General Bindings as 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 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. 2018. General Bindings as Functors. *Proc. ACM Program. Lang.* 1, ICFP, Article 1 (January 2018), 27 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, jasmin.blanchette@mpi-inf.mpg.de; 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.

2018. 2475-1421/2018/1-ART1 $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.

Much of this paper is dedicated to discovering 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 for the proof of their basic 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 yields a better understanding of the issues of acquiring fresh variables, allowing us to *go beyond finitary syntax*. Our theory seamlessly extends the scope of techniques for reasoning about bindings to infinitely branching and non-well-founded terms.

From an engineering perspective, our work targets the Isabelle/HOL proof assistant, a popular implementation of higher-order logic (Section 2.1). Our starting point is the class of bounded natural functors, abbreviated BNFs (Section 2.2). Unlike arbitrary functors, BNFs provide an explicit account of an element appearing as content in a shape, and multiple interconnected mechanisms to access elements inside shapes, via functorial operators on the function and on the relation level. Moreover, BNFs can be represented in higher-order logic and have been implemented in Isabelle/HOL, where they form the basis of a modular specification mechanism for (binding-unaware) datatypes (Section 2.3). The current work turns 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 look into the *effectiveness* aspect (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. This indicates that not only bound variables but also free variables are handled properly. This also suggests that we have identified a "sweet spot" between the assumptions and the guarantees when constructing datatypes. On the way, we provide a general account of standard reasoning principles (such as fresh induction in the style of nominal logic) and discover new principles (including fresh coinduction).

Our contribution improves on previous abstract frameworks such as nominal logic and other category-theoretic approaches in terms of generality, flexibility, and modularity. (Section 6). The definitions and theorems presented here have been mechanically checked in Isabelle/HOL. In the paper, we focus on the main ideas and intuitions.

2 PRELIMINARIES

We first need to set up the stage for our contribution, by presenting higher-order logic and 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 (HOL) with Hilbert choice, the axiom of infinity, and rank-1 polymorphism. HOL is based on Church's simple type theory [Church 1940]. It is the logic of Gordon's original HOL system [Gordon and Melham 1993] and of its many successors and emulators, notably 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. Thus, polymorphic types give rise to *type constructors*—i.e., operators on the collection of types. For convenience, we use the notation $\overline{\alpha}\ T$ both for a given polymorphic type $\overline{\alpha}\ T$ and its associated type constructor T. Moreover, we refer to α_i as the ith *input* of the type constructor $\overline{\alpha}\ T$.

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 $bool \to bool\ list \to bool\ list$.

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) "$\alpha\ list$ is a unary type constructors" and (2) "given any type α and any function $f : \alpha \to \alpha$, such and such holds." In the second case, f is a monomorphic function considered on a fixed type α. Following a convention from dependent type theory, we can indicate the type of a polymorphic function using the $\forall$ quantifier, as in $\mathrm{Cons} : \forall\alpha.\ \alpha \to \alpha\ list \to \alpha\ list$.

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 $\forall\alpha.\ \forall x : \alpha.\ x = x$.

The above conventions allow us to express the concepts in a more "semantic" style, closer to standard mathematical notation. For the purposes of this paper, we can forget we are working in HOL and think of types as nonempty sets.

The only primitive mechanism for defining new types in HOL is the *typedef* mechanism: For any existing (possibly polymorphic) type $\overline{\alpha}\ T$ and predicate $P : \overline{\alpha}\ T \to bool$ that is true at at least one point, we can introduce a type that corresponds to the (nonempty) set of all elements of α that satisfy P. We denote this type by $\{x : \overline{\alpha}\ T \mid P\ x\}$. In set theory, this mechanism corresponds to set comprehension, which is enabled by the separation axiom.

Unlike dependent type theory, HOL does not have (co)datatypes as primitives. However, datatypes [Berghofer and Wenzel 1999; Gunter 1994; Harrison 1995; Melham 1989; Traytel et al. 2012] and, more recently, codatatypes [Traytel et al. 2012] are supported via derived specification mechanisms. Users can write fixpoint definitions in an ML-style syntax, and the system (1) defines the type using nonrecursive type definitions, ultimately appealing to *ind* and $\to$; (2) defines the constructors and related operators; and (3) proves characteristic properties, such as injectivity of constructors and (co)induction principles.

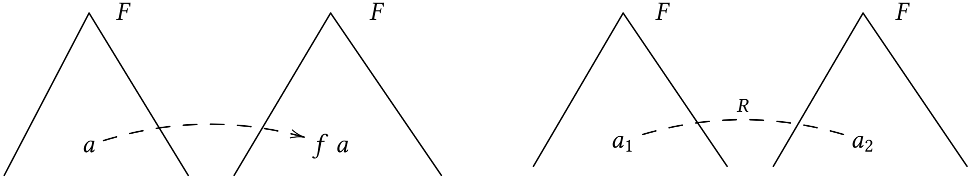


Fig. 1. $\mathsf{map}_F\ f$ (left) and $\mathsf{rel}_F\ R$ (right)

2.2 Bounded Natural Functors

We assume that (co)datatypes are available. Often it is useful to think in terms of type constructors. For example, *list* is a type constructor in one variable, whereas sum types (+) and product types (×) are binary type constructors. Most type constructors are not only operators on types but have a richer structure, that of *bounded natural functors* [Traytel et al. 2012].

Let $[n] = \{1, \ldots, n\}$. Let α *set* denote the powertype of α, consisting of sets of elements of α; it is isomorphic to $\alpha \to bool$.

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

(where fst and snd are standard projection functions on the product type ×). F is an n-ary *bounded natural functor* (*BNF*) 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.

(The requirement 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 mapper, in that mapping with a function f has the same effect as taking its graph Gr f and relating through Gr f. For example, if $n = 1$, we have $y = \mathsf{map}_F\ f\ x \Longleftrightarrow \mathsf{rel}_F\ (\mathsf{Gr}\ f)\ x\ y$.

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 mapper 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 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.

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

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 BNFs

We call the following *basic type constructors*: sum and product (both binary), for each fixed β the function space $\beta \to \alpha$ (unary) and, for each n and $i \in [n]$ the n-ary i-projection $\overline{\alpha}\, F = \alpha_i$. We will also consider the following *nonfree type constructors*: for any cardinal κ, the κ-bounded powerset $\alpha\ set^{\kappa}$ (containing all subsets of α such that $|\alpha| < \kappa$), of which a particular case (taking $\kappa = \aleph_0$) is the finite powertype denoted $\alpha\ fset$, and the finite multi-powerset (i.e., type of multisets), $\alpha\ mset$.

BNFs include the basic type constructors (the *basic BNFs*) and the nonfree type constructors (the *nonfree BNFs*), and are closed under composition and (least and greatest) fixpoint definitions [Traytel et al. 2012]. They also include non-syntactic types such as fuzzy-powertypes and probability distributions. These properties enable a modular approach to mixing and nesting any BNF when defining (co)datatypes. In Isabelle/HOL, users can additionally register new type constructors as BNFs if they define the BNF structure and prove the axioms.

Datatypes $\overline{\alpha}\ T$, where $|\overline{\alpha}| = m$, can be defined recursively from $(m+1)$-ary BNFs $(\overline{\alpha}, \tau)\, F$, by taking their least fixpoint (initial algebra), namely, the minimal solution of the following recursive equation up to isomorphism:

$$\overline{\alpha}\ T \ \simeq\ (\overline{\alpha}, \overline{\alpha}\ T)\, F$$

Concretely, we have the $\overline{\alpha}$-polymorphic bijection $\mathsf{ctor} : (\overline{\alpha}, \overline{\alpha}\ T)\, F \to \overline{\alpha}\ T$ and the minimality is expressed in the following *structural induction* proof principle:

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

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

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

If instead we interpret the above equation maximally, it yields the codatatype $\overline{\alpha}\ T$—we write the equation by adding the superscript ∞ as a reminder that it is interpreted maximally, which leads to non-well-founded entities (of possibly infinite depth):

$$\overline{\alpha}\ T\ \simeq^{\infty}\ (\overline{\alpha},\ \overline{\alpha}\ T)\, F$$

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

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

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

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

Above, $[(=)]^m$ indicates m occurrences of the equality relation, one for each component of $\overline{\alpha}$.

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

$$(\alpha, \alpha\ T)\,F \overset{\text{ctor}}{\rightarrow} \alpha\ T$$

(Thus, we have $m = 1$.) Fig. 2a 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 $\text{set}_1^F\ x$, which are members of α, and the items in $\text{set}_2^F\ x$, 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 ca 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. 2b pictures a recursive component path of in such a tree.

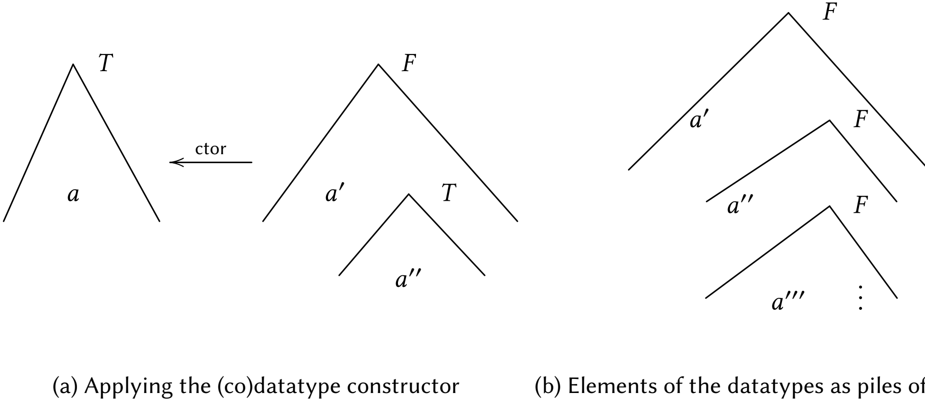
The essential property of the datatype is that all such trees are well founded, meaning that they all end in items t that have no recursive components ($\text{set}_2^F\ t = \varnothing$). This is precisely what the structural induction principle (SI) says, in a slightly different, higher-order formulation 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 $t \in \text{set}_F^2\,(\text{ctor}\ x)$.

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. 2 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 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 $\varphi : \alpha\ T \longrightarrow \alpha\ T \longrightarrow bool$ 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. For example, if we take $(\alpha, \tau)\,F$ to be $unit + \alpha \times \tau$ (where *unit* is a fixed singleton type), the associated datatype $\alpha\ T$ is $\alpha\ list$. Then, taking $(\alpha, \tau)\,F$ to be $\alpha + \tau\ list$, the associated datatype $\alpha\ T$ is $\alpha\ tree$, the type of α-labeled rose trees—more precisely, finitely branching well-founded rose trees. If instead of lists we use lazy lists, $\alpha\ llist$, defined the same as lists but taking the codatatype instead of the datatype, we obtain possibly infinitely branching well-founded rose trees. Moreover, if $\alpha\ tree$ is defined as a codatatype rather than a datatype, then the rose trees are not required to be well founded.

The above definitional modularity is matched by a proof-principle modularity: The (co)induction principle associated to a (co)datatype respects the abstraction barrier of the (co)datatypes nested

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



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

Fig. 2. Visualizing a datatype

in it, in that it does not refer to their definition or their constructors; instead, it only uses their BNF interfaces, consisting of mappers, setters 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

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

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}.

3 TOWARDS AN ABSTRACT NOTION OF BINDER

The literature describes a variety of binding notions relying on complex syntactic formats [Pottier 2006; Sewell et al. 2010; Urban and Kaliszyk 2012]. In contrast, here we ask a semantic question: Can we provide an abstract, syntax-free axiomatization of binders that subsumes the various syntactic formats and is *effective*, in the sense of catering for intuitive notions associated to syntax with bindings, such as free variables, alpha-equivalence, and substitution?

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 single variables a in single terms t, obtaining $\lambda a.\ t$. The term t may contain several free variables. If a is one of them, adding a λ-abstraction binds it. Suppose t is $b\ a$ ("b applied to a"), where a and b are distinct free 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 standard binders take into consideration a wider context than just the body. The "let" construct let $a = t_1$ in t_2 binds the variable a in the term t_2 but lets the term t_1 unaffected. In

the expression let $a = b\,a$ in $b\,a$ the first occurrence of a is the binding occurrence, the second occurrence is free (i.e., not in the scope of the binder), and the third occurrence is bound (i.e., in the scope of the binder). In general, we must distinguish between the components that fall under a binder's scope and those that do not.

To further complicate matters, there can be multiple terms affected by one binding variable. For example, 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, so that, 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, there can be multiple variables affecting a single term: $\lambda(a, b).\ t$ binds simultaneously 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 arbitrarily high complexity. The pattern-based "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 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 patterns, where l_i are unique labels.

3.2 Abstract Binder Types

A general feature of binders is that they distinguish between binding variables and the other entities, typically terms, which could be either inside or outside the scope of the binder. So we can think of a binder type as a type constructor $(\overline{\alpha}, \overline{\tau})\,F$, with $m = |\overline{\alpha}|$ and $n = |\overline{\tau}|$, 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, with the understanding that its actual binding takes place according to the binding dispatcher θ: If $(i, j) \in \theta$, then all the variables of type α_i occurring in x bind in all terms of type τ_j occurring in x.

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

The examples of Section 3.1 can be expressed as follows (writing α instead of α_1 if $m = 1$ and similarly for $\overline{\tau}$):

- 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 $(\alpha, \tau)\,F = \alpha \times \alpha \times \tau \times \tau$.
- For pattern-let $p = t_1$ in t_2, we take $m = 1$, $n = 2$, $\theta = \{(1, 2)\}$, and $(\alpha, \tau_1, \tau_2)\,F = \alpha\,P \times \tau_1 \times \tau_2$, where $\alpha\,P$ is the datatype defined recursively as $\alpha\,P \simeq \alpha \times \mathit{type} + (\mathit{label}, \alpha\,P)\,\mathit{record}$, where *type* and *label* are fixed types (as specified in the POPLmark challenge) and $(\beta_1, \beta_2)\,\mathit{record}$ is a type constructor that represents the type 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 important for accurately describing the binder's structure; but the actual terms corresponding to τ_1 and τ_2 will of course be allowed to be the same, which would correspond to the usage of F as $(\alpha, \tau, \tau)\, F$. One may wonder why the binder should care about potential terms that fall outside the scope of its binding variables. Why not keep this aspect outside our notion of binder? The answer is that this could lead to severe lack of precision, as 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 format. However, letting F unspecified brings a lot of flexibility compared to the syntactic approach. For example, F can incorporate arbitrarily complex notions of binders, including the datatype $\alpha\ P$ needed in the "pattern-let" case. In particular, it can seamlessly accommodate new situations. For example, capturing the above "parallel let" 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, our semantic notion is modular: We simply take a type constructor that achieves this: $(\alpha, \tau_1, \tau_2)\ F = (\alpha \times \tau_1)\ \mathit{list} \times \tau_2$, with $\theta = \{(1, 2)\}$. As another example, the type schemes in Hindley–Milner type inference [] 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, taking $(\alpha, \tau)\ F = \alpha\ \mathit{fset} \times \tau$, 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.*

This first proposal is not particularly impressive. It is extremely general, but it tells us nothing about how to construct actual terms with bindings or reasoning principles about them. To address this objection, 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 essentially follows our intuition associated to BNF, discussed in 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 effective, given all we can do with BNFs. In particular, we could use their mappers to perform renaming of bound variables, an essential operation for developing a theory of syntax with bindings. 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 no longer a functor, since its mapper 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 $\mathsf{map}_F\ f\ \mathsf{id}\ x$ 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 mapper, 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 $(\alpha, \tau)\, F = \{(a, b) : \alpha \times \alpha \mid a \neq b\} \times \tau$, fall in this category. We call this refined notion *map-restricted BNF* (*MRBNF*, or $\overline{\alpha}$*-MRBNF*) and use it as the basis of a new proposal.

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

MRBNFs retain the advantage of generality, while offering a sound mechanism for renaming bound variables. To validate this candidate, we ask two questions, which will be addressed in Sections 4 and 5: (1) How can nonrepetitive MRBNFs be constructed from potentially 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 (lazy) lists may contain repeated elements.

The absence of good fixpoint behavior implies that complex nonrepetitive MRBNFs cannot be built recursively from simpler pieces. 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 simpler BNF $\mathit{unit} + \alpha \times \beta$), we construct the MRBNF of nonrepetitive lists, $\{xs : \alpha\ \mathit{list} \mid \mathsf{nonrep}_{\mathit{list}}\ xs\}$. Likewise, from the pattern-let BNF $\alpha\, P$ (built recursively from the simpler BNF $\alpha \times \mathit{type} + (\mathit{label}, \beta)\ \mathit{record}$), we construct the MRBNF of nonrepetitive patterns, $\{xs : \alpha\, P \mid \mathsf{nonrep}_P\ xs\}$. 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 $p : \alpha\, P$, 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, that have 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 shape of a list is fully characterized by its length. So we can define

the notion of two members x, x' of $\alpha\, F$ having the same shape, written $\mathsf{sameShape}_F\; x\; x'$, to mean that there exists a relation such 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 $R := \top$. One can check that sameShape gives the intuitive notions for concrete (co)datatypes: For lists and lazy lists, it means "same length," and for various kinds of trees it corresponds to the notion of two trees becoming 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 $\mathsf{sameShape}_F\; x\; x'$, $|\mathsf{set}_F\; x'| \le |\mathsf{set}_F\; x|$. 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.

So we need a more abstract approach, not in terms of cardinality, but exploiting the functorial structure. An essential property of the nonrepetitive lists $as = [a_1, \ldots, a_n]$ is their ability to be *pattern-matched* against any other list $as' = [a'_1, \ldots, a'_n]$ of the same length n; and the pattern-matching process produces the function f that sends each a_i to a'_i (and leaves everything else 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', i.e., such that $x' = \mathsf{map}_F\; f\; x$. This gives us 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. Now we can define the corresponding nonrepetitive MRBNF:

Theorem 1. If $\alpha\, F$ is a BNF, 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 a nonrepetitive 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 inputs α_i, or more generally to any $\overline{\alpha}$-MRBNF $(\overline{\alpha}, \overline{\tau})\, F$, which can be further restricted to nonrepetitive members with respect to any of its unrestricted 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 $[(a, b),\ (a', b)]$ but not $[(a, b),\ (a, b')]$.

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 $m = |\overline{\alpha}|$, $n = |\overline{\tau}|$, together with a binding dispatcher $\theta \subseteq [m] \times [n]$; and 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.

To define actual terms, we must prepare for a dual phenomenon to the binding of variables: To bind variables, the terms must be allowed to have *free* variables in the first place. Thus, in addition to means for binding variables, we need means for injecting free variables into potential terms. We can upgrade F to handle this additional task: 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 $|\overline{\beta}| = |\overline{\alpha}| = m$ and employ F as $(\overline{\alpha}, \overline{\alpha}, \overline{\tau})\, F$ when defining actual terms. However, 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, let us 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—the fully general case, of multiple (mutually

recursive) term types is a straightforward but technical generalization. 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 $\overline{\alpha}\ T$.

Example 2. Consider the syntax of λ-calculus, where the collection $\alpha\ T$ of terms t with variables in α are defined by the following grammar:

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

Thus, a term is either a variable, or an abstraction, or an application. This is supported by our abstract scheme by taking $m = 1$, $n = 2$, $\theta = \{(1, 1)\}$ and $(\beta, \alpha, \tau_1, \tau_2)\ F = \beta + \alpha \times \tau_1 + \tau_2 \times \tau_2$, thus making $\alpha\ T$ satisfy the familiar recursive equation

$$\alpha\ T\ \simeq\ \alpha + \alpha \times \alpha\ T + \alpha\ T \times \alpha\ T$$

What is not visible in this final equation is how F distinguishes between:

- the free-variable type β and the binding-variable type α, which 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
- two different types of potential terms, τ_1 and τ_2, which ensures, via θ, that, in the second summand α binds in its neighboring $\alpha\ T$, but not in the $\alpha\ T$ occurring in the third summand.

This additional information is needed for the proper treatment of the bindings.

Thus, F handles both the notions of binding and of injecting free variables. Despite this dual role, we will stick to calling F a binder type. Multiple binding or free-variable injecting operators can of course be handled by F.

Example 3. Consider the extension of the λ-calculus syntax with parallel let binders:

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

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 relating α with τ_1 but not with τ_2, observe the notion that the term t, and not the terms t_i, are in the scope of the binding variables a_i.

Thus, during our discussion we have extended our MRBNF F with a further vector of inputs, $\overline{\beta}$, obtaining $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F$ with the following interpretation of its 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 $\overline{\alpha}$ only with respect to endobijections. But how about on $\overline{\beta}$? A natural answer would be again "full functor," 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 addition, useful operations like substitution should be allowed to introduce repetitions, e.g., by substituting a for a' while a was already free in the term, or by simultaneously substituting a for both a' and a''. So for now we will assume full functoriality on $\overline{\beta}$. It is time to formulate a refinement of Proposal 3 that reflects our later discussion:

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.

Moving forward with this refined proposal, we wish to see if it is able to accommodate the constructions on terms with bindings. We will use a small running example, which exhibits enough binding diversity.

Example 4. Consider a variation of the λ-calculus syntax where abstractions bind simultaneously two variables in two terms:

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

with the usual requirement that in $\lambda(a,b).\,t$ 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, so that we have $\mathsf{Inl} : \beta \to (\beta,\alpha,\tau)\,F$ and $\mathsf{Inr} : (\alpha\times\alpha)\ @\ \alpha\times\tau\times\tau \to (\beta,\alpha,\tau)\,F$.

5.1 Free Variables

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

- the elements of $\mathsf{set}_i^F\ x$ for $i\in[m]$, which are members of α_i and called the *top-free variables* $\mathsf{topFree}_i\ x$ below, representing the free variables injected by the topmost constructor of t,
- the elements of $\mathsf{set}_{m+i}^F\ x$ for $i\in[m]$, which are members of α_i and called the *top-binding variables* $\mathsf{topBind}_i\ x$ below, representing the the binding variables introduced by the topmost constructor of t,
- the elements of $\mathsf{set}_{2m+j}^F\ x$ for $j\in[n]$, which are members of $\overline{\alpha}\ T$ and called the *recursive components* $\mathsf{topRec}_j\ x$ of t below.

All of t's top-binding variables are assumed to be binding in all of t's recursive components, simultaneously. Thus, a free variables of a term will be either a top-free variables or, recursively, a free variable of some recursive component that is not among the top-binding variables. Formally, $\mathsf{FVars}_i\ t$ for $i\in[m]$ is defined inductively by the following rules:

$$\frac{a\in\mathsf{topFree}_i\ x}{a\in\mathsf{FVars}_i\ (\mathsf{ctor}\ x)}\qquad\qquad \frac{t\in\mathsf{topRec}_j\ x \qquad a\in\mathsf{FVars}_i\ t\setminus\begin{cases}\mathsf{topBind}_i\ x & \text{if } (i,j)\in\theta\\ \varnothing & \text{otherwise}\end{cases}}{a\in\mathsf{FVars}_i\ (\mathsf{ctor}\ x)}$$

In the context of our running Example 4, (here and later) let us assume that a,b,c,d etc. are mutually distinct variables. First, consider the term $t = \mathsf{Var}\ c$. It can be written as $\mathsf{ctor}\ x$, where $x = \mathsf{Inl}\ c$, so that $\mathsf{topFree}\,x = \{c\}$, $\mathsf{topBind}\,x = \varnothing$ and $\mathsf{topRec}\,x = \varnothing$. (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 $x = \mathsf{Inr}\,((a,b),\ (\mathsf{Var}\ a,\mathsf{Var}\ c))$, so that $\mathsf{topFree}\,x = \varnothing$, $\mathsf{topBind}\,x = \{a,b\}$ and $\mathsf{topRec}\,x = \{\mathsf{Var}\ a,\ \mathsf{Var}\ c\}$. 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 $\mathsf{Var}\ c\in\mathsf{topRec}\,x$ 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 via the map function of $\alpha\ T$, defined recursively as follows:

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

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

Intuitively, two terms should be alpha-equivalent if they are the same *save for a renaming of their bound variables*. More precisely, we have the following situation, pictured in Fig. 3:

- 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 one (marked as t_1) are positionwise alpha-equivalent to the recursive components of the other (marked as t_2)—in symbols, $\mathsf{map}_T\,\overline{f}\ t_1 \equiv t_2$.

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 $\equiv\ : \alpha\ T \to \alpha\ T \to$ *bool* inductively by the following clause:

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

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. Clearly, $f_i : \alpha_i \to \alpha_i$ must be a bijection, 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 top-binding in $\mathsf{ctor}\ x_1$. We thus take $\mathsf{cond}_i\,(f_i)$ to be

$$f_i\ \mathsf{bijective} \wedge \forall a \in (\textstyle\bigcup_{j:(i,j)\in\theta} \bigcup_{t_1 \in \mathsf{topRec}_j\, x_1} \mathsf{FVars}\ t_1) \setminus \mathsf{topBind}_i\, x_1.\ f_i\ a = a$$

Back to the context of Example 4, first note that, for every a, we have $\mathsf{Var}\ a \equiv \mathsf{Var}\ a$—this is shown by applying the definitional clause of $\equiv$ with taking f to be the identity. The first hypothesis can be immediately verified: Since $\mathsf{Var}\ a$ has no top-binding variables or recursive components, all that needs to be checked is the condition concerning the top-free variables, which is $a = a$.

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: nothing to check concerning positionwise equality of the top-free variables (since t_1 and t_2 have none); concerning the top-binding variables, checking that $f\ a = b$ and $f\ b = a$; concerning the recursive components, checking 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 $\mathsf{Var}\ (f\ a) \equiv \mathsf{Var}\ b$ etc., i.e., $\mathsf{Var}\ b \equiv \mathsf{Var}\ b$ etc. Finally, verifying the second hypothesis, $\mathsf{cond}\,(f)$, amounts to checking that f is bijective and that f is 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.

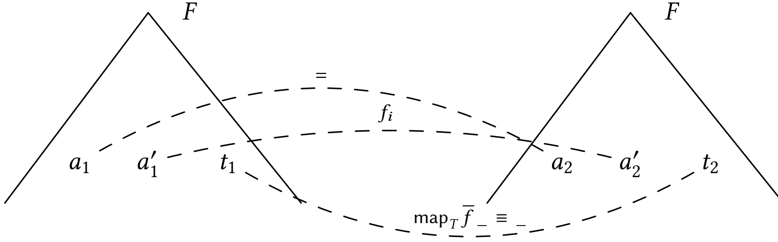


Fig. 3. Alpha-equivaence

The above is one of the several possible choices to define alpha-equivalence. First, one could pose harsher conditions on the f_i, allowing them to change *only* the top-binding variables, whereas we also allowed 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 consist in a symmetric formulation: Rather than renaming variables of the left term only, one renames the variables of both left and right terms to a third term, having all its variables distinct from those of the two. All these variants of introducing alpha-equivalence have different virtues in terms of the ease or elegance of proving various basic properties, but produce the same notion.

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 5. Alpha-equivalence is compatible with:

- the term constructor, in that $\mathsf{rel}_F\ [(=)]^m\ [(=)]^m\ [(\equiv)]^m\ x_1\ x_2$ 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 variables 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 mapper of T, in that, if $f_i : \alpha \to \alpha$ for $i \in [m]$ are (endo)bijections, then $t_1 \equiv t_2$ implies $\mathsf{map}_T\ \overline{f}\ t_1 \equiv \mathsf{map}_T\ \overline{f}\ t_2$

Theorem 6. Alpha-equivalence is an equivalence relation on $\overline{\alpha}\ T$.

5.3 Alpha-Quotiented Terms

Alpha-equivalence being an equivalence compatible with the term constructor, free variable operator and mapper operators on $\overline{\alpha}\,T$ allows us to define the quotient $\overline{\alpha}\,\boldsymbol{T} = (\overline{\alpha}\,T)/\equiv$, and lift these operators to $\overline{\alpha}\,\boldsymbol{T}$, yielding using overloaded notation $\mathsf{ctor} : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\,\boldsymbol{T}]^n)\, F \to \overline{\alpha}\,\boldsymbol{T}$, $\mathsf{FVars} : \overline{\alpha}\,\boldsymbol{T} \to \alpha\ \mathit{set}$ 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.[2] Its injectivity fails due to quotienting, which allows one to bind different variables in different terms but obtain equal results. E.g., 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 $x_2 = \mathsf{Inr}\ ((b, a), (\mathsf{Var}\ b, \mathsf{Var}\ c))$, however, $x_1 \neq x_2$, since, e.g., $\mathsf{Var}\ a \neq \mathsf{Var}\ b$.

On the other hand, we have the following injectivity up to a renaming, which follows immediately from the definition of α and its compatiblity with ctor and map_T:

Proposition 7. Given $x_1, x_2 : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\,\boldsymbol{T}]^n)\, F$, the following are equivalent:

[2] Using the fact that ctor stays surjective when switching from $\overline{\alpha}\,T$ to $\overline{\alpha}\,\boldsymbol{T}$, we will continue our ctor-matching habit, writing typical terms as $\mathsf{ctor}\ x$ where $x : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\,\boldsymbol{T}]^n)\, F$.

- $\mathsf{ctor}\ x_1 = \mathsf{ctor}\ x_2$
- there exist functions $f_i : \alpha_i \to \alpha_i$ satisfying $\mathsf{cond}_i\ (f_i)$ for $i \in [m]$ such that $\mathsf{rel}_F\ [(=)]^m\ (\mathsf{Gr}\ f_1) \cdots (\mathsf{Gr}\ f_m)\ [(\lambda t_1, t_2.\ \mathsf{map}_T\ \overline{f}\ t_1 = t_2)]^n\ x_1\ x_2$.

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 can thus summarize the definition of this binding-aware datatype as

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

emphasizing that this time we do not have a true isomorphism, but only an isomorphism up to the notion of alpha-equivalence determined by θ.

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

Terminology. From now on, we will apply the following convention, which is customary in the literature of syntax with bindings: We call *terms* the members of $\overline{\alpha}\ \boldsymbol{T}$ (which so far were called quotiented terms). When we need to refer to the members of $\overline{\alpha}\ T$, we call them *raw terms*.

5.4 Variable-for-Variable Substitution

An important operation we wish to have on $\overline{\alpha}\ \boldsymbol{T}$ is capture-avoiding substitution. With our current infrastructure, we should hope to be able to define *simultaneous substitution of variables for variables*, $\mathsf{vsub} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$. 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 $f\ a$. Then unary substitution will be a particular case of simultaneous substitution, defined as $t\,[a/b] = \mathsf{vsub}\ f_{a,b}\ t$ where $f_{a,b}$ sends b to a and all other variables to themselves.

A first candidate for vsub that suggests itself is $\mathsf{map}_{\boldsymbol{T}}$ inherited by $\overline{\alpha}\ \boldsymbol{T}$ from $\overline{\alpha}\ T$. However, this operator is not suitable, since it only works with bijections f_i. The fundamental property of vsub we are after concerns its recursive behavior on terms of the form $\mathsf{ctor}\ x$. The following should hold:

$$\mathsf{vsub}\ \overline{f}\ (\mathsf{ctor}\ x) = \mathsf{ctor}\ (\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\mathsf{vsub}\ \overline{f}]^n\ x) \quad \text{if } \forall i \in [m].\ \mathsf{supp}\ f_i \cap \mathsf{topBind}_i\ x = \varnothing \quad (*)$$

where $\mathsf{supp}\ f$, the *support* of an (endo)function f, is defined as the union between the set $\{a : \alpha \mid f\ a \neq a\}$ and its image through f, $\{f\ a \mid f\ a \neq a\}$. That is, $\mathsf{vsub}\ \overline{f}$ should distribute over the term constructor, provided the f_i does not interfere with the top-binding variables.

To achieve this property, it must be possible to replace any binding variables that happen to be in $\mathsf{supp}\ f_i$ with some variables that are outside $\mathsf{supp}\ f_i$. Moreover, all these new variables should also not clash with the free variables of the term under consideration, $\mathsf{ctor}\ x$—i.e., they should not belong to $\mathsf{FVars}_i\ (\mathsf{ctor}\ x)$. Such a replacement process, e.g., producing x' from x, would be invisible as far as $\mathsf{ctor}\ x$ is concerned: Alpha-equivalence being equality on $\alpha\ \boldsymbol{T}$, $\mathsf{ctor}\ x'$ and $\mathsf{ctor}\ x$ would be equal. By this argument, whenever we want to apply substitution, it would be legitimate to assume $\mathsf{supp}\ f_i \cap \mathsf{topBind}_i\ x = \varnothing$ for all $i \in [m]$.

An issue is that both the support of f_i and the set of free variables of the terms may be too large—indeed, for all we know, they may even exhaust the entire type α_i. So we need to find 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 is not *a priori* fixed: Since $\overline{\alpha}\ \boldsymbol{T}$ is a type constructor, we are allowed to consider a sufficiently large supply of variables. Noting that the cardinal bd_F is an overapproximation of the branching factor of terms regarded as trees (as

discussed in Section 2.3) and that the free variables are located at the leaves of such trees, we can prove that $\forall i \in [m].\ |\mathsf{FVars}_i\ t| \leq \mathsf{bd}_F$ for all raw terms t, hence also for all terms t. Thus, any α_i whose cardinality is strictly greater than bd_F should ensure that $|\mathsf{FVars}_i\ t| < |\alpha_i|$. We can therefore prove the existence of a function $\mathsf{vsub} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$ satisfying $(*)$ in case $\mathsf{bd}_F < |\alpha_i|$ and $|\mathsf{supp}\ f_i| < |\alpha_i|$.

However, this solution seems to require too many variables for the particular case of finitary functors F, i.e., such that $\mathsf{bd}_F = \aleph_0$ (the countable cardinal): This is the case of all finitely branching datatypes. With our approach, here we would need α to be uncountable, which contrasts the fact that countably infinitely many variables are actually enough.

One could argue that variable countability is not important, and any infinite cardinality would do. Indeed, some textbooks only assume "an infinite supply of variables," without mentioning countability. On the other hand, countability becomes important when considering practical aspects such as executability, so it is worth salvaging it 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 for $|\alpha_i| \geq \mathsf{bd}_F$ (with nonstrict equality) if $|\alpha_i|$ is a regular cardinal. We obtain:

Theorem 8. There exists a (polymorphic) function $\mathsf{vsub} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$ satisfying $(*)$ for all α_i and f_i in case $|\alpha_i|$ is regular, $\mathsf{bd}_F \leq |\alpha_i|$ and $|\mathsf{supp}\ f_i| < |\alpha_i|$.

This solution works for any MRBNF F: Since there exist arbitrarily large regular cardinals, for any bd_F we can choose suitable α_is (for example, α_i having the cardinal successor of bd_F as cardinality suffices). Moreover, the solution gracefully generalizes the finitary case: Because $\aleph_0$ is regular, for bd_F countable we can choose countable α_is.

5.6 Term-for-Variable Substitution?

So far, we only discussed variable-for-variable substitutions. Often one wishes to perform a *term for variable* substitution, again in a capture-avoiding way. For example, 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. For example, the process terms in the π-calculus [Milner 2001] contain channel variables (names), which can only be substituted by other channel variables, not by processes.

So when is term-for-variable substitution possible? The key to answering this is to note that, unlike the π-calculus, the λ-calculus allows for the embedding of single variables into terms. This is done either explicitly via a "Var" operator, or implicitly by declaring variables being terms.

It turns we can express such situations abstractly in our framework, by requiring that our framing MRBNF $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F$ can itself accommodate such embeddings. Namely, we assume injective natural transformations $\eta_i : \beta_i \to (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F$ for $i \in [m]$[3], such that $\mathsf{set}_i^F(\eta_i\ a) = \{a\}$. Then it also follows from η_i's naturality that $\mathsf{set}_k^F(\eta_i\ a) = \varnothing$ for all $k \neq i$. Moreover, we assume that η_i is the only source of variables in F: for all x such there is no a with $x = \eta_i\ a$, we require set_i^F to be empty. The injections of variables into terms, $\mathsf{Var}_i : \alpha_i \to \overline{\alpha}\ \boldsymbol{T}$, is defined as $\mathsf{Var}_i = \mathsf{ctor} \circ \eta_i$ and can be shown to be an injection. For the syntax of our running Example 4, where $(\beta, \alpha, \tau)\, F = \beta + (\alpha \times \alpha) \mathbin{@} \alpha \times \tau \times \tau$, we have that $\eta : \beta \to (\beta, \alpha, \tau)\, F$ is the injection of the leftmost summand; and similarly for the syntax of λ-calculus (Example 2).

[3] In principle, it is also possible to define the term for variable substitution only for a subset $I \subseteq [m]$ of free variables.

Now we can define simultaneous term-for-variable substitution, satisfying a property similar to that of variable-for-variable substitution:

$$\text{sub}\ \overline{f}\ (\text{ctor}\ x) = \begin{cases} f_i\ a & \text{if } x \text{ has the form } \eta_i\ a \\ \text{ctor}\ (\text{map}_F\ [\text{id}]^m\ [\text{id}]^m\ [\text{sub}\ \overline{f}]^n\ x) & \text{otherwise} \end{cases} \qquad (**)$$
$$\text{provided that } \forall i \in [m].\ \text{supp}\ f_i \cap \text{topBind}_i\ x = \varnothing$$

The support of a function $f_i : \alpha_i \to \overline{\alpha}\ \boldsymbol{T}$ is defined as the natural generalization of the support of functions of type $\alpha_i \to \alpha_i$, taking into account the free variables of the terms: as the union between the set $\{a : \alpha_i \mid f_i\ a \neq \text{Var}\ a\}$ and its image through $\text{FVars}_i \circ f_i$, namely $\bigcup_{a \in \text{supp}\ f_i} \text{FVars}_i\ (f_i\ a)$.

Note that, for a term $t = \text{ctor}\ x$, saying that x having the form $\eta_i\ a$ is the same as saying that t has the form $\text{Var}_i\ a$—so that corresponds to the base case of a variable (i.e., a term that is the embedding of a variable) in the recursive equation for sub.

The "otherwise" case is the recursive case of a non-variable constructor. The recursive call context of sub is $\text{ctor}\ (\text{map}_F\ [\text{id}]^m\ [\text{id}]^m\ [_]^n\ x)$, which differs from vsub's one, $\text{ctor}\ (\text{map}_F\ \overline{f}\ [\text{id}]^m\ [_]^n\ x)$. The difference stems from the fact that sub makes no attempt to substitute any top-free variables that are not part of a base case $\text{Var}_i\ a$. Our assumptions about η_i make sure that there is no loss by doing this. Indeed, the following property is a consequence of η_i being the only source of variables.

Proposition 9. If a term ctor x does not have the form $\text{Var}_i\ a$, then the set $\text{topFree}_i\ x$ is empty.

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

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

5.7 Proof Principle

Let *par* be a type, of items we will refer to as "parameters," and a predicate $\varphi : \overline{\alpha}\ T \to par \to bool$ (taking as inputs raw terms and parameters). If we want to prove $\forall t : \overline{\alpha}\ T.\ \forall p : par.\ \varphi\ t\ p$, we have at our disposal the structural induction associated with the datatype $\overline{\alpha}\ T$ (discussed in Section 2.3), which says that to obtain the above it suffices to prove that, for each term t, the predicate $\lambda t.\ \forall p : par.\ \varphi\ t\ p$ holds for it provided that holds for t's recursive components:

$$\forall (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ T]^n)\ F.\ (\forall j \in [n].\ \forall t \in \text{topRec}_j\ x.\ \forall p : par.\ \varphi\ t\ p) \longrightarrow (\forall p : par.\ \varphi\ (\text{ctor}\ x)\ p)$$

For terms (i.e., members of $\overline{\alpha}\ \boldsymbol{T}$), we can obtain a improvement of this principle, if in addition we assume that the parameters come with a notion of (free) variable such that the variables of all parameter are fewer than our total supply. In this case, we have the following *fresh structural induction* (*FSI*) proof principle, in the style of nominal logic [Pitts 2006]:

Theorem 11 (FSI). Let $\text{PVars}_i : par \to \alpha_i\ set$ with $\forall p : par.\ |\text{PVars}_i\ p| < |\alpha_i|$. Given a predicate $\varphi : \overline{\alpha}\ \boldsymbol{T} \to par \to bool$, if the condition

$$\forall x : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\ \boldsymbol{T}]^n)\ F.\ (\forall j \in [n].\ \forall t \in \text{topRec}_j\ x.\ \forall p : par.\ \varphi\ t\ p)$$
$$\longrightarrow (\forall p : par.\ (\forall i \in [m].\ \text{PVars}_i\ p \cap \text{topBind}_i\ x = \varnothing) \longrightarrow \varphi\ (\text{ctor}\ x)\ p)$$

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

Above, we highlighted the two differences from raw-term structural induction: We assume that parameters p come with sets of variables $\text{PVars}_i\ p$ which are smaller than α_i. This allows us to weaken what we must prove in the induction step for a given term ctor x, by allowing us to further assume that parameter's variables are fresh for (i.e., disjoint with) its top-binding variables.

The (FSI) principle is very useful when the parameters are themselves terms, variables, or functions on them, which is often the case. The properties of the substitution operators vsub and sub are a main customer of this principle. For example, consider the property of vsub commuting with composition of small-support endobijection:

Proposition 12. We have $\mathsf{vsub}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m) = \mathsf{vsub}\ \overline{g} \circ \mathsf{vsub}\ \overline{f}$ 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 $i \in [m]$.

Its proof goes by (FSI), taking the parameters to be tuples $(\overline{f}, \overline{g})$ of endofunctions of small support, $\mathsf{PVars}_i\ (\overline{f}, \overline{g})$ to be $\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i$, and $\varphi\ t\ (\overline{f}, \overline{g})$ to be $\mathsf{vsub}\ (g \circ f)\ t = (\mathsf{vsub}\ \overline{g} \circ \mathsf{vsub}\ \overline{f})\ t$. In the inductive case, we must prove

$$\mathsf{vsub}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m)\ (\mathsf{ctor}\ x) = \mathsf{vsub}\ \overline{g}\ (\mathsf{vsub}\ \overline{f}\ (\mathsf{ctor}\ x))$$

assuming that the fact holds for all recursive components of x and that the binding variables of x are not in $\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i$ for each $i \in [m]$. Thanks to this second assumption, we are able push the substitution inside the components of ctor x according to the recursive law $(*)$ for vsub, thus reducing what we need to prove to

$$\begin{aligned}&\mathsf{ctor}\ (\mathsf{map}_F\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m)\ [\mathsf{id}]^m\ [\mathsf{vsub}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m)]^n\ x) = \\ &\mathsf{ctor}\ (\mathsf{map}_F\ \overline{g}\ [\mathsf{id}]^m\ [\mathsf{vsub}\ g]^n\ (\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\mathsf{vsub}\ f]^n\ x)).\end{aligned}$$

From here on, the fact follows easily from the induction hypothesis, applying the functoriality and congruence properties of F.

5.8 Non-Well-Founded Terms

So far, we discussed the theory of well-founded terms framed by an abstract binder type F and a binding dispatcher θ. A similar development results in a theory for possibly non-well-founded terms, yielding non-well-founded terms modulo the alpha-equivalence induced by θ.

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

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

Given this inductive-coinductive asymmetry, it seems that there is no hope for a duality principle, which allows proof reuse, or at least the copying of proofs from well-founded to non-well-founded terms. However, a solution comes from an unexpected direction: 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, which means that its least and greatest fixpoints are equal, meaning $\equiv\ = \mathsf{lfp}\ \mathsf{Op}_{\equiv} = \mathsf{gfp}\ \mathsf{Op}_{\equiv}$.[4] 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.

[4] The symmetric statement is not true: When taking $\overline{\alpha}\ T$ to consist of non-well-founded terms, the same operator $\mathsf{Op}_{\equiv}$ will no longer have a unique fixpoint. Indeed, uniqueness relies on the well-foundedness of the terms.

The consequence of the above identities is remarkable: Many properties concerning the constructor, alpha-equivalence, free variables, the mapper, and their combination on raw terms, including Theorems 5 and 6 (which bootstrap the construction of $\overline{\alpha}\ \boldsymbol{T}$), can be given *identical* proofs.

All the theorems shown in Sections 5.1 to 5.6 hold for non-well-founded terms as well, with identical formulations. In particular, the issues with acquiring a sufficient amount of fresh variables can be solved using the same approach, which already caters for infinitary entities.

Concerning the notion of binding-aware proof principles for $\overline{\alpha}\ \boldsymbol{T}$, we encounter an interesting discrepancy from the inductive case. We can prove a fresh variation the coinduction proof principle (SC) presented in Section 2.3, using the ideas discussed in Section 5.7, where we again emphasize the enhancements compared to SC:

Theorem 13 (FSC). Let $\mathsf{PVars}_i : par \to \alpha_i\ set$ with $\forall p : par.\ |\mathsf{PVars}_i\ p| < |\alpha_i|$. Given a binary relation $\varphi : \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T} \to par \to bool$, if the condition

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

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

However, this proof principle turns out to be not as useful as its inductive counterpart. Consider the task of proving Proposition 12 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 $\mathsf{PVars}_i\ (\overline{f}, \overline{g})$ to be $\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i$. We let $\varphi\ t_1\ t_2\ (\overline{f}, \overline{g})$ be $\exists t.\ t_1 = \mathsf{vsub}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m)\ t\ \wedge$ $t_2 = (\mathsf{vsub}\ \overline{g} \circ \mathsf{vsub}\ \overline{f})\ t$. Then, since the fact to be proved can be rephrased as the conclusion of (FSC), it suffices to verify its hypothesis. So we assume that, for all endofunctions of small support f_i and g_i, (1) $\mathsf{ctor}\ x_1 = \mathsf{vsub}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m)\ t$ and $\mathsf{ctor}\ x_2 = (\mathsf{vsub}\ \overline{g} \circ \mathsf{vsub}\overline{f})\ t$, 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 vsub under the constructor in the equalities (1), obtaining (4) $\mathsf{ctor}\ x_1 = \mathsf{ctor}\ x_1'$ and $\mathsf{ctor}\ x_2 = \mathsf{ctor}\ x_2'$ where

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

At this point, we are stuck: To prove (3), what seems to be required is that (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 seem to 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 in the above proof is a certain synchronization requirement between the top-binding variables of x_1 and x_2, $\mathsf{topBind}_i\ x_1$ and $\mathsf{topBind}_i\ x_2$, which is not accounted for by the freshness hypothesis. To accommodate such a synchronization, we prove a different enhancement of the structural coinduction principle, (ESC). Instead on focusing on explicitly avoiding clashes with presumptive parameters, the principle 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 14 (ESC). Given a binary relation $\varphi : \overline{\alpha}\,\boldsymbol{T} \to \overline{\alpha}\,\boldsymbol{T} \to bool$, if the condition

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

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

(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.

5.9 Modularity Considerations

We have started with θ and $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\,F$, an $\overline{\alpha}$-MRBNF and have constructed the binding-aware datatype $\overline{\alpha}\,\boldsymbol{T}$ as $\overline{\alpha}\,\boldsymbol{T} \simeq_\theta (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\,\boldsymbol{T}]^n)\,F$ (and also the binding-aware codatatype—our next discussion applies to it as well). We can prove the following:

THEOREM 15. $\overline{\alpha}\,\boldsymbol{T}$ is an $\overline{\alpha}$-MRBNF, whose mapper is $\mathsf{map}_{\boldsymbol{T}}$ and whose setters are FVars_i.

This suggests that our framework are modular, in that we can employ $\boldsymbol{T}$ in further constructions of binding-aware datatypes. And indeed, this is the case if we want to use the variables $\overline{\alpha}$ "exported" by $\overline{\alpha}\,\boldsymbol{T}$ as binding variables. For example, if $|\overline{\alpha}| = 1$ and we take $(\beta, \alpha, \tau)\,F'$ to be $\beta + \alpha\ \boldsymbol{T} \times \tau$, then F' is an α-MRBNF, which can further build 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 $|\overline{\alpha}| = 1$ 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 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 currently requires full functoriality of $(\beta, \alpha, \tau)\,F'$ on the first input, β. Unfortunately, this second scenario, which fails in our current approach, seems to be the most useful. The next example illustrates a common instance of this scenario: a syntactic category of types allows binding type variables, while it also participates as annotations in a syntactic category of terms which also allows binding type variables.

EXAMPLE 16. Consider the syntax of System F types:

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

considered up to the notion of alpha-equivalence standardly induced by the $\forall$ binders. In our framework, this can be expressed 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 the syntax of (Church-style) System F terms, where we write a' for term variables:

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

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 $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\,F' = \beta_2 + \alpha_1 \times \tau_1 + \alpha_2 \times \beta_1\,\boldsymbol{T} \times \tau_2 + \tau_3 \times \tau_3$ with $|\overline{\beta}| = |\overline{\alpha}| = 2$ and $|\overline{\tau}| = 3$.

Indeed, this would give the overall fixpoint equation:

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

In this scheme, α_1 stores the System F type variables, and α_2 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 currently not possible, because, 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 also map-restricted on β_1.

The above problem would disappear if $\boldsymbol{T}$ were a full functor (with respect to arbitrary functions). However, the mapper'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 reason its 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 also have the capture-avoiding substitution operator $\mathsf{vsub} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\ \boldsymbol{T} \to \overline{\alpha}\ \boldsymbol{T}$. 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 (smaller 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 $\overline{\alpha}\ T$ and its properties. It turns out that the answer is no. It is enough to assume functoriality with respect to small-support endofunctions in to recover everything we developed, while performing minor changes to the definitions—namely, assuming 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 the final revision of our proposal for binder types:

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

- *distinguishes between free-variable, binding-variable and potential term inputs and*
- *puts a small-support endobijection map restriction on the binding-variable inputs*
- *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, $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\ F$ will be assumed to be functor with respect to endofunctions of small support on $\overline{\beta}$, with respect to small-support endobijections on $\overline{\alpha}$ and with arbitrary functions (i.e., a full functor) on $\overline{\tau}$. Note that full functoriality on $\overline{\tau}$ cannot be relaxed, since it is crucial for being able to solve the fixpoint equations that defined 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 the "free" or "binding" qualifiers of the corresponding vector of inputs is empty.

This final notion of MRBNF achieves useful modularity, in the sense that the free variables of terms are really a "free" MRBNF component:

Theorem 17. $\overline{\alpha}\ \boldsymbol{T}$ *is an* $\overline{\alpha}$*-free MRBNF, whose mapper is* vsub *and whose setters are* FVars_i.

We conclude by defining MRBNFs formally in their full glory. All previously shown theorems hold with respect to this definition.

Definition 2. 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) $(\forall i \in [m_1].\ |\mathsf{supp}\ u_i| \le \mathsf{bd}_F) \wedge (\forall i \in [m_2].\ |\mathsf{supp}\ v_i| \le \mathsf{bd}_F \wedge \mathsf{bij}\ v_i) \longrightarrow$
$\mathsf{rel}_F\ \overline{u}\ \overline{v}\ \overline{R}\ x\ y \iff \exists z.\ (\forall i \in [n].\ \mathsf{set}_F^{m_1+m_2+i}\ z \subseteq \{(a, a') \mid R_i\ a\ a'\})$
$\wedge\ \mathsf{map}_F\ [\mathsf{id}]^{m_1}\ [\mathsf{id}]^{m_2}\ [\mathsf{fst}]^n\ z = x \wedge \mathsf{map}_F\ \overline{u}\ \overline{v}\ [\mathsf{snd}]^n\ z = y$

(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.,

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

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

$(\forall i \in [m_1].\ |\mathsf{supp}\ u_i| \le \mathsf{bd}_F) \wedge (\forall i \in [m_2].\ |\mathsf{supp}\ v_i| \le \mathsf{bd}_F \wedge \mathsf{bij}\ v_i) \longrightarrow$
$(\forall i \in [m_1].\ \mathsf{set}_F^i \circ \mathsf{map}_F\ \overline{u}\ \overline{v}\ \overline{f} = \mathsf{image}\ u_i \circ \mathsf{set}_F^i) \wedge$
$(\forall i \in [m_2].\ \mathsf{set}_F^{m_1+i} \circ \mathsf{map}_F\ \overline{u}\ \overline{v}\ \overline{f} = \mathsf{image}\ v_i \circ \mathsf{set}_F^{m_1+i}) \wedge$
$(\forall i \in [n].\ \mathsf{set}_F^{m_1+m_2+i} \circ \mathsf{map}_F\ \overline{u}\ \overline{v}\ \overline{f} = \mathsf{image}\ f_i \circ \mathsf{set}_F^{m_1+m_2+i};)$

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

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

(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.,

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

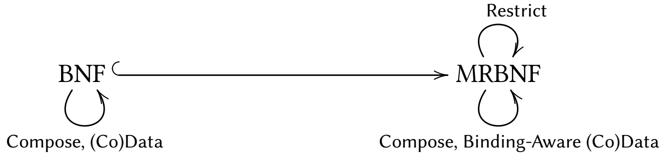


Fig. 4. Operations on BNFs and MRBNFs

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 mapper.

5.10 Summary of Our Findings

We have introduced MRBNFs, an extension of the notion of BNF that can accommodate arbitrarily complex statically scoped bindings without committing to any syntactic format. MRBNFs are multiple-input functors: On some inputs they behave like BNFs (i.e., are full functors, with respect to arbitrary functions), while on others they are restricted to either small-support endofunctions or small-support endobijections. The former type of restriction model the behavior of free variables, and the latter that of binding/bound variables.

Figure 4 summarizes the operations available on MRBNFs. First, there are the operations analogous to those on BNFs, namely composition and fixpoint constructions, the latter producing binding-aware (co)inductive terms, where the desired bindings are regulated by a binding dispatcher that relates the MRBNF inputs. (For BNFs and MRBNFs we have only shown the fixpoint construction for single (co)datatypes. A straightforward extension yields any number of mutually recursive (co)datatypes, given by a system of fixpoint equations.) In addition, there is a specific MRBNF operation that allows restricting an input to ensure nonrepetitiveness with respect to elements of that input—which caters for the specification of binding patterns.

Similarly to the case of BNFs, the (co)datatypes produced as MRBNF fixpoints come with a rich set of operations, including simultaneous variable-for variable and (under additional assumptions) term-for-variable substitution. They also come with theorems expressing properties of these operators (e.g., substitution is compositional) and binding-aware (co)induction proof principles.

6 RELATED WORK

There is a substantial 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. In some cases, one can draw a precise correspondence between the two frameworks. Thus, assume $\overline{\alpha}\,F$ is an $\overline{\alpha}$-binding MRBNF (having all the inputs as binding inputs) that is finitary, i.e., $\mathsf{bd}_F = \aleph_0$. Then we can fix $\overline{\alpha}$ to some countable types and define the nominal operator of swapping two atoms $a, a' : \alpha_i$ as vsub $\overline{f}$ where $f_i : \alpha_i \to \alpha_i$ swaps a and a' and all the other f_j's are identity functions. In this context, $\overline{\alpha}\,F$ can be shown to be a (multi-atom) nominal set having $\overline{\alpha}$ as sets of atoms. Our MRBNF restriction to small-support functions, as well as our fresh induction proof principle, are explicitly inspired by the nominal approach [Pitts 2006].

However, there are some important differences. A first difference rests in our employment of functors for modeling the presence of variables, rather than employing an atom-enriched notion of set, like the nominal sets. From a foundational point of view, our approach has the advantage of relying on a mechanism that is already present at the logical core: the dependence of the type constructors on type variables. Moreover, unlike nominal sets, which are assigned fixed collections

of atoms, the inputs to our functors have the freedom to change. This allows us to remove the finite support restriction, and accept terms that are infinitely branching and/or of infinite depths; to accommodate such larger entities, all we need to do is increase the inputs to a sufficiently large cardinality. Solutions to cope with infinitary structures in a nominal setting exist in the literature, but they either require a nonstandard set theoretical foundation [Gabbay 2007] or unnatural restrictions such as the infinitary terms being finitely supported [Kurz et al. 2013]. In addition, for achieving modularity in an infinitary setting, it is crucial that the supplies of variables be flexible, because we may not know in advance how many variables we need. For example, in a closed world, finitary terms need only countably many variables. However, if we later want to employ them in a wider context, e.g., as data communicable by some infinitely branching processes, the same terms would require a larger supply of variables. Our functor-based theory accommodates this phenomenon, because it is parameterized by unspecified collections of variables.

A second, more practical difference concerns the amount of "theory" (structure and properties) that is built in the framework versus the amount that must be 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). Consequently, nominal sets only know how to swap/permute, whereas our functors know how to apply arbitrary substitutions. Moreover, we can substitute *terms* for variables, which would correspond to a second-order extension of nominal logic (where members of nominal sets could replace atoms). Our richer infrastructure allows us to standardize the constructions of (1) complex binders and (2) binding-aware (co)datatypes. In contrast, nominal logic has no built-in mechanisms for such constructions. A number of datatype construction schemes have been proposed on top of nominal logic, which are confined either to syntactic formats [Pitts 2006; Urban and Kaliszyk 2012] or to basic binders [Kurz et al. 2013].

Concerning the reasoning infrastructure, we study and criticize the notion of fresh coiduction, and produce an improved version based on dynamically changing binding variables in terms. To our knowledge, these concepts represent unexplored territory in the nominal literature. On the other hand, there are various works in the vicinity of nominal logic that propose binding-aware recursion [Gheri and Popescu 2017; Gordon and Melham 1996; Norrish 2004; Pitts 2006; Popescu and Gunter 2011; Urban and Tasson 2005] and corecursion [Kurz et al. 2012, 2013] principles—an aspect currently missing in our framework.

Datatypes of terms with bindings modulo alpha-equivalence have been characterized as initial objects in presheaf categories [Ambler et al. 2003; Fiore et al. 1999; Hofmann 1999]. Like the category-theoretic constructions on top of nominal logic, and unlike our framework, these works deliver recursion principles but address only very simple notions of binder.

By contrast, some syntactic formats proposed in the literature [Pottier 2006; Sewell et al. 2010; Weirich et al. 2011] introduce sophisticated binders, and they have acted as a litmus test for the expressiveness of our semantic framework—which has the advantage of higher flexibility and modularity. We conjecture that our framework can capture all these formats, and we leave the rigorous proof of this as future work. Another development which is left as future work is employing our framework to give an elegant solution to the POPLmark challenge: In contrast to the current three complete solutions to this challenge, which construct the recursive patterns by hand, our framework will produce them automatically and treat them as an ordinary binder.

Our framework is currently formalized as a prototype in Isabelle/HOL, but will be turned into a full definitional package that will extend Isabelle's BNF-based (co)datatypes [Blanchette et al. 2014]. Its competitor in this arena will be Isabelle's Nominal2 package [Urban and Kaliszyk 2012]. This is an impressive piece of engineering that caters for complex binders specified as recursive datatypes, but suffers from the lack of flexibility and modularity specific to syntactic formats. It cannot be

combined with datatypes specified outside the framework, in particular, its nominal datatypes cannot nest standard (co)datatypes. Frameworks similar to Nominal2, but having less sophisticated notions of binders have been developed in Coq [Aydemir et al. 2007, 2008; Hirschowitz and Maggesi 2012; Schäfer et al. 2015].

A well-established alternative to nominal logic concerning the representation of syntax with bindings in proof assistants or logical frameworks is higher-order abstract syntax (HOAS) [Chlipala 2008; Despeyroux et al. 1995; Felty and Momigliano 2012; Harper et al. 1987; Kaiser et al. 2017], which reduces the bindings of the object syntax to those of the metalogic. Such shallow representations often allow for a substantial simplification in reasoning thanks to relying on implicit (meta-level) substitution and proof contexts [Felty et al. 2015; Pientka 2018]. On the other hand, due to their reliance on the single (typically simple) binder of the metalogic, they do not fare well on handling complex binding patterns.

REFERENCES

S. J. Ambler, Roy L. Crole, and Alberto Momigliano. 2003. A definitional approach to primitivexs recursion over higher order abstract syntax. In *Eighth ACM SIGPLAN International Conference on Functional Programming, Workshop on Mechanized reasoning about languages with variable binding, MERLIN 2003, Uppsala, Sweden, August 2003*. https://doi.org/10.1145/976571.976572

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*. 50–65. https://doi.org/10.1007/11541868_4

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

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

Stefan Berghofer and Markus Wenzel. 1999. Inductive Datatypes in HOL—Lessons Learned in Formal-Logic Engineering. In *TPHOLs '99 (LNCS)*, Vol. 1690. 19–36.

Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. 2014. Truly Modular (Co)datatypes for Isabelle/HOL. In *ITP 2014 (LNCS)*, Vol. 8558. Springer, 93–110.

Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 143–156. https://doi.org/10.1145/1411204.1411226

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 and Alberto Momigliano. 2012. Hybrid - A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax. *J. Autom. Reasoning* 48, 1 (2012), 43–105.

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. https://doi.org/10.1007/s10817-015-9327-3

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

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

Lorenzo Gheri and Andrei Popescu. 2017. A Formalized General Theory of Syntax with Bindings. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*. 241–261. https://doi.org/10.1007/978-3-319-66107-0_16

Andrew D. Gordon and Thomas F. Melham. 1996. Five Axioms of Alpha-Conversion. In *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*. 173–190. https://doi.org/10.1007/BFb0105404

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

Elsa L. Gunter. 1994. A Broader Class of Trees for Recursive Type Definitions for HOL. In *HUG '93 (LNCS)*, Vol. 780. Springer, 141–154.

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

John Harrison. 1995. Inductive Definitions: Automation and Application. In *TPHOLs '95 (LNCS)*, Vol. 971. Springer, 200–213.

André Hirschowitz and Marco Maggesi. 2012. Nested Abstract Syntax in Coq. *Journal of Automated Reasoning* 49, 3 (2012), 409–426.

Martin Hofmann. 1999. Semantical Analysis of Higher-Order Abstract Syntax. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. 204–213. https://doi.org/10.1109/LICS.1999.782616

Jonas Kaiser, Brigitte Pientka, and Gert Smolka. 2017. Relating System F and Lambda2: A Case Study in Coq, Abella and Beluga. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*. 21:1–21:19. https://doi.org/10.4230/LIPIcs.FSCD.2017.21

Alexander Kurz, Daniela Petrisan, Paula Severi, and Fer-Jan de Vries. 2012. An Alpha-Corecursion Principle for the Infinitary Lambda Calculus. In *Coalgebraic Methods in Computer Science - 11th International Workshop, CMCS 2012, Colocated with ETAPS 2012, Tallinn, Estonia, March 31 - April 1, 2012, Revised Selected Papers*. 130–149. https://doi.org/10.1007/978-3-642-32784-1_8

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

Thomas F. Melham. 1989. Automating Recursive Type Definitions in Higher Order Logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer, 341–386.

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.

Brigitte Pientka. 2018. POPLMark reloaded: mechanizing logical relations proofs (invited talk). In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. 1. https://doi.org/10.1145/3167102

Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Inf. Comput.* 186, 2 (2003), 165–193. https://doi.org/10.1016/S0890-5401(03)00138-X

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

Andrei Popescu and Elsa L. Gunter. 2011. Recursion principles for syntax with bindings and substitution. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 346–358. https://doi.org/10.1145/2034773.2034819

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 *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. 359–374. https://doi.org/10.1007/978-3-319-22102-1_24

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. https://doi.org/10.1017/S0956796809990293

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 2012*. IEEE Comput. Soc., 596–605.

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). https://doi.org/10.2168/LMCS-8(2:14)2012

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

Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. 2011. Binders unbound. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 333–345. https://doi.org/10.1145/2034773.2034818