 13 (FSI). Let $\mathsf{PFVars}_i : \overline{\alpha}\ P \to \alpha_i\ \mathit{set}$ with $\forall p : \overline{\alpha}\ P.\ |\mathsf{PFVars}_i\ p| < |\alpha_i|$. Given a predicate $\varphi : \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ P \to \mathit{bool}$, if the condition

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

holds, then the following also holds: $\forall t : \overline{\alpha}\ \boldsymbol{T}.\ \forall p : \overline{\alpha}\ P.\ \varphi\ t\ p$.

Above, we highlighted the two differences from 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 is no-clashing 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 14. We have $\mathsf{sub}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m) = \mathsf{sub}\ \overline{g} \circ \mathsf{sub}\ \overline{f}$ 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) the problem is avoided: Taking $(\overline{f}, \overline{g})$ as parameters enables us to assume that capture does not happen and smoothly apply sub's recursive law $(*)$.

6.2 Coinduction

Next, we take $\overline{\alpha}\ \boldsymbol{T}$ to be the type of non-well-founded terms (as in Section 5.7). 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-bismulation (i.e., if it is preserved by F's relator). Using the ideas discussed in Section 6.1, we can prove a parameter-based fresh variation of this principle, where we again emphasize the binding-specific enhancements:

Theorem 15 (FSC). Let $\overline{\alpha}\ P$ and PFVars_i be like in Theorem 13. Given a binary relation $\varphi : \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ P \to \mathit{bool}$, if the condition

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

holds, then the following holds: $\forall t_1, t_2 : \overline{\alpha}\ \boldsymbol{T}.\ \forall p : \overline{\alpha}\ P.\ \varphi\ t_1\ t_2\ p \longrightarrow t_1 = t_2$.

However, this proof principle turns out to be not as useful as its inductive counterpart. Consider the task of proving Proposition 14 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 $\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i$. Define $\varphi\ t_1\ t_2\ (\overline{f}, \overline{g})$ as $\exists t.\ t_1 = \mathsf{sub}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m)\ t\ \wedge$

$t_2 = (\mathsf{sub}\ \overline{g} \circ \mathsf{sub}\ \overline{f})\, t$. Then, it suffices to verify (FSC)'s hypothesis. So we assume that, for all endofunctions of small support f_i and g_i, (1) $\mathsf{ctor}\ x_1 = \mathsf{sub}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m)\ t$ 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 can 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

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

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 on 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 16 (ESC). Given a binary relation $\varphi : \overline{\alpha}\ \boldsymbol{T} \longrightarrow \overline{\alpha}\ \boldsymbol{T} \longrightarrow \mathit{bool}$, if the condition

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

holds, then the following holds: $\forall t_1, t_2 : \overline{\alpha}\ \boldsymbol{T}.\ \varphi\ t_1\ t_2 \longrightarrow t_1 = t_2$.

(ESC) is more general than, and in fact can easily prove, (FSC). It resolves the problem in our concrete example with substitution (because it allows us to dynamically shift 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 aspects have not been formally addressed so far. The first concerns Theorems 8 and 9, 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 to define 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 the category of BNF algebras [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 resolution of both aspects rests on the availability of suitable (co)recursion definitional principles, (*co*)*recursors* for short, for the types of terms. The main technical difficulty in developing such a recursor is that terms do not form a free datatype, which means that defining functions on terms is no longer possible by simply listing some constructor-based recursive clauses. Instead, the recursor must be aware of the nonfreeness introduced by bindings. And a similar (dual) problem holds for the corecursor. The key to address these problem 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. Appendix D 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 17. 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:

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

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

Next, we take $\overline{\alpha}\ \boldsymbol{T}$ to be type of (well-founded) terms. 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 18. 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 additionally have a constructor-like operator.

Definition 19. 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 20. 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) $(\forall i \in [m].\ \mathsf{noClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{PFVars}_i\ p = \varnothing) \longrightarrow$
$H\ (\mathsf{ctor}\ x)\ p = \mathsf{Uctor}\ (\mathsf{map}_F\ [\mathsf{id}]^{2*m}\ [H]^n\ x)\ p$

(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 such a function H, which is additionally 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 (no-clashing) 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, just as the parameter twist of fresh induction (Theorem 13) 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

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

(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 20, we must check its hypotheses, which amount to standard identities on terms. (Appendix C gives more details.) 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) $(\forall i \in [m].\ \mathsf{noClash}_i\ x\ \wedge\ \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing)$
$\longrightarrow\ \mathsf{sub}\ (\mathsf{ctor}\ x)\ \overline{f}\ =\ \mathsf{ctor}\ (\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\lambda pu.\ pu\ \overline{f}]^n\ (\mathsf{map}_F\ [\mathsf{id}]^{2*m}\ [\mathsf{sub}]^n\ x))$

By (restricted) functoriality, $\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\lambda pu.\ pu\ \overline{f}]^n\ (\mathsf{map}_F\ [\mathsf{id}]^{2*m}\ [\mathsf{sub}]^n\ x) = \mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\lambda t.\ \mathsf{sub}\ t\ \overline{f}]^n\ x$, making (C) equivalent to Section 5.4's clause $(*)$, hence proving the desired behavior for substitution (Theorem 8)—that is, 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 5.6) 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 20:

Corollary 21. $(\overline{\alpha}\ \boldsymbol{T}, \overline{\mathsf{FVars}}, \mathsf{map}_{\boldsymbol{T}}, \mathsf{ctor})$ 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 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 dtor : $\overline{\alpha}\ \boldsymbol{T} \to ((\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\ F)\ set$ as dtor $t = \{x \mid t = \text{ctor } x\}$. Crucially, this destructor is still *deterministic up to a renaming* of the top-binding variables. Indeed, Prop. 7 ensures that, for any $x, x' \in \text{dtor } t$, we have $x' = \text{map}_F\ [\text{id}]^m\ \overline{f}\ [\text{map}_{\boldsymbol{T}}\ \overline{f}]^n\ x$ for some small-support endobijections $\overline{f}$ subject to some suitable conditions. This suggests the following axiomatization of corecursive models:

Definition 22. A *comodel* is a quadruple $\mathcal{U} = (\overline{\alpha}\ U, \overline{\text{UFVars}}, \text{Umap}, \text{Udtor})$, where

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

such that the following properties hold:

(Dne) Udtor $d \neq \varnothing$

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

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

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

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)—generalizing properties of the term destructor. Moreover, (MD) generalizes the term property dtor $(\text{map}_{\boldsymbol{T}}\ \overline{f}\ t) \subseteq \text{image } (\text{map}_F\ \overline{f}\ \overline{f}\ [\text{map}_{\boldsymbol{T}}\ \overline{f}]^n)\ (\text{dtor } t)$, which after expanding the definition of dtor from ctor becomes $\text{map}_{\boldsymbol{T}}\ \overline{f}\ t = \text{ctor } x' \longrightarrow \exists x.\ t = \text{ctor } x \wedge x' = \text{map}_F\ \overline{f}\ \overline{f}\ [\text{map}_{\boldsymbol{T}}\ \overline{f}]^n\ x$. 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 $\text{map}_{\boldsymbol{T}}$ and map_F we can rewrite the property into

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

then into $t = \text{map}_{\boldsymbol{T}}\ \overline{g}\ (\text{ctor } x') \longrightarrow \exists x.\ t = \text{ctor } x \wedge \text{map}_F\ \overline{g}\ \overline{g}\ [\text{map}_{\boldsymbol{T}}\ \overline{g}]^n\ x' = x$ and finally into $\text{map}_{\boldsymbol{T}}\ \overline{g}\ (\text{ctor } x') = \text{ctor } (\text{map}_F\ \overline{g}\ \overline{g}\ [\text{map}_{\boldsymbol{T}}\ \overline{g}]^n\ x')$. 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 $\forall i \in [m].\ \text{FVars}_i\ (\text{ctor } x) =$

$\mathsf{topFree}\ x \cup \bigcup_{j\in[n]} (\bigcup_{t\in \mathsf{rec}_j\ x} \mathsf{FVars}_i\ t) \setminus \mathsf{topBind}_{i,j}\ x$ which is weakened by (VC) and (VD) into inclusions of opposite polarities. An indeed, comodels achieve the dual of what models achieve:

Theorem 23. Given a comodel $\mathcal{U}$, 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) $\mathsf{map}_F\ [\mathsf{id}]^{2*m}\ [H]^n\ (\mathsf{Udtor}\ d) \subseteq \mathsf{dtor}\ (H\ d)$

(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 as $y \in \mathsf{Udtor}\ d \longrightarrow \mathsf{map}_F\ [\mathsf{id}]^{2*m}\ [H]^n\ y \in \mathsf{dtor}\ (H\ d)$ and further, expanding the definition of dtor from ctor, as

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

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 does not matter.

Corollary 24. $(\overline{\alpha}\ \boldsymbol{T}, \overline{\mathsf{FVars}}, \mathsf{map}_{\boldsymbol{T}}, \mathsf{dtor})$ 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 on the very topic of explicit parameters.) The corecursive definition of substitution is a good illustration of this phenomenon. We define the comodel $\mathcal{U}$ 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 $\mathsf{Udtor}\ (t, \overline{f}) = \{\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [(\lambda t'.\ (t', \overline{f}))]^n\ x \mid x \in \mathsf{dtor}\ t \wedge \mathsf{noClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing\}$. 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 23 offers the function $\mathsf{sub} : \overline{\alpha}\ U \to \overline{\alpha}\ \boldsymbol{T}$ that satisfies three clauses, among which

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

After uncurrying, this is equivalent to $(*)$ by the functoriality of map_F and the fact that $x \in \mathsf{dtor}\ t$ means $t = \mathsf{ctor}\ x$.

8 ISABELLE FORMALIZATION AND IMPLEMENTATION

All our results have been formalized in Isabelle/HOL (see the supplementary material), in a slightly less general case than presented in this paper. Namely, while the formalization is abstract in that it works with arbitrary type constructors and constants (such as F and map_F), it is concrete in that it fixes certain 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 an F of a fixed arity, $(\beta_1, \alpha_1, \tau_1, \tau_2)\ F$ taking $m = n = 1$ and $\theta = \{(1, 2)\}$.

This is the best we can do while working in the Isabelle/HOL user space, given that in the HOL logic 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 away the category theory under some user-friendly notation, e.g., splitting, for sum types, the single abstract constructor into multiple user-named constructors.

As a first step in the usability direction, we have implemented in Standard ML a tool that automates the process of instantiating an Isabelle formalization parameterized by some type constructors and polymorphic constants with any desired user specified instances—e.g., to switch from an arbitrary functor F to the particular one required by the syntax of the λ-calculus. This tool has already been very helpful with instantiating the arbitrary (co)models used by our (co)recursion principles into the specific ones needed to define substitution.

The distance between our current work and a fully usable definitional package is a good opportunity to reflect on the benefits and limitations of our bindings as functors approach. The main benefit is the semantic treatment of binders, which allows us to "plug and play" arbitrarily complex binders into (co)datatypes—an improvement over the syntactic approaches which essentially inline the whole complexity of binders into the (co)datatypes. On the other hand, we should stress that the functorial approach is no substitute for the usual combinatorial complexity stemming from many-sortedness: multiple types of variables bound in multiple types of terms. To cope with these, we are forced to consider multiple arguments for our functors (and also mutually recursive (co)datatypes). In conclusion, unlike the syntactic approaches we can hide the binding complexity, but like the syntactic approaches we must face the many-sortedness complexity.

9 RELATED WORK

The literature includes some major frameworks and paradigms for syntax with bindings, featuring a variety of mechanisms for specification and reasoning.

Major Frameworks for Bindings. There is considerable overlap between our framework and nominal logic [Pitts 2003], which is itself a syntax-free axiomatization of term-like entities that can contain variables, called atoms. We can draw 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 having $\overline{\alpha}$ as 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 function $\mathsf{map}_{\boldsymbol{T}}$ 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 [Pitts 2006].

Nevertheless, there are some 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 larger entities, all we need to do is instantiate type variables with suitably large types. A second 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). Our functors allow us to apply not only swappings or permutations but arbitrary substitutions.

A well-established alternative to nominal logic is higher-order abstract syntax (HOAS) [Harper et al. 1987; Pfenning and Elliott 1988], which reduces the bindings of the object syntax to those of

the metalogic. HOAS can often simplify reasoning [Felty et al. 2015; Pientka 2018], but its reliance on the metalogic's binder makes it difficult (if possible at all) to encode complex binding patterns.

Scope graphs [van Antwerpen et al. 2016] are a recent language-independent framework for specifying bindings. This research is not concerned with definitional or reasoning principles, but with the integration of bindings with programming language parsers, compilers and static analyzers.

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], and Isabelle Nominal2 [Urban and Kaliszyk 2012]. By contrast, we axiomatize binders through their fundamental properties and show that any binder satisfying the axiomatization can participate in binding-aware (co)datatypes. For example, the Isabelle Nominal2 package [Urban and Kaliszyk 2012] is an impressive piece of engineering that caters for complex binders specified as recursive datatypes, but suffers from the lack of flexibility specific to syntactic formats. It cannot be combined with datatypes specified outside the framework, in particular, its nominal datatypes cannot nest standard (co)datatypes. Our approach owes its generality and modularity to the use of category theory. To our knowledge, our approach is the first in which category theory is used not only for the construction of (co)datatypes but also for capturing complex binders; other category-theoretic frameworks [Fiore et al. 1999; Hofmann 1999; Kurz et al. 2013] confine themselves to trivial binders.

Reasoning and Definitional Infrastructure. Besides specification expressiveness, another criterion for assessing a formal framework is the amount of infrastructure built around the specification language, including reasoning and definitional mechanisms. For syntactic approaches, 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 approach.

Binding-aware recursion is technically more complex than induction, given the requirement that one produces 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 [Urban and Berghofer 2006] and in Coq [Aydemir et al. 2007]), and the essential variations due to Norrish [2004] (implemented in HOL4) and Gheri and Popescu [2017] (implemented in Isabelle). Our recursor improves on the expressiveness of all these recursors, by combining their respective strengths: It uses a flexible notion of (dynamic) parameter as in Norrish [2004] with an improved Horn-style axiomatization as in Gheri and Popescu [2017] and circumvents the nominal recursor's limitation that freshness must be definable from the permutation action. (In Appendix E, we have formally proved these claims for the particular case of the λ-calculus.)

The last paragraph only covers nominal-style recursors, where the recursive clauses refer to the native "first-order" binding operators, such as $\lambda : \alpha \times \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$ for the λ-calculus. This has the important advantage of manipulating terms in a natural way, reflecting the informal practice in describing logics and programming languages. Other recursors in the literature emphasize different constructors, such as $\lambda_{\text{de Bruijn}} : \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$ and $\lambda_{\text{weak HOAS}} : (\alpha \to \alpha\ \boldsymbol{T}) \to \alpha\ \boldsymbol{T}$. These circumvent the difficulties arising from quotienting, at the cost of having to filter out unwanted terms and introduce an encoding overhead. Several elaborate frameworks have been developed within these approaches (some not limited to recursors), such as presheaf-based abstract syntax [Ambler et al.

2003; Fiore et al. 1999; Hofmann 1999], bindings embedded in nested datatypes [Bird and Paterson 1999], bindings embedded in dependent types [Allais et al. 2017] (formalized in Agda), the locally nameless representation [Aydemir et al. 2008; Charguéraud 2012], Autosubst [Schäfer et al. 2015], weak HOAS [Despeyroux et al. 1995] and parametric HOAS [Chlipala 2008] (the last four formalized in Coq). In the end, these different approaches target the same platonic concept of term, and our binding specification framework could in principle offer these alternative "views" of the term datatype by defining the alternative constructors and proving their associated recursors.

For non-well-founded syntax with bindings such as infinite-depth λ-calculus terms (also known as Böhm trees [Barendregt 1984]), coinduction seems to be largely unexplored territory—where we study and criticize parameter-based fresh coinduction, and produce an improved version based on dynamically changing binding variables in terms. By contrast, corecursion has been studied for Böhm trees [Kurz et al. 2012] and more generally for nominal codatatypes [Kurz et al. 2013]. However, these works impose the finite-support restriction even for infinite objects. With the help of regular cardinals, we are able to lift this artificial restriction. A different approach to go beyond finite support has been taken by Gabbay [2007], as an infinitary extension of his previous work on nonstandard set-theoretic foundations of nominal logic.

Acknowledgment. Blanchette has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Popescu has received funding from UK's Engineering and Physical Sciences Research Council (EPSRC) via the grant EP/N019547/1, Verification of Web-based Systems (VOWS). The authors are listed alphabetically.

REFERENCES

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-scope safe programs and their proofs. In *CPP*. 195–207.

S. J. Ambler, Roy L. Crole, and Alberto Momigliano. 2003. A definitional approach to primitivexs recursion over higher order abstract syntax. In *MERLIN*.

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 *TPHOLs*. 50–65.

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.

Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. In *POPL*. 3–15.

H. P. Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier.

Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.* 9, 1 (1999), 77–91.

Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. 2014. Truly Modular (Co)datatypes for Isabelle/HOL. In *ITP*. 93–110.

Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reasoning* 49, 3 (2012), 363–408.

Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP*. 143–156.

Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *J. Symb. Logic* 5, 2 (1940), 56–68.

Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. 1995. Higher-Order Abstract Syntax in Coq. In *TLCA*. 124–138.

Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. 2015. 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.

Marcelo Fiore, Gordon Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding (Extended Abstract). In *LICS*. 193–202.

Denis Firsov and Aaron Stump. 2018. Generic derivation of induction for impredicative encodings in Cedille. In *CPP*. ACM, 215–227.

Murdoch Gabbay. 2007. A general mathematics of names. *Inf. Comput.* 205, 7 (2007), 982–1011.

Lorenzo Gheri and Andrei Popescu. 2017. A Formalized General Theory of Syntax with Bindings. In *ITP*. 241–261.

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

Robert Harper, Furio Honsell, and Gordon Plotkin. 1987. A Framework for Defining Logics. In *LICS*. 194–204.

Martin Hofmann. 1999. Semantical Analysis of Higher-Order Abstract Syntax. In *LICS*. 204–213.

Jonas Kaiser, Brigitte Pientka, and Gert Smolka. 2017. Relating System F and Lambda2: A Case Study in Coq, Abella and Beluga. In *FSCD*. 21:1–21:19.

Alexander Kurz, Daniela Petrisan, Paula Severi, and Fer-Jan de Vries. 2012. An Alpha-Corecursion Principle for the Infinitary Lambda Calculus. In *CMCS*. 130–149.

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).

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375.

Robin Milner. 2001. *Communicating and mobile systems: the π-calculus*. Cambridge.

Michael Norrish. 2004. Recursive Function Definition for Types with Binders. In *TPHOLs*. 241–256.

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *PLDI*, Richard L. Wexelblat (Ed.). 199–208.

Brigitte Pientka. 2018. POPLMark reloaded: mechanizing logical relations proofs (invited talk). In *CPP*. 1.

Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Inf. Comput.* 186, 2 (2003), 165–193.

Andrew M. Pitts. 2006. Alpha-structural recursion and induction. *J. ACM* 53, 3 (2006).

François Pottier. 2006. An Overview of CαMl. *Electron. Notes Theor. Comput. Sci.* 148, 2 (2006), 27–52.

J. J. M. M. Rutten. 1998. Relators and Metric Bisimulations. *Electr. Notes Theor. Comput. Sci.* 11 (1998), 252–258.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *ITP*. 359–374.

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. 2010. Ott: Effective tool support for the working semanticist. *J. Funct. Program.* 20, 1 (2010), 71–122.

Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. 2012. Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In *LICS*. 596–605.

Christian Urban and Stefan Berghofer. 2006. A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL. In *IJCAR*. 498–512.

Christian Urban, Stefan Berghofer, and Michael Norrish. 2007. Barendregt's Variable Convention in Rule Inductions. In *CADE*. 35–50.

Christian Urban and Cezary Kaliszyk. 2012. General Bindings and Alpha-Equivalence in Nominal Isabelle. *Logical Methods in Computer Science* 8, 2 (2012).

Christian Urban and Christine Tasson. 2005. Nominal Techniques in Isabelle/HOL. In *CADE*. 38–53.

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 *PEPM*. 49–60.

Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. 2011. Binders unbound. In *ICFP*. 333–345.

APPENDIX

This appendix is included as part of the supplementary material.

A MORE DETAILS ABOUT BNFS AND BNF-BASED (CO)DATATYPES

The properties of BNFs and their employment in the construction of modular (co)datatypes are described in Traytel et al. [2012] and Blanchette et al. [2014]. Here we recall the reasoning principles emerging from these constructions, referring to the notations in Section 2.3.

The minimality of the datatype construction is expressed in the following *structural induction* proof principle:

(SI) Given the predicate $\varphi : \overline{\alpha}\ T \to \mathit{bool}$, if the condition

$$\forall x : (\overline{\alpha}, \overline{\alpha}\ T)\, F.\ (\forall t \in \mathrm{set}_F^{m+1} x.\ \varphi\ t) \longrightarrow \varphi\ (\mathrm{ctor}\ x)$$

holds, then the following holds: $\forall t : \alpha\ T.\ \varphi\ t$.

For codatatypes, we no longer have a structural induction proof principle, but a *structural coinduction* principle:

(SC) Given the binary relation $\varphi : \overline{\alpha}\ T \to \overline{\alpha}\ T \to \mathit{bool}$, if the condition

$$\forall x, y : (\overline{\alpha},\ \overline{\alpha}\ T)\, F.\ \varphi\ (\mathrm{ctor}\ x)\ (\mathrm{ctor}\ y) \longrightarrow \mathrm{rel}_F\ [(=)]^m\ \varphi\ x\ y$$

holds, then the following holds: $\forall s, t : \alpha\ T.\ \varphi\ s\ t \longrightarrow s = t$.

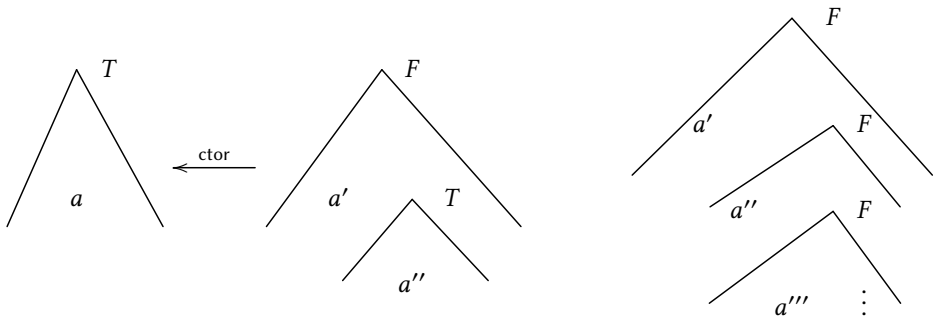
Visual intuition. Datatype and codatatype (and the difference between them) can be viewed as "shape plus content." To simplify notations, consider the binary BNF $(\alpha, \tau)\, F$ and its datatype on the second component, $\alpha\ T$, so that we have $\alpha\ T \simeq (\alpha, \alpha\ T)\, F$ via the isomorphism

$$(\alpha, \alpha\ T)\, F \xrightarrow{\mathrm{ctor}} \alpha\ T$$

(Thus, we have $m = 1$.) Fig. 3a illustrates the effect of decomposing, or "pattern-matching" a member of the datatype, $t : \alpha\ T$. Such an element will have the form ctor x, where $x : (\alpha, \alpha\ T)\, F$. In turn, x has two types of atoms: the items in $\mathrm{set}_1^F\ x$, which are members of α, and the items in $\mathrm{set}_2^F\ x$, which themselves members of the datatype—we call the latter the *recursive components of* t. By repeated applications of ctor, set_1^F, and set_2^F, any element of the datatype can be unfolded into an F-branching tree, which has two types of nodes: ones that represent members of α, and ones that represent elements of the datatype. The former are always leaves, whereas the latter are leaves if and only if they have no recursive components themselves, i.e., applying set_2^F to them yields $\varnothing$. Fig. 3b pictures a recursive component path of such a tree.

The essential property of the datatype is that all such trees are well founded, meaning that they all end in items t that have no recursive components ($\mathrm{set}_2^F\ t = \varnothing$). This is precisely what the structural induction principle (SI) says, in a slightly different, higher-order formulation that is more suitable for proof development: A predicate φ ends up being true for the whole datatype if, for each element ctor x, φ is true for ctor x provided φ is true for all its recursive components $t \in \mathrm{set}_F^2\ (\mathrm{ctor}\ x)$.

If instead of a datatype we consider the codatatype $\alpha\ T$ defined as $\alpha\ T \simeq^{\infty} (\alpha, \alpha\ T)\, F$, the pictures in Fig. 3 remain relevant. The difference is that members of $\alpha\ T$ can now be unfolded into possibly *non-well-founded* trees—i.e., trees that are allowed to have infinite recursive-component paths, corresponding to an infinite number of applications of the constructor. As a result, induction is no longer a valid proof principle. However, we can take advantage of the fact that the tree obtained by fully unfolding a member t of the codatatype determines t uniquely—along the principle "to be



(a) Applying the (co)datatype constructor (b) Elements of the datatypes as piles of F-shapes

Fig. 3. Visualizing a datatype

is to do," where "to be" refers to t's identity and "to do" refers to t's unfolding behavior.[1] Thus, s and t will be equal whenever they are bisimilar as F-trees—that is, if there exists an F-bisimilarity relation $\varphi : \alpha\ T \to \alpha\ T \to bool$ such that $\varphi\ s\ t$ holds. The notion of φ being an F-bisimilarity means that, whenever φ relates two F-trees ctor x and ctor y, their top F-layers have the same shapes, positionwise equal α-atoms, and positionwise φ-related components. As seen in Section 2.2, such positionwise relations can be expressed using the relator of F. The structural coinduction principle (SC) embodies the above reasoning pattern.

Modularity. The type constructors T resulting from (co)datatype definitions are themselves BNFs, hence can be used in later (co)datatype definitions. This allows one to freely mix and nest (co)datatypes in a modular fashion.

The above definitional modularity is matched by a proof-principle modularity: The (co)induction principle associated with a (co)datatype respects the abstraction barrier of the (co)datatypes nested in it, in that it does not refer to their definition or their constructors; instead, it only uses their BNF interfaces, consisting of map functions, set functions and relators. For example, here is the structural coinduction principle for finitely branching possibly non-well-founded rose trees, defined as a codatatype by $\alpha\ tree_\infty \simeq \alpha \times (\alpha\ tree_\infty)\ list$, where for its constructor we write Node instead of ctor:

(SCP$_{tree_\infty}$) Given $\varphi : \alpha\ tree_\infty \to \alpha\ tree_\infty \to bool$, if

$$\forall ts, ss : (\alpha\ tree_\infty)\ list.\ \varphi\ (\mathsf{Node}\ a\ ts)\ (\mathsf{Node}\ b\ ss) \longrightarrow a = b \wedge rel_{list}\ \varphi\ ts\ ss$$

then $\forall s, t : \alpha\ tree.\ \varphi\ s\ t \longrightarrow s = t$

Thus, the codatatype $tree_\infty$ nests the datatype $list$, but its coinduction principle only refers to $list$'s relator structure, rel_{list}. In proofs, one is free to also use the particular definition of rel_{list}, which is the componentwise lifting of a relation to lists—but the coinduction principle for $tree_\infty$ does not depend on such details. The $list$ type constructor is seen as an arbitrary BNF. To define unordered rose trees, we could use the finite powerset BNF $fset$ instead of $list$, and the coinductive principle would remain the same, except with rel_{fset} instead of rel_{list}.

[1] This formulation is Jan Rutten's import of the famous existentialist dogma into the realm of fully abstract coalgebras.

B FULL DEFINITION OF MRBNFS

Sections 3, 4, and 5 develop the notion of MRBNF through a sequence of refinements. While it can be inferred from the refinements, it may also be useful to list the end product as a single definition.

Let $F = (F, \mathsf{map}_F, (\mathsf{set}_F^i)_{i\in[m_1+m_2+n]}, \mathsf{bd}_F)$, where

- F is an $m_1 + m_2 + n$-ary type constructor;
- bd_F is an infinite cardinal number;
- $\mathsf{bd}_F \le |\beta_i|$ and $|\beta_i|$ is a regular cardinal for all $i \in [m_1]$;
- $\mathsf{bd}_F \le |\alpha_i|$ and $|\alpha_i|$ is a regular cardinal for all $i \in [m_2]$;
- $\mathsf{map}_F : (\beta_1 \to \beta_1) \to \cdots \to (\beta_{m_1} \to \beta_{m_1}) \to (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_{m_2} \to \alpha_{m_2}) \to (\tau_1 \to \tau_1') \to \cdots \to (\tau_n \to \tau_n') \to (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to (\overline{\beta}, \overline{\alpha}, \overline{\tau'})\, F$;
- $\mathsf{set}_F^i : (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to \beta_i$ *set* for $i \in [m_1]$;
- $\mathsf{set}_F^{m_1+i} : (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to \alpha_i$ *set* for $i \in [m_2]$;
- $\mathsf{set}_F^{m_1+m_2+i} : (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to \tau_i$ *set* for $i \in [n]$;

F's action on relations $\mathsf{rel}_F : (\beta_1 \to \beta_1) \to \cdots \to (\beta_{m_1} \to \beta_{m_1}) \to (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_{m_2} \to \alpha_{m_2}) \to (\tau_1 \to \tau_1' \to \mathit{bool}) \to \cdots \to (\tau_n \to \tau_n' \to \mathit{bool}) \to (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to \mathit{bool}$ is defined by

(DefRel) $(\forall i \in [m_1].\ |\mathsf{supp}\ u_i| < |\beta_i|) \wedge (\forall i \in [m_2].\ |\mathsf{supp}\ v_i| < |\alpha_i| \wedge \mathsf{bij}\ v_i) \longrightarrow$
$\mathsf{rel}_F\ \overline{u}\ \overline{v}\ \overline{R}\ x\ y \longleftrightarrow \exists z.\ (\forall i \in [n].\ \mathsf{set}_F^{m_1+m_2+i}\ z \subseteq \{(a, a') \mid R_i\ a\ a'\})$
$\wedge\ \mathsf{map}_F\ [\mathsf{id}]^{m_1}\ [\mathsf{id}]^{m_2}\ [\mathsf{fst}]^n\ z = x \wedge \mathsf{map}_F\ \overline{u}\ \overline{v}\ [\mathsf{snd}]^n\ z = y$

(where bij is a predicate expressing that a function is a bijection). F is an *$\overline{\beta}$-free $\overline{\alpha}$-binding map-restricted bounded natural functor* (*MRBNF*) if it satisfies the following properties:

(Fun) (F, map_F) is an n-ary functor—i.e., map_F commutes with function composition and preserves the identities, i.e.,

$$\begin{aligned}
&\mathsf{map}_F\ [\mathsf{id}]^{m_1}\ [\mathsf{id}]^{m_2}\ [\mathsf{id}]^n = \mathsf{id}\\
&(\forall i \in [m_1].\ |\mathsf{supp}\ u_i| < |\beta_i| \wedge |\mathsf{supp}\ u_i'| < |\beta_i|) \wedge\\
&(\forall i \in [m_2].\ |\mathsf{supp}\ v_i| < |\alpha_i| \wedge \mathsf{bij}\ v_i \wedge |\mathsf{supp}\ v_i'| < |\alpha_i| \wedge \mathsf{bij}\ v_i') \longrightarrow\\
&\quad \mathsf{map}_F\ (u_1 \circ u_1') \cdots (u_{m_1} \circ u_{m_1}')\ (v_1 \circ v_1') \cdots (v_{m_2} \circ v_{m_2}')\ (g_1 \circ f_1) \cdots (g_n \circ f_n) =\\
&\quad \mathsf{map}_F\ \overline{u}\ \overline{v}\ \overline{g} \circ \mathsf{map}_F\ \overline{u'}\ \overline{v'}\ \overline{f};
\end{aligned}$$

(Nat) each set_F^i is a natural transformation between the functor (F, map_F) and the powerset functor $(\mathit{set}, \mathsf{image})$, i.e.,

$$\begin{aligned}
&(\forall i \in [m_1].\ |\mathsf{supp}\ u_i| < |\beta_i|) \wedge (\forall i \in [m_2].\ |\mathsf{supp}\ v_i| < |\alpha_i| \wedge \mathsf{bij}\ v_i) \longrightarrow\\
&\quad (\forall i \in [m_1].\ \mathsf{set}_F^i \circ \mathsf{map}_F\ \overline{u}\ \overline{v}\ \overline{f} = \mathsf{image}\ u_i \circ \mathsf{set}_F^i) \wedge\\
&\quad (\forall i \in [m_2].\ \mathsf{set}_F^{m_1+i} \circ \mathsf{map}_F\ \overline{u}\ \overline{v}\ \overline{f} = \mathsf{image}\ v_i \circ \mathsf{set}_F^{m_1+i}) \wedge\\
&\quad (\forall i \in [n].\ \mathsf{set}_F^{m_1+m_2+i} \circ \mathsf{map}_F\ \overline{u}\ \overline{v}\ \overline{f} = \mathsf{image}\ f_i \circ \mathsf{set}_F^{m_1+m_2+i};)
\end{aligned}$$

(Cong) map_F only depends on the value of its argument functions on the elements of set_F^i, i.e.,

$$\begin{aligned}
&(\forall i \in [m_1].\ |\mathsf{supp}\ u_i| < |\beta_i| \wedge |\mathsf{supp}\ u_i'| < |\beta_i|) \wedge\\
&(\forall i \in [m_2].\ |\mathsf{supp}\ v_i| < |\alpha_i| \wedge \mathsf{bij}\ v_i \wedge |\mathsf{supp}\ v_i'| < |\alpha_i| \wedge \mathsf{bij}\ v_i') \wedge\\
&(\forall i \in [m_1].\ \forall a \in \mathsf{set}_F^i\ x.\ u_i\ a = u_i'\ a) \wedge\\
&(\forall i \in [m_2].\ \forall a \in \mathsf{set}_F^{m_1+i}\ x.\ v_i\ a = v_i'\ a) \wedge\\
&(\forall i \in [n].\ \forall a \in \mathsf{set}_F^{m_1+m_2+i}\ x.\ f_i\ a = g_i\ a) \longrightarrow\\
&\quad \mathsf{map}_F\ \overline{u}\ \overline{v}\ \overline{f}\ x = \mathsf{map}_F\ \overline{u'}\ \overline{v'}\ \overline{g}\ x;
\end{aligned}$$

(Bound) the elements of set_F^i are bounded by bd_F, i.e.,

$$\forall i \in [m_1 + m_2 + n].\ \forall x : (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F.\ \ \left|\mathsf{set}_F^i\ x\right| < \mathsf{bd}_F;$$

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

$$\begin{array}{l}\mathsf{rel}_F\ [\mathsf{id}]^{m_1}\ [\mathsf{id}]^{m_2}\ [(=)]^n = (=)\\ (\forall i \in [m_1].\ |\mathsf{supp}\ u_i| < |\beta_i| \wedge |\mathsf{supp}\ u_i'| < |\beta_i|) \wedge\\ (\forall i \in [m_2].\ |\mathsf{supp}\ v_i| < |\alpha_i| \wedge \mathsf{bij}\ v_i \wedge |\mathsf{supp}\ v_i'| < |\alpha_i| \wedge \mathsf{bij}\ v_i') \longrightarrow\\ \quad \mathsf{rel}_F\ (u_1 \circ u_1') \cdots (u_{m_1} \circ u_{m_1}')\ (v_1 \circ v_1') \cdots (v_{m_2} \circ v_{m_2}')\ (R_1 \circledcirc S_1) \cdots (R_n \circledcirc S_n) =\\ \quad \mathsf{rel}_F\ \overline{u}\ \overline{v}\ \overline{R} \circledcirc \mathsf{rel}_F\ \overline{u'}\ \overline{v'}\ \overline{S}.\end{array}$$

Clause (Rel) shows that, on restricted inputs, the MRBNF relator operates not on relations that form the graph of endofunctions or endobijections, but, to the same effect, directly on the functions themselves. In other words, on restricted inputs the relator collapses into the map function.

C MORE DETAILS ON THE (CO)RECURSIVE DEFINITION OF SUBSTITUTION

We show the main proof obligations that must be discharged when defining the variable-for-variable substitution on $\overline{\alpha}\ \boldsymbol{T}$, i.e., the conditions from Definitions 19 and 22 instantiated with the corresponding substitution-specific definitions introduced in Sections 7.1 and 7.2. We omit the easy-to-prove obligations that the instantiations yield term-like structures or parameter structures.

For well-founded terms, we obtain the following two conditions:

(MC) $(\forall i \in [m].\ \mathsf{bij}\ g_i \wedge |\mathsf{supp}\ g_i| < |\alpha_i| \wedge |\mathsf{supp}\ f_i| < |\alpha_i|)$
$\longrightarrow \mathsf{map}_{\boldsymbol{T}}\ \overline{g}\ (\mathsf{ctor}\ (\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\lambda pu.\ pu\ \overline{f}]^n\ y)) =$
$\mathsf{ctor}\ (\mathsf{map}_F\ (\overline{g} \circ \overline{f} \circ \overline{g}^{-1})\ [\mathsf{id}]^m\ [\lambda pu.\ pu\ (\overline{g} \circ \overline{f} \circ \overline{g}^{-1})]^n\ (\mathsf{map}_F\ \overline{g}\ \overline{g}\ [\mathsf{map}_{\boldsymbol{T}}\ \overline{g}]^n\ y))$

(VC) $(\forall i \in [m].\ |\mathsf{supp}\ f_i| < |\alpha_i|) \wedge (\forall i \in [m].\ \mathsf{topBind}_i\ y \cap \mathsf{supp}\ f_i = \varnothing) \wedge$
$(\forall i \in [m].\ \forall j \in [n].\ \forall pu \in \mathsf{rec}_j\ y.\ \forall f.\ \mathsf{FVars}_i\ (pu\ f) \setminus \mathsf{topBind}_{i,j}\ y \subseteq \mathsf{supp}\ f_i)$
$\longrightarrow\ \forall i \in [m].\ \mathsf{FVars}_i\ (\mathsf{ctor}\ (\mathsf{map}_F\ f\ [\mathsf{id}]^m\ [\lambda pu.\ pu\ f]^n\ y)) \subseteq \mathsf{topFree}\ y \cup \mathsf{supp}\ f_i$

The conditions follow by routine reasoning from (restricted) functoriality, naturality, and simple properties of bijections. For non-well-founded terms, we obtain the following four conditions, which are similarly easy to discharge:

(Dne) $\{\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [(\lambda t'.\ (t', \overline{f}))]^n\ x \mid x \in \mathsf{dtor}\ t \wedge (\forall i \in [m].\ \mathsf{noClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing)\} \neq \varnothing$

(DRen) $y, y' \in \{\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [(\lambda t'.\ (t', \overline{f}))]^n\ x \mid x \in \mathsf{dtor}\ t \wedge (\forall i \in [m].\ \mathsf{noClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing)\} \longrightarrow \exists \overline{g}.\ (\forall i \in [m].\ \mathsf{bij}\ g_i \wedge |\mathsf{supp}\ g_i| < |\alpha_i| \wedge$
$(\forall a \in (\bigcup_{j \in [n]} (\bigcup_{u \in \mathsf{rec}_j\ y} \mathsf{FVars}_i\ u \cup \mathsf{supp}\ f_i) \setminus \mathsf{topBind}_{i,j}\ y).\ g_i\ a = a) \wedge$
$y' = \mathsf{map}_F\ [\mathsf{id}]^m\ \overline{g}\ [\lambda(t, \overline{f}).\ (\mathsf{map}_{\boldsymbol{T}}\ \overline{g}\ t, \overline{g} \circ \overline{f} \circ \overline{g}^{-1}))]^n\ y$

(MD) $\{\mathsf{map}_F\ (\overline{g} \circ \overline{f} \circ \overline{g}^{-1}))\ [\mathsf{id}]^m\ [(\lambda t'.\ (t', (\overline{g} \circ \overline{f} \circ \overline{g}^{-1})))]^n\ x \mid x \in \mathsf{dtor}\ (\mathsf{map}_{\boldsymbol{T}}\ \overline{g}\ t) \wedge (\forall i \in [m].\ \mathsf{noClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ (g_i \circ f_i \circ g_i^{-1}) = \varnothing)\} \subseteq$
$\mathsf{image}\ (\mathsf{map}_F\ \overline{g}\ \overline{g}\ [\lambda(t, \overline{f}).\ (\mathsf{map}_{\boldsymbol{T}}\ \overline{g}\ t, \overline{g} \circ \overline{f} \circ \overline{g}^{-1}))]^n)\ (\{\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [(\lambda t'.\ (t', \overline{f}))]^n\ x \mid x \in \mathsf{dtor}\ t \wedge (\forall i \in [m].\ \mathsf{noClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing)\})$

(VD) $y \in \{\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [(\lambda t'.\ (t', \overline{f}))]^n\ x \mid x \in \mathsf{dtor}\ t \wedge (\forall i \in [m].\ \mathsf{noClash}_i\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{supp}\ f_i = \varnothing)\} \longrightarrow \forall i \in [m].\ \mathsf{topFree}\ y \cup \bigcup_{j \in [n]} (\bigcup_{u' \in \mathsf{rec}_j\ y} \mathsf{FVars}_i\ u' \cup \mathsf{supp}\ f_i) \setminus \mathsf{topBind}_{i,j}\ y \subseteq \mathsf{FVars}_i\ t \cup \mathsf{supp}\ f_i$

The (co)recursor-based definitions of tsub for both well-founded and non-well-founded terms are very similar to the one of sub. We refer to the formalization for the precise definitions.

D USEFUL VARIATIONS OF THE (CO)RECURSION PRINCIPLES

D.1 A Fixed-Parameter Restriction

Recall that our recursors employ a notion of dynamically varying parameter, whose free variables must be avoided. Let us introduce some notation for a useful particular case: that of static (fixed) parameters, more precisely, that of fixed sets of variables that must be avoided. Technically, we assume that the parameter type is a singleton, which is the same as replacing the parameter structure with a tuple $\mathcal{A}$ consiting of fixed small sets of variables $A_i \subseteq \alpha_i$ (each A_i representing the set of variables of the unique parameter).[2] Also, since $\overline{\alpha}\ P$ is a singleton, we can replace $\overline{\alpha}\ P \to \overline{\alpha}\ U$ with $\overline{\alpha}\ U$.

Definition 25. Given a tuple $\mathcal{A}$ of small sets, an $\mathcal{A}$-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
- $\mathsf{Umap} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ U \to \overline{\alpha}\ U$

such that the following hold:

(MC) $(\forall i \in [m].\ \mathsf{supp}\ f_i \cap A_i = \varnothing)$
$\longrightarrow \mathsf{Umap}\ \overline{f}\ (\mathsf{Uctor}\ y) = \mathsf{Uctor}\ (\mathsf{map}_F\ \overline{f}\ \overline{f}\ [\mathsf{Umap}\ \overline{f}]^n\ y)$

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

Then Theorem 20 instantiates to:

Theorem 26. Given a tuple of small sets $\mathcal{A}$ and an $\mathcal{A}$-model $\mathcal{U}$, there exists a unique function $H : \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ U$ such that:

(C) $(\forall i \in [m].\ \mathsf{noClash}\ x \wedge \mathsf{topBind}_i\ x \cap A_i = \varnothing)$
$\longrightarrow H\ (\mathsf{ctor}\ x) = \mathsf{Uctor}\ (\mathsf{map}_F\ [\mathsf{id}]^{2*m}\ [H]^n\ x)$

(M) $(\forall i \in [m].\ \mathsf{supp}_i\ f_i \cap A_i = \varnothing) \longrightarrow H\,(\mathsf{map}_{\boldsymbol{T}}\ \overline{f}\ t) = \mathsf{Umap}\ \overline{f}\ (H\ t)$

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

Since the majority of binding-aware recursive definitions seem to require fixed rather than dynamic parameters, in our Isabelle formalization of the recursor we wire in $\mathcal{A}$ as a primitive (in addition to $\mathcal{P}$)—this avoids the bureaucracy of having to instantiate $\mathcal{P}$ to a singleton for handling fixed parameters.

D.2 The Full-Fledged Primitive (Co)recursor

In Section 7 we have presented a restricted form of (co)recursors that are usually known as (co)iterators. Here we formulate the full-fledged (co)recursors, which constitute a theoretically straightforward but practically useful extension of the (co)iterators.

The difference between a recursor and an iterator is that the former allows the value of a function H applied to a given term ctor x to depend not only on the values of H on the recursive components t of x, but also on the components themselves. To cater for this, we routinely enhance our notions of term-like structure and model with additional term arguments, as highlighted below:

Definition 27. An *extended 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 $\mathsf{DFVars}_i : \overline{\alpha}\ P \to \overline{\alpha}\ \boldsymbol{T} \to \alpha_i\ \mathit{set}$ for $i \in [m]$

[2] The smallness of a $A_i \subseteq \alpha_i$ means, as usual, that $|A_i| < |\alpha_i|$.

- $\mathsf{Dmap} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\, D \to \overline{\alpha}\, \boldsymbol{T} \to \overline{\alpha}\, D$

are such that the following hold:

- $\mathsf{Dmap}\ [\mathsf{id}]^m\ t = \mathsf{id}$
- $\mathsf{Dmap}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m)\ t = \mathsf{Dmap}\ \overline{g}\ t \circ \mathsf{Dmap}\ \overline{f}\ t$
- $(\forall i \in [m].\ \forall a \in \mathsf{DFVars}_i\ t\ \ d.\ f_i\, a = a) \longrightarrow \mathsf{Dmap}\ \overline{f}\ t\ d = d$
- $a \in \mathsf{DFVars}_i\ (\mathsf{map}_{\boldsymbol{T}}\ \overline{f}\ t)\ (\mathsf{Dmap}\ \overline{f}\ t\ d) \longleftrightarrow f_i^{-1}\, a \in \mathsf{DFVars}_i\ t\ d$

Definition 28. Given a parameter structure $\mathcal{P}$, an extended $\mathcal{P}$-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 an extended term-like structure
- $\mathsf{Uctor} : (\overline{\alpha}, \overline{\alpha}, [\ \overline{\alpha}\, \boldsymbol{T} \times (\overline{\alpha}\, P \to \overline{\alpha}\, U)]^n)\, F \to \overline{\alpha}\, P \to \overline{\alpha}\, U$

such that the following hold:

(MC) $\mathsf{Umap}\ \overline{f}\ (\mathsf{ctor}\ x_y)\ (\mathsf{Uctor}\ y\ p) = \mathsf{Uctor}\ (\mathsf{map}_F\ \overline{f}\ \overline{f}\ [\ \langle \mathsf{map}_{\boldsymbol{T}}, \mathsf{Umap} \rangle\ \overline{f}]^n\ y)\ (\mathsf{Pmap}\ \overline{f}\ p)$

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

Above, x_y and $x_{y'}$ are shorthands for $\mathsf{map}_F\ [\mathsf{id}]^{2*m}\ [\mathsf{fst}]^n\ y$ and $\mathsf{map}_F\ [\mathsf{id}]^{2*m}\ [\mathsf{fst}]^n\ y'$, respectively. Also recall that fst and snd are the standard first and second projection functions on the product type $\times$. Moreover, $\langle \mathsf{map}_{\boldsymbol{T}}, \mathsf{Umap} \rangle \overline{f}$ denotes the function $\lambda(t, pu).\ (\mathsf{map}_{\boldsymbol{T}}\ \overline{f}\ t, \mathsf{Umap}\ \overline{f}\ t\ pu)$.

Note that, for (VC), the additional structure brought by the extended models makes the presence of topFree y redundant. Indeed, it is easy to check that $\mathsf{topFree}\ y = \mathsf{topFree}\ x_y$, meaning that $\mathsf{topFree}\ y \subseteq \mathsf{FVars}_i\ (\mathsf{ctor}\ y)$. In short, $\mathsf{topFree}\ y$ can be removed from the conclusion of (VC), without affecting this property.

The recursion theorem follows suit with this term-argument extension.

Full-fledged recursion extension of Theorem 20: Given a parameter structure $\mathcal{P}$ and a $\mathcal{P}$-model $\mathcal{U}$, there exists a unique function $H : \overline{\alpha}\, \boldsymbol{T} \to \overline{\alpha}\, P \to \overline{\alpha}\, U$ such that:

(C) $(\forall i \in [m].\ \mathsf{noClash}\ x \wedge \mathsf{topBind}_i\ x \cap \mathsf{PFVars}_i\ p = \varnothing) \longrightarrow$
$H\ (\mathsf{ctor}\ x)\ p = \mathsf{Uctor}\ (\mathsf{map}_F\ [\mathsf{id}]^{2*m}\ [\ \langle \mathsf{id}, H \rangle\ ]^n\ x)\ p$

(M) $H\ (\mathsf{map}_{\boldsymbol{T}}\ \overline{f}\ t)\ p = \mathsf{Umap}\ \overline{f}\ t\ (f\ t\ (\mathsf{Pmap}\ \overline{f}^{-1}\ p))$

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

A similar game can be played with the corecursor, where the additional term inputs occur in the result of the function, with the following intuition: In addition to the option of delving into a corecursive call, we now also have the option to stop the corecursion immediately returning an indicated term. For example, the constructor-like operator Udtor of an extended comodel will have the type $\overline{\alpha}\, U \to ((\overline{\alpha}, \overline{\alpha}, [\ \overline{\alpha}\, \boldsymbol{T} + \overline{\alpha}\, U]^n)\, F)\ \mathit{set}$.

It is easy to infer the extended version of the (co)recursion theorems from their original version. However, in our Isabelle formalization we directly prove the extended versions.

D.3 A Constructor-Based Variation

For an extended model, instead of assuming that it forms an extended term-like structure we can assume that it satisfies the following axiom, obtaining what we cal call *weak extended models*:

(CC) $(\forall j \in [n].\ \forall\, t, pu, p.\ (t,\ pu\ ) \in \mathrm{rec}_j\ y \cup \mathrm{rec}_j\ y'.\ \forall i \in [m].\ \mathrm{UFVars}_i\ (pu\ p) \subseteq \mathrm{FVars}_i\ t \cup \mathrm{PFVars}_i\ p) \wedge$
$(\forall i \in [m].\ \mathrm{supp}\ f_i\ \cap\ (\ \mathrm{FVars}_i\ (\mathrm{ctor}\ x_y) \cup\ \mathrm{PFVars}_i\ p) = \varnothing) \wedge$
$(\forall i \in [m].\ f_i\ (\mathrm{topBind}_i\ y)\ \cap\ \mathrm{topBind}_i\ y = \varnothing) \wedge$
$(\forall i \in [m].\ \mathrm{supp}\ f'_i\ \cap\ (\ \mathrm{FVars}_i\ (\mathrm{ctor}\ x_{y'}) \cup\ \mathrm{PFVars}_i\ p) = \varnothing) \wedge$
$(\forall i \in [m].\ f'_i\ (\mathrm{topBind}_i\ y')\ \cap\ \mathrm{topBind}_i\ y' = \varnothing) \wedge$
$\mathrm{map}_F\ \overline{f}\ \overline{f}\ [\ \langle \mathrm{map}_{\boldsymbol{T}}, \mathrm{Umap}\ \rangle\ \overline{f}]^n\ y\ =\ \mathrm{map}_F\ \overline{f'}\ \overline{f'}\ [\ \langle \mathrm{map}_{\boldsymbol{T}}, \mathrm{Umap}\ \rangle\ \overline{f'}]^n\ y$
$\longrightarrow \mathrm{Uctor}\ y = \mathrm{Uctor}\ y'$

(CC) postulates a condition under which the application of the constructor Uctor to two different arguments, y and y', yields the same result. This generalizes (and weakens by adding additional premises) the following property of terms, stating that the constructor produces the same results its the arguments are equal modulo small-support endobijective renamings $\overline{f}$ and $\overline{f'}$:

$(\forall i \in [m].\ \mathrm{supp}\ f_i\ \cap\ \mathrm{FVars}_i\ (\mathrm{ctor}\ x) = \varnothing\ \wedge\ \mathrm{supp}\ f'_i\ \cap\ \mathrm{FVars}_i\ (\mathrm{ctor}\ x') = \varnothing) \wedge$
$\mathrm{map}_F\ \overline{f}\ \overline{f}\ [\mathrm{map}_{\boldsymbol{T}}\ \overline{f}]^n\ x = \mathrm{map}_F\ \overline{f'}\ \overline{f'}\ [\mathrm{map}_{\boldsymbol{T}}\ \overline{f'}]^n\ x'\ \longrightarrow\ \mathrm{ctor}\ x = \mathrm{ctor}\ x'$

This alternative axiomatization seems to be more complex to check in particular cases. However, it is indeed slightly weaker, hence leads to a slightly *stronger* (more expressive) recursor:

Theorem 29. Any weak extended model is also an extended model, hence (the extended version of) Theorem 20 also holds for weak extended $\mathcal{P}$-models.

For this reason, our formalization also includes this constructor-based variation of the recursor (together with the proof of its higher expressiveness power).

E FORMAL COMPARISON WITH RECURSORS FROM THE LITERATURE

To make a comparison with previous work in terms of what we call *intrinsic* recursor expressiveness[3] we need to consider a syntax with bindings that follows in the scope of all these results. We choose the minimalistic syntax of λ-calculus (Example 3).

Now, $\alpha\ \boldsymbol{T}$ denotes the type of λ-terms with variables in α. To avoid confusion with the meta-level, we will write $\mathrm{App} : \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$ and $\mathrm{Lam} : \alpha \to \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$ for the application and λ-abstraction constructors on terms. Note that our abstract constructor $\mathrm{ctor} : (\alpha, \alpha, \alpha\ \boldsymbol{T}, \alpha\ \boldsymbol{T})\ F = \alpha + \alpha\ \boldsymbol{T} \times \alpha\ \boldsymbol{T} + \alpha \times \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$ is the joining of the "concrete" Var, App and Lam constructors. We will prefer to present the λ-calculus instance of our theorem in terms of the concrete constructors. This will also apply to models, where the abstract constructor-like operator Uctor will be correspondingly split into three operators UVar, UApp and ULam. The abstract (single-constructor) and the concrete (multi-constructor) views are the same, modulo the trivial transformations of sum splitting/joining and (un)currying of functions.

Thus, for this particular case, the notion of extended term-like structure becomes a triple $\mathcal{D} = (\alpha\ D, \mathrm{DFVars}, \mathrm{Dmap})$, where[4]

- $\alpha\ D$ is a polymorphic type
- $\mathrm{DFVars} : \alpha\ D \to\ \alpha\ \boldsymbol{T} \to \alpha\ set$

[3] Namely, expressiveness that refers not to the complexity of the binders that it can handle, but to the class of functions that are definable for a given fixed syntax, e.g., that of λ-calculus.

[4] As usual, we highlight the additional structure brought by the full-fledged recursor.

- $\mathsf{Dmap} : (\alpha \to \alpha) \to \alpha\ \boldsymbol{T} \to \alpha\ D \to \alpha\ D$

are such that the following hold:

- $\mathsf{Dmap}\ \mathsf{id}\ t = \mathsf{id}$
- $\mathsf{Dmap}\ (g \circ f)\ t = \mathsf{Dmap}\ g\ t \circ \mathsf{Dmap}\ f\ t$
- $(\forall a \in \mathsf{DFVars}\ t\ d.\ f\ a = a) \longrightarrow \mathsf{Dmap}\ f\ t\ d = d$
- $a \in \mathsf{DFVars}\ (\mathsf{map}_{\boldsymbol{T}}\ f\ t)\ (\mathsf{Dmap}\ f\ t\ d) \longleftrightarrow f^{-1}\ a \in \mathsf{DFVars}\ t\ d$

And the notion of extended $\mathcal{P}$-model becomes a tuple $U = (\alpha\ U, \mathsf{Umap}, \mathsf{UFVars}, \mathsf{UVar}, \mathsf{UApp}, \mathsf{ULam})$ where

- $(\alpha\ U, \mathsf{Umap}, \mathsf{UFVars})$ is a term-like structure
- $\mathsf{UVar} : \alpha \to \alpha\ P \to \alpha\ U$,
  $\mathsf{UApp} : \alpha\ \boldsymbol{T} \to (\alpha\ P \to \alpha\ U) \to \alpha\ \boldsymbol{T} \to (\alpha\ P \to \alpha\ U) \to \alpha\ P \to \alpha\ U$,
  $\mathsf{ULam} : \alpha \to \alpha\ \boldsymbol{T} \to (\alpha\ P \to \alpha\ U) \to \alpha\ P \to \alpha\ U$

satisfying the following properties for all finitely supported endobijections $f : \alpha \to \alpha$:

(MC) • $\mathsf{Umap}\ f\ (\mathsf{Var}\ a)\ (\mathsf{UVar}\ a\ p) = \mathsf{UVar}\ (f\ a)\ (\mathsf{Pmap}\ f\ p)$

- $\mathsf{Umap}\ f\ (\mathsf{UApp}\ t_1\ u_1\ t_2\ u_2\ p) =$
  $\mathsf{UApp}\ (\mathsf{map}_{\boldsymbol{T}}\ f\ t_1)\ (\mathsf{Umap}\ f\ t_1\ u_1)\ (\mathsf{map}_{\boldsymbol{T}}\ f\ t_2)\ (\mathsf{Umap}\ f\ t_2\ u_2)\ (\mathsf{Pmap}\ f\ p)$
- $\mathsf{Umap}\ f\ (\mathsf{ULam}\ a\ t\ u\ p) = \mathsf{ULam}\ (f\ a)\ (\mathsf{map}_{\boldsymbol{T}}\ f\ t)\ (\mathsf{Umap}\ f\ u)\ (\mathsf{Pmap}\ f\ p)$

(VC) • $\mathsf{UFVars}\ (\mathsf{UVar}\ a\ p) \subseteq \mathsf{FVars}\ (\mathsf{Var}\ a) \cup \mathsf{PFVars}\ p$

- $\mathsf{UFVars}\ (pu_1\ p) \subseteq \mathsf{FVars}\ t_1 \cup \mathsf{PFVars}\ p\ \wedge\ \mathsf{UFVars}\ (pu_2\ p) \subseteq \mathsf{FVars}\ t_2 \cup \mathsf{PFVars}\ p$
  $\longrightarrow \mathsf{UFVars}\ (\mathsf{UApp}\ t_1\ pu_1\ t_2\ pu_2\ p) \subseteq \mathsf{FVars}\ (\mathsf{App}\ t_1\ t_2) \cup \mathsf{PFVars}\ p$
- $a \notin \mathsf{PFVars}\ p\ \wedge\ \mathsf{UFVars}\ (pu\ p) \setminus \{a\} \subseteq \mathsf{FVars}\ t \setminus \{a\} \cup \mathsf{PFVars}\ p$
  $\longrightarrow \mathsf{UFVars}\ (\mathsf{ULam}\ t\ pu\ p) \subseteq \mathsf{FVars}\ (\mathsf{Lam}\ a\ t) \cup \mathsf{PFVars}\ p$

And weak extended models are obtained from the above by replacing the extended term-like structure condition with:

(CC) • $\mathsf{UFVars}\ (pu_1\ p) \subseteq \mathsf{FVars}\ t_1 \cup \mathsf{PFVars}\ p\ \wedge\ \mathsf{UFVars}\ (pu_2\ p) \subseteq \mathsf{FVars}\ t_2 \cup \mathsf{PFVars}\ p\ \wedge$
$\mathsf{UFVars}\ (pu_1'\ p) \subseteq \mathsf{FVars}\ t_1' \cup \mathsf{PFVars}\ p\ \wedge\ \mathsf{UFVars}\ (pu_2'\ p) \subseteq \mathsf{FVars}\ t_2' \cup \mathsf{PFVars}\ p$
$\mathsf{supp}\ f\ \cap\ (\mathsf{FVars}\ (\mathsf{App}\ t_1\ t_2) \cup \mathsf{PFVars}\ p) = \varnothing\ \wedge$
$\mathsf{supp}\ f'\ \cap\ (\mathsf{FVars}\ (\mathsf{App}\ t_1'\ t_2') \cup \mathsf{PFVars}\ p) = \varnothing\ \wedge$
$\mathsf{map}_{\boldsymbol{T}}\ f\ t_1 = \mathsf{map}_{\boldsymbol{T}}\ f'\ t_1'\ \wedge\ \mathsf{Umap}\ f\ t_1\ pu_1 = \mathsf{Umap}\ f'\ t_1'\ pu_1'\ \wedge$
$\mathsf{map}_{\boldsymbol{T}}\ f\ t_2 = \mathsf{map}_{\boldsymbol{T}}\ f'\ t_2'\ \wedge\ \mathsf{Umap}\ f\ t_2\ pu_2 = \mathsf{Umap}\ f'\ t_2'\ pu_2'$
$\longrightarrow \mathsf{UApp}\ t_1\ pu_1\ t_2\ pu_2 = \mathsf{UApp}\ t_1'\ pu_1'\ t_2'\ pu_2'$

- $\{a, a'\} \cap \mathsf{PFVars}\ p = \varnothing\ \wedge$
  $\mathsf{UFVars}\ (pu\ p) \subseteq \mathsf{FVars}\ t \cup \mathsf{PFVars}\ p\ \wedge\ \mathsf{UFVars}\ (pu'\ p) \subseteq \mathsf{FVars}\ t' \cup \mathsf{PFVars}\ p\ \wedge$
  $\mathsf{supp}\ f\ \cap\ (\mathsf{FVars}\ (\mathsf{Lam}\ a\ t) \cup \mathsf{PFVars}\ p) = \varnothing\ \wedge\ f\ a \neq a\ \wedge$
  $\mathsf{supp}\ f'\ \cap\ (\mathsf{FVars}\ (\mathsf{Lam}\ a'\ t') \cup \mathsf{PFVars}\ p) = \varnothing\ \wedge\ f'\ a' \neq a'\ \wedge$
  $f\ a = f'\ a'\ \wedge\ \mathsf{map}_{\boldsymbol{T}}\ f\ t = \mathsf{map}_{\boldsymbol{T}}\ f'\ t'\ \wedge\ \mathsf{Umap}\ f\ t\ pu = \mathsf{Umap}\ f'\ t'\ pu'$
  $\longrightarrow \mathsf{ULam}\ a\ t\ pu = \mathsf{ULam}\ a'\ t'\ pu'$

Because of using concrete constructors, each of the abstract clauses splits into three clauses—one for each of concrete constructor. An exception is (CC), where the clause for UVar is trivial due to the nonexistence of recursive components. Some of the abstract premises become simpler in the concrete case. For example, consider the premise $\forall i \in [m].\ f_i\ (\mathsf{topBind}_i\ y)\ \cap\ \mathsf{topBind}_i\ y = \varnothing$ in (CC). First, since $m = 1$, the index i is omitted. Moreover, since App introduces no bindings, the premise becomes vacuous (and omitted) in the case of App/UApp; and becomes $f\ a \neq a$ in the case of Lam/ULam, since this constructor has a single top-binding variable, say, a. As another example, in the Lam/ULam case the (CC) premise $\mathsf{map}_F\ \overline{f}\ \overline{f}\ [\langle \mathsf{map}_{\boldsymbol{T}}, \mathsf{Umap}\rangle\ \overline{f}]^n\ y\ =\ \mathsf{map}_F\ \overline{f'}\ \overline{f'}\ [\langle \mathsf{map}_{\boldsymbol{T}}, \mathsf{Umap}\rangle\ \overline{f'}]^n\ y$ becomes $f\ a = f'\ a'\ \wedge\ \mathsf{map}_{\boldsymbol{T}}\ f\ t = \mathsf{map}_{\boldsymbol{T}}\ f'\ t'\ \wedge\ \mathsf{Umap}\ f\ t\ pu = \mathsf{Umap}\ f'\ t'\ pu'$.

Instantiating (the extended version of) Theorem 20 and Theorem 29 for this syntax gives us the following (again, after performing the splitting according to concrete constructors):

Corollary 30. Given a parameter structure $\mathcal{P}$ and a (weak) extended $\mathcal{P}$-model $\mathcal{U}$, there exists a unique function $H : \alpha\ \boldsymbol{T} \to \alpha\ P \to \alpha\ U$ such that

(C) • $H\ (\mathsf{Var}\ a)\ p = \mathsf{UVar}\ a\ p$
- $H\ (\mathsf{App}\ t_1\ t_2)\ p = \mathsf{UApp}\ \boxed{t_1}\ (H\ t_1)\ \boxed{t_2}\ (H\ t_2)\ p$
- $a \notin \mathsf{PFVars}\ p \longrightarrow H\ (\mathsf{Lam}\ a\ t)\ p = \mathsf{ULam}\ a\ \boxed{t}\ (H\ t)\ p$

(M) $H\ (\mathsf{map}_{\boldsymbol{T}}\ f\ \boxed{t})\ p\ =\ \mathsf{Umap}\ f\ \boxed{t}\ (H\ t\ (\mathsf{Pmap}\ f^{-1}\ p))$

(V) $\mathsf{UFVars}\ \boxed{t}\ (H\ t\ p)\ \subseteq\ \mathsf{FVars}\ t\ \cup\ \mathsf{PFVars}\ p$

Thus, as we would expect, for the λ-calculus our recursor gives us a function H satisfying some recursive clauses w.r.t. the constructors, and also some clauses expressing the preservation of the map function and free-variable operator—all modulo a notion of parameter, with respect to which the λ-binding variables must be fresh in the recursive clause for Lam.

If above we remove the highlighted text, we obtain the more primitive, iterative form of the recursor. It is also worth spelling out the fixed-parameter instance of the iterator (described in general in Section D.1). Fixing $\mathcal{A}$ to consist of a single finite[5] set A, the notion of $\mathcal{A}$-model (for the λ-calculus syntax) becomes a quadruple $\mathcal{U} = (\alpha\ U, \mathsf{UFVars}, \mathsf{Umap}, \mathsf{Uctor})$, where:

- $(\alpha\ U, \mathsf{UFVars}, \mathsf{Umap})$ is a term-like structure
- $\mathsf{Umap} : (\alpha \to \alpha) \to \alpha\ U \to \alpha\ U$

satisfying the following properties for all finitely supported endobijections $f : \alpha \to \alpha$:

(MC) • $\mathsf{supp}\ f\ \cap\ A = \varnothing\ \longrightarrow\ \mathsf{Umap}\ f\ (\mathsf{UVar}\ a) = \mathsf{UVar}\ (f\ a)$
- $\mathsf{supp}\ f\ \cap\ A = \varnothing\ \longrightarrow\ \mathsf{Umap}\ f\ (\mathsf{UApp}\ u_1\ u_2) = \mathsf{UApp}\ (\mathsf{Umap}\ f\ u_1)\ (\mathsf{Umap}\ f\ u_2)$
- $\mathsf{supp}\ f\ \cap\ A = \varnothing\ \longrightarrow\ \mathsf{Umap}\ f\ (\mathsf{ULam}\ a\ u) = \mathsf{ULam}\ (f\ a)\ (\mathsf{Umap}\ f\ u)$

(VC) • $\mathsf{UFVars}\ (\mathsf{UVar}\ a) \subseteq \mathsf{FVars}\ (\mathsf{Var}\ a)\ \cup\ A$
- $\mathsf{UFVars}\ u_1 \subseteq A\ \wedge\ \mathsf{UFVars}\ u_2 \subseteq A\ \longrightarrow\ \mathsf{UFVars}\ (\mathsf{UApp}\ u_1\ u_2) \subseteq A$
- $a \notin A\ \wedge\ \mathsf{UFVars}\ u \subseteq A\ \longrightarrow\ \mathsf{UFVars}\ (\mathsf{ULam}\ a\ u) \subseteq A$

Then Theorem 26 instantiates to:

Corollary 31. Given an $\mathcal{A}$-model $\mathcal{U}$, there exists a unique function $H : \alpha\ \boldsymbol{T} \to \alpha\ U$ such that:

(C) • $H\ (\mathsf{Var}\ a) = \mathsf{UVar}\ a$
- $H\ (\mathsf{App}\ t_1\ t_2) = \mathsf{UApp}\ (H\ t_1)\ (H\ t_2)$
- $a \notin A \longrightarrow H\ (\mathsf{Lam}\ a\ t) = \mathsf{ULam}\ a\ (H\ t)$

(M) $\mathsf{supp}\ f\ \cap\ A = \varnothing\ \longrightarrow H\ (\mathsf{map}_{\boldsymbol{T}}\ f\ t)\ =\ \mathsf{Umap}\ f\ (H\ t)$

(V) $\mathsf{UFVars}\ (H\ t)\ \subseteq\ \mathsf{FVars}\ t\ \cup\ A$

[5] Recall that, in this finitary case, "small" means "finite."

Comparison with the nominal recursor Recall that, for this particular finitary syntax:

- Our type variable α, assumed countable, corresponds to a Nominal set of atoms
- Our functions $f : \alpha \to \alpha$ of small support correspond to permutations of finite support
- Our map function map_T corresponds to the swapping action on terms

Thus, we can formulate (the λ-calculus instance of) the nominal recursor [Pitts 2006] using the terminology of this paper:

Definition 32. The notion *nominal $\mathcal{A}$-model* is obtained from that of $\mathcal{A}$-model by removing the last two (out of four) term-like structure axioms and the (VC) axiom and adding instead the following axioms:

(FfromM) $\mathsf{UFVars}\ u = \{a \mid \{a' \mid \mathsf{Umap}\ (a \leftrightarrow a')\ u \neq u\}\ \text{finite}\}$
(FCB) $\exists u.\ \forall a.\ a \notin A \wedge a \notin \mathsf{UFVars}\ u$

In clause (FfromM), $(a \leftrightarrow a')$ denotes the swapping function in $\alpha \to \alpha$, sending a to a', a' to a and everything else to itself. Note that we use a free-variable-like operator $\mathsf{UFVars} : U \to \alpha\ set$, whereas the nominal logic literature considers a freshness operator $\mathsf{Ufresh} : \alpha\ U \to \alpha \to bool$ (and usually writes $a\#u$ instead of $\mathsf{Ufresh}\ u\ a$). These are of course inter-definable via negation, as follows:

- $\mathsf{Ufresh}\ u\ a$, as $a \notin \mathsf{UFVars}\ u$
- $\mathsf{UFVars}\ u$, as $\{a \mid \neg\ \mathsf{Ufresh}\ u\ a\}$

With this translation, we see that (FfromM) states that freshness is definable from mapping (i.e., from the nominal permutation action)[6] and (FCB) is the so-called *freshness condition for binders*, both familiar from nominal logic:

(FfromM) $\mathsf{Ufresh}\ a\ u \longleftrightarrow \{a' \mid \mathsf{Umap}\ (a \leftrightarrow a')\ u \neq u\}$ infinite
(FCB) $\exists u.\ \forall a.\ a \notin A \wedge a \notin \mathsf{Ufresh}\ a\ u$

We can show that the nominal $\mathcal{A}$-model axioms are stronger than those of the $\mathcal{A}$-models. Indeed, the UFVars operator defined in a nominal $\mathcal{A}$-model from the Umap operator can be shown to satisfy the (VC) axioms.

Proposition 33. Any nominal $\mathcal{A}$-model is an $\mathcal{A}$-model.

As a consequence, we obtain the following:

Corollary 34. We have that Corollary 31 stays true if we replace $\mathcal{A}$-models with nominal $\mathcal{A}$-models.

This last corollary is essentially the sort-directed alpha-structural recursion theorem (Theorem 5.1) of [Pitts 2006]—henceforth abbreviated ASRT—instantiated to the λ-calculus syntax (thus, taking the signature Σ to consist of the λ-calculus constructors Var, Lam, App).

Indeed, concerning the input to the two recursion theorems: Our model carrier $\alpha\ U$ corresponds to the the ASRT target domain X, and our constructor-like operators UVar, ULam, UApp correspond to the ASRT functions f_k considered on the target domain. Our set A corresponds to the ASRT set A. Our first two term-like structure axioms (the only ones kept in the notion of $\mathcal{A}$-nominal model) correspond the ASRT requirement that the permutation action satisfies the first nominal-set axioms concerning the identity and composition of permutation actions. Our axioms (MC) correspond to

[6] See in [Urban and Tasson 2005], Def. 3. Pitts [2006] gives a slightly more complex definition (as the minimal set that supports and entity)—these two definitions are known to be equivalent in the presence of finite support, which is a pervasive assumption in nominal logic.

the fact that the functions f_k are supported by A.[7] Our condition (FCB) corresponds to the identically named ASRT condition—noting that for the λ-calculus syntax this condition is only meaningful for Lam, the only constructor that actually binds variables. Finally, our nominal $\mathcal{A}$-models have Ufresh as part of their structure, whereas the ASRT target domain does not have any freshness operator as a primitive. However, the ASRT target domain is required to be a nominal set, hence it has a notion of freshness definable from the permutation action—which is exactly what our (FfromM) axiom imposes. In short, there is a precise (bijective) correspondence between our nominal $\mathcal{A}$-models and the structure considered on the ASRT target domains.

Now, concerning the output of the two recursion theorems: Our function H corresponds to the ASRT function $\hat{f}$. Our clause (C) corresponds to ASRT's recursive clauses for $\hat{f}$ (labeled as identity (47) in the paper) and (M) corresponds to ASRT concluding that the defined function $\hat{f}$ is itself supported by A. On the other hand, there is nothing in ASRT that matches our clause (V); but in the nominal case, with freshness definable from mapping, this clause is redundant, i.e., follows from the others. In short, via the aforementioned correspondence, the recursor based on nominal $\mathcal{A}$-models (which is a particular case of our more general recursor) produces the same results as ASRT.

Comparison with the Norrish and the Gheri-Popescu recursors These recursors operate with swapping actions $(a \leftrightarrow b) : \alpha \to \alpha$, also known as transpositions, rather than arbitrary permutations. The possibility to restrict the focus in this way is based on the fact that all finite-support permutations are generated from transpositions via composition . (This does not scale, however, to the infinitary case.)

Indeed, in the finitary case our axiomatization of term-like structures using arbitrary small-support (here finite-support) endobijections is equivalent to the following axiomatization using a swapping-like operator:

Definition 35. A swapping-based term-like structure is a triple $\mathcal{D} = (\alpha\ D, \mathsf{DFVars}, \mathsf{Dswap})$, where

- $\alpha\ D$ is a polymorphic type
- $\mathsf{DFVars} : \alpha\ D \to \alpha\ set$
- $\mathsf{Dswap} : \alpha \times \alpha \to \alpha\ D \to \alpha\ D$

are such that the following hold:[8]

- $\mathsf{Dswap}\ a\ a'\ d = d$
- $\mathsf{Dswap}\ b\ b'\ (\mathsf{Dswap}\ a\ a'\ d) = \mathsf{Dswap}\ ((b \leftrightarrow b')\ a)\ ((b \leftrightarrow b')\ a')\ (\mathsf{Dswap}\ b\ b'\ d)$
- $a, a' \notin \mathsf{DFVars}\ d \longrightarrow \mathsf{Dswap}\ a\ a'\ d = d$
- $a \in \mathsf{DFVars}\ (\mathsf{Dswap}\ b\ b'\ d) \longleftrightarrow (b \leftrightarrow b')\ a \in \mathsf{DFVars}\ d$

The above swapping-based axiomatization is equivalent to the mapping-based axiomatization. Indeed, from a term-like structure we obtain a swapping-based term-like structure by simply taking Dswap to be the restriction of Dmap, namely $\mathsf{Dswap}\ a\ a' = \mathsf{Dmap}\ (a \leftrightarrow a')$. Conversely, from a swapping-based term-like structure we obtain a term-like structure by defining $\mathsf{Dmap}\ f$ as the composition

$$\mathsf{Dmap}\ a_1\ a_1' \circ \ldots \circ \mathsf{Dmap}\ a_n\ a_n \qquad (\$)$$

where

$$f = (a_1, a_1') \circ \ldots \circ (a_n, a_n') \qquad (\$\$)$$

[7] Indeed, saying that a function is supported by a set of atoms A (as in ASRT) is the same as saying that it is equivariant (i.e., commutes with the permutation action) with respect to all atoms outside of A (as in our corollary).

[8] Note that $(b \leftrightarrow b')\ a$ denotes the application of the swapping function $(b \leftrightarrow b')$ to a.

Note that any finite-support bijection f can be written as a composition ($$) of transpositions—although not uniquely. Thanks to the swapping-based term-like structure axioms, it can be shown that the result of ($) is the same for any decomposition ($$).

Proposition 36. The above correspondence (extended to morphisms in the expected way) is an equivalence between the categories of swapping-based term-like structures and that of term-like structures.

Based on this correspondence, it is straightforward to adapt our permutation based recursor to a swapping-based recursor. It employs swapping-based models, which feature an operator $\mathsf{Uswap} : \alpha \times \alpha \to \alpha\ U \to \alpha\ U$ instead of $\mathsf{Umap} : (\alpha \to \alpha) \to \alpha\ U \to \alpha\ U$, etc. Its final theorem is a swapping-based variant of Corollary 31 that replaces clause (M) with the following:

(M') $a, a' \notin A \longrightarrow H\,(\mathsf{map}_{\boldsymbol{T}}\ (a \leftrightarrow a')\ t) = \mathsf{Uswap}\ a\ a'\ (H\ t)$

Notice how, in this replacement, the premise $\mathsf{supp}\ f \cap A = \varnothing$ has become $a, a' \notin A$; this is because f is now the transposition $(a \leftrightarrow a')$, and therefore $\mathsf{supp}\ f = \{a, a'\}$. This last formulation allows us to see that:

- our recursion theorem is a slightly stronger form of the recursor in Norrish [2004]
- our constructor-based variation is a parameter-based improvement of the recursor in [Gheri and Popescu 2017]

Our Isabelle formalization contains formal proofs of this facts (for the syntax of λ-calculus).