Animating MRBNFs: Truly Modular Binding-Aware Datatypes in Isabelle/HOL

Jan van Brügge
Heriot-Watt University, Edinburgh, UK

Andrei Popescu
University of Sheffield, UK

Dmitriy Traytel
University of Copenhagen, Denmark

Abstract

Nominal Isabelle provides powerful tools for meta-theoretic reasoning about syntax of logics or programming languages, in which variables are bound. It has been instrumental to major verification successes, such as Gödel's incompleteness theorems. However, the existing tooling is not compositional. In particular, it does not support nested recursion, linear binding patterns, or infinitely branching syntax. These limitations are fundamental in the way nominal datatypes and functions on them are constructed within Nominal Isabelle. Taking advantage of recent theoretical advancements that overcome these limitations through a modular approach using the concept of map-restricted bounded natural functor (MRBNF), we develop and implement a new definitional package for binding-aware datatypes in Isabelle/HOL, called MrBNF. We describe the journey from the user specification to the end-product types, constants and theorems the tool generates. We validate MrBNF in two formalization case studies that so far were out of reach of nominal approaches: (1) Mazza's isomorphism between the finitary and the infinitary affine λ-calculus, and (2) the POPLmark 2B challenge, which involves non-free binders for linear pattern matching.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases syntax with bindings, datatypes, inductive predicates, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.11

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.15756655`

Funding *Andrei Popescu*: Acknowledges support from the EPSRC grants EP/V039156/1 (*Security of Digital Twins in Manufacturing*) and and EP/X015114/1 *Safe and secure COncurrent programming for adVancEd aRchiTectures* (*COVERT*).
Dmitriy Traytel: Acknowledges support from the Novo Nordisk Foundation (start package grant NNF20OC0063462) and the Independent Research Fund Denmark (Sapere Aude grant 3120-00057B, titled *DISCOVER: Distributed Streaming Computations Verified*).

Acknowledgements We thank the ITP reviewers for their insightful comments. Martin Elsman challenged MrBNF with advanced syntaxes, which overall resulted in our tool becoming more robust. The authors are listed alphabetically regardless of individual contributions or seniority.

1 Introduction

Most programming languages involve variable-binding constructs, or simply binders, such as the lambda abstraction or the recursive and non-recursive let operators. For the study of these languages' metatheory, the specific choice of bound variable names in a language expression is immaterial. For example, it is customary to treat the lambda calculus expressions $\lambda x.\ x$ and $\lambda y.\ y$ as being syntactically equal and to choose bound variables in a way that avoids name clashes with surrounding free variables – this is known as Barendregt's variable convention [6].

The mechanization of programming language metatheory in proof assistants struggles to keep up with this informal convention. The POPLmark challenge [4] initiated a flurry of approaches to mechanized binders, each with its own strengths and weaknesses (§2). The used approaches can be categorized into three main paradigms: (1) the *nameless* representation that replaces bound variables in terms with pointers to the binding position [17, 23]; (2) the *nameful* or *nominal* representation that includes bound variable names but identifies terms modulo alpha-equivalence, i.e., up to bound variable renaming [18]; and (3) the *reductive* representation that embeds the programming language's binders into the metalogic's binders [19, 28, 30].

The nominal representation faithfully encodes Barendregt's variable convention in that the accompanying reasoning principles (e.g., nominal induction) allow their users to assume that bound variables do not clash with surrounding free variables whenever bound variables are introduced. Nominal Isabelle [21] implements the nominal representation in the Isabelle/HOL proof assistant with successful applications ranging from Gödel's incompleteness theorems [29] to verifying the correctness of Haskell's compiler optimizations [13] and the security of authenticated data structures [14]. All these developments use the expected binder constructs: lambda abstractions, existential quantifiers, and (parallel) recursive let constructs.

Nominal Isabelle is fundamentally restricted to syntaxes with finite support, i.e., expressions may only contain finitely many free and bound variables. This rules out applications like Mazza's infinitary affine lambda calculus [22]. Moreover, nested recursion, which e.g., is needed to model function applications to multiple arguments, is not directly supported. Sometimes this limitation can be overcome by using mutual recursion instead, but this workaround is limited to cases where the nesting type is a datatype itself. Nesting through coinductive datatypes such as streams or lazy lists or non-free structures such as finite or countable sets remains problematic. Also nominal datatypes, even in their flexible variant provided by Nominal 2 [42], cannot directly incorporate linear patterns, which are required in most of the complex binder structures including those of POPLmark 2B.

Blanchette et al. [10] have proposed map-restricted bounded natural functors (MRBNFs) as a new modular foundation for binding-aware datatypes that overcomes the above limitations. MRBNFs generalize bounded natural functors (BNFs) [41], which underly Isabelle's datatypes and codatatypes [11]. In this paper, we present the journey from the theoretical MRBNF framework to a practical package in Isabelle/HOL, called MrBNF (pronounced "Mister BNF").

MrBNF's heart is the `binder_datatype` command for declaring binding-aware datatypes. Behind the scenes, the command composes and takes least fixed points of MRBNFs, defines the new type as the quotient of a raw nameful datatype by alpha-equivalence, lifts the raw constructors to the quotient, and proves nominal induction principles as well as a wealth of constructor properties (§3). The command also provides a nominal recursor infrastructure, which is crucial for defining recursive functions. All constructions are carried out foundationally in Isabelle/HOL: no axioms have been introduced (§4). Our main contributions are twofold:

- We extend Isabelle/HOL with a foundational package for defining binding-aware datatypes that supports nested recursion, complex inductive binders, and types that may have infinitely many free or bound variables. This involves proving that MRBNFs, the key notion underlying our approach, are closed under composition and least fixed points. We design and automate these mechanized proofs as Isabelle/ML tactics. Our implementation includes a user-friendly proof method for applying the nominal induction principles and a nominal recursor for defining binding-aware primitive recursive functions on datatypes with binders.

- Two case studies illustrate our tool's usefulness. First, we prove Mazza's result [22] that the λ-calculus is isomorphic to an infinitary affine λ-calculus (§5). Second, we formalize the POPLmark challenge [4], i.e., type soundness of System $\mathrm{F}_{<:}$, including parts 1B and 2B, which extend the language with records and pattern matching (§6). To the best of our knowledge, this is the first formalization of these extensions using a nominal approach.

Our implementation and case studies are publicly available [44].

2 Related Work

We refer to Blanchette et al. [10, Section 9] for a broad overview of syntax with bindings approaches in programming languages and proof assistants. Here, we focus our attention on how these approaches manifest themselves in proof assistants and discuss strength and weaknesses.

The representation of variables as de Bruijn indices [17] is widely popular in proof assistants [8,16,20,25,35,37,40,45] because it is readily available via standard datatypes. Thereby bound variables point to the respective binders using a simple indexing scheme: a number indicates how many binders to skip when traversing the syntax tree towards its root. Binders such as λ-abstractions do not need to mention the bound variable. Free variables are numbers, too, namely those larger than the number of binders above them. For example, $\mathsf{Lm}\ (\mathsf{Lm}\ (\mathsf{Ap}\ (\mathsf{Ap}\ 1\ 0)\ 2))$ is the de Bruijn version of the λ-calculus term $\lambda x.\ \lambda y.\ ((y\ x)\ z)$. Working with indices frequently requires shifting when a term is moved under a binder, e.g., during substitution. Good automation as provided by Autosubst in Coq [36,39] can eliminate much of the tedium of index shifting. Nonetheless the internal representation occasionally leaks: index shifts may pop up in induction proofs and sometimes even lemma statements.

The related locally nameless representation [15,33] combines de Bruijn indices for bound variables with the named representation for free variables, which allows to use readable names for the free variables. Locally nameless replaces shifting by opening terms such that bound variables are turned into free ones. One downside of the locally nameless approach is that terms with loose bounds are malformed and need to be ruled out using a predicate (or a subtype).

Nominal Logic [18] provides a nameful alternative to the two above approaches: binders carry explicit bound variable names, but the syntax is quotiented by a notion of alpha-equivalence which makes the name choice immaterial. Still the explicit mention of the bound meta-variable in the binders allows us to refer to it explicitly and choose it to avoid other surrounding variables, which enforces Barendregt's variable convention. Nominal Isabelle is the Isabelle/HOL implementation of nominal logic [21,42], which has been used in several substantial formalizations efforts [7,13,14,29]. Our contribution follows the nominal approach, while generalizing the support for nested recursion and allowing infinitely many variables in terms.

Higher-order abstract syntax (HOAS) [30] uses binding primitives available in the meta-logic to represent binders of the object of study. For example, abstraction in the λ-calculus becomes $\mathsf{Lm} : (\mathit{var} \to \mathit{term}) \to \mathit{term}$ under weak HOAS and $\mathsf{Lm} : (\mathit{term} \to \mathit{term}) \to \mathit{term}$ under HOAS, reusing the λ-abstraction available in the language when writing specific lambda terms, e.g., $\mathsf{Lm}\ (\lambda x.\ \mathsf{Lm}\ (\lambda y.\ \mathsf{Ap}\ (\mathsf{Ap}\ y\ x)\ (\mathsf{Vr}\ z)))$. A challenge with (weak) HOAS are so-called exotic terms, i.e., terms that do not constitute valid λ-calculus terms because they observe aspects of the meta-language that the object language should not see. HOAS is popular in logical frameworks pioneered by Twelf [31] and refined and extended in Beluga [32] and Abella [5].

Berghofer and Urban [9] provide a detailed comparison between the de Bruijn and nominal approaches; Momigliano et al. [24] perform a similar exercise for de Bruijn and (weak) HOAS. Ambal et al. [2] compare all above approaches in the context of a higher-order π-calculus. Norrish and Vestergaard [27] establish a formal connection between de Bruijn and nominal terms.

Solutions using different above techniques [1] target the POPLmark challenge [4]. However, only four cover all proof-related parts, in particular including complex binders for linear pattern matching: three using de Bruijn indices in Isabelle [8] and Coq [38,45] and one using HOAS in Twelf [3]. We provide the first complete solution following the nominal approach.

3 MrBNF in Action

As users, what do we want to be the effect of specifying a datatype with bindings, such as those of λ- or π-calculus syntax? We want the following: (1) a type capturing the syntax fully abstractly, i.e., not distinguishing between alpha-equivalent terms and not including "junk", i.e., invalid terms; (2) constants corresponding to the syntactic constructors and other syntactic operators such as renaming and free-variables; (3) propositions describing the basic properties of non-binding constructors), and quasi-injectivity for the binder constructors; (4) propositions describing the basic properties concerning the interaction of constructors and the renaming and free-variable operators; (5) a proposition stating a binding-aware structural induction principle; and (6) a proposition stating the characteristic equations of a binding-aware structural recursion principle.

Importantly, we would not care how such a type and constants have been defined internally, because (a subset of) the above properties *characterize the type uniquely up to an isomorphism.* This ensures that these internal definitions, however they proceed, give us the correct result.

In the remainder of this section, using a sequence of increasingly sophisticated syntaxes with bindings we will illustrate how our MrBNF definitional package achieves these goals.

3.1 Preliminaries on cardinals and permutations

Isabelle has a well-developed theory of ordinals and cardinals [12]. In a nutshell: an ordinal is just a well-order, while a cardinal is an ordinal that is minimal under the preorder relation $\leq_o$ on ordinals defined as follows: $r \leq_o r'$ iff there exists a well-order embedding between r and r'; we also write $<_o$ for the strict counterpart of this preorder, and also $=_o$ for its induced equivalence relation. Given any set $A : {'a}\ \mathit{set}$, we define $|A|$ to be its cardinality; technically, this is a (necessarily unique up to an order isomorphism) choice of a cardinal on $'a$ whose domain is A and that forms a well-order on A. Instead of $|\mathsf{UNIV} : {'a}\ \mathit{set}|$, the cardinality of the set of all elements of type $'a$, we will simply write $|'a|$, and refer to it as the cardinality of the type $'a$.

For a function $\sigma : {'a} \to {'a}$, we write $\mathsf{supp}\ \sigma$ for its *support*, defined as the set of elements that σ modifies, $\{x \mid \sigma\ x \neq x\}$. We call *permutation* any such function that (1) is bijective and (2) has the cardinality of its support strictly smaller than that of its underlying type. Formally, the (polymorphic) predicate $\mathsf{perm} : ({'a} \to {'a}) \to \mathit{bool}$ reflects this as $\mathsf{perm}\ \sigma \longleftrightarrow \mathsf{bij}\ \sigma\ \wedge\ |\mathsf{supp}\ \sigma| <_o |'a|$. When $'a$ is countably infinite, being a permutation amounts to being a bijection of finite support, so this generalizes the standard nominal logic assumption. We let σ range over permutations and write σ^{-1} for the inverse of σ. We let $a \leftrightarrow b$ denote the swapping permutation, which takes a to b, takes b to a, and leaves everything else unchanged.

3.2 λ-calculus terms

Let us start with the paradigmatic example of syntax with bindings, that of untyped λ-calculus. Using our package, this can be declared as the following datatype lterm of λ-terms, which is polymorphic in the type of variables, i.e., depends on the Isabelle type-variable $'var$:

```
binder_datatype 'var lterm = Vr 'var | Ap "'var lterm" "'var lterm"
  | Lm x::'var t::"'var lterm" binds x in t
```

When using the type $'var\ \mathit{lterm}$, we will always implicitly assume that $'var$ has at least countably infinite cardinality. (This is achieved in practice via a type class $\mathsf{large}_{\mathit{lterm}}$, i.e., being "large enough", which means having cardinality at least as large as $\mathsf{bound}_{\mathit{lterm}}$, and $\mathsf{bound}_{\mathit{lterm}}$ is a cardinal bound specific to each datatype – here, for lterm, it is a countable cardinal, i.e., $\mathsf{bound}_{\mathit{lterm}} = \aleph_0$, so smallness means "at least countably infinite" – see §4 for more details.)

The command produces the following constants, all polymorphic in $'var$:

- the constructors $\mathsf{Vr} : {'var} \to {'var}\ \mathit{lterm}$, $\mathsf{Ap} : {'var}\ \mathit{lterm} \to {'var}\ \mathit{lterm} \to {'var}\ \mathit{lterm}$ and $\mathsf{Lm} : {'var} \to {'var}\ \mathit{lterm} \to {'var}\ \mathit{lterm}$;
- the free-variable operator $\mathsf{FV}_{\mathit{lterm}} : {'var}\ \mathit{lterm} \to {'var}\ \mathit{set}$;
- the permutation operator $\mathsf{PERM}_{\mathit{lterm}} : ({'var} \to {'var}) \to {'var}\ \mathit{lterm} \to {'var}\ \mathit{lterm}$, where we write $t[\sigma]_{\mathit{lterm}}$ instead of $\mathsf{PERM}_{\mathit{lterm}}\ \sigma\ t$;
- a cardinal bound, $\mathsf{bound}_{\mathit{lterm}}$ (which, as explained above, in this case it is $\aleph_0$);
- a binding-aware recursion combinator

$$\begin{aligned}\mathsf{rec}_{\mathit{lterm}} \quad : \quad & (({'var} \to {'var}) \to ({'p} \to {'p})) \to \quad ({'p} \to {'var}\ \mathit{set}) \to \\ & (({'var} \to {'var}) \to ({'a} \to {'a})) \to \quad ({'a} \to {'var}\ \mathit{set}) \to \\ & ({'var} \to ({'p} \to {'a})) \to \\ & (({'p} \to {'a}) \to ({'p} \to {'a}) \to ({'p} \to {'a})) \to \\ & ({'var} \to ({'p} \to {'a}) \to ({'p} \to {'a})) \to \\ & {'var}\ \mathit{lterm} \to ({'p} \to {'a})\ .\end{aligned}$$

We write FV and $\_[\_]$ instead of $\mathsf{FV}_{\mathit{lterm}}$ and $\_[\_]_{\mathit{lterm}}$ (and similarly for other examples).

The following properties are generated (stated and proved) by our command:

▶ **Prop 1.**

(I) Distinctness and (quasi-)injectivity of the constructors:
(1) $\mathsf{Vr}\ x \neq \mathsf{Ap}\ t_1\ t_2$; (2) $\mathsf{Vr}\ x \neq \mathsf{Lm}\ x'\ t$; (3) $\mathsf{Ap}\ t_1\ t_2 \neq \mathsf{Lm}\ x\ t$;
(4) $\mathsf{Vr}\ x = \mathsf{Vr}\ x' \longleftrightarrow x = x'$; (5) $\mathsf{Ap}\ t_1\ t_2 = \mathsf{Ap}\ t_1'\ t_2' \longleftrightarrow t_1 = t_1' \wedge t_2 = t_2'$;
(6) $\mathsf{Lm}\ x\ t = \mathsf{Lm}\ x'\ t' \longleftrightarrow (x' \notin \mathsf{FV}\ t \vee x = x') \wedge t = t'[x \leftrightarrow x']$;

(II) Equivariance of the constructors:
(1) $\mathsf{perm}\ \sigma \longrightarrow (\mathsf{Vr}\ x)[\sigma] = \mathsf{Vr}\,(\sigma\ x)$; (2) $\mathsf{perm}\ \sigma \longrightarrow (\mathsf{Ap}\ t_1\ t_2)[\sigma] = \mathsf{Ap}\ (t_1[\sigma])\ (t_2[\sigma])$;
(3) $\mathsf{perm}\ \sigma \longrightarrow (\mathsf{Lm}\ x\ t)[\sigma] = \mathsf{Lm}\,(\sigma\ x)\ (t[\sigma])$;

(III) Smallness (here, equivalently, finiteness) of the set of free variables:
(1) $|\mathsf{FV}\ t| <_o \mathsf{bound}_{\mathit{lterm}}$;

(IV) Interaction between free variables and constructors:
(1) $\mathsf{FV}\,(\mathsf{Vr}\ x) = \{x\}$; (2) $\mathsf{FV}\,(\mathsf{Ap}\ t_1\ t_2) = \mathsf{FV}\ t_1 \cup \mathsf{FV}\ t_2$; (3) $\mathsf{FV}\,(\mathsf{Lm}\ x\ t) = \mathsf{FV}\ t \setminus \{x\}$.

(V) Permutation identity and compositionality:
(1) $t[\mathsf{id}] = t$; (2) $\mathsf{perm}\ \sigma \wedge \mathsf{perm}\ \sigma' \longrightarrow t[\sigma][\sigma'] = t[\sigma' \circ \sigma]$;

(VI) Interaction between free variables and permutation (infix $`$ denotes image):
(1) $\mathsf{perm}\ \sigma \longrightarrow \mathsf{FV}\ (t[\sigma]) = \sigma\ `\ \mathsf{FV}\ t$; (2) $\mathsf{perm}\ \sigma \wedge (\forall x \in \mathsf{FV}\ t.\ \sigma\ x = x) \longrightarrow t[\sigma] = t$.

Note that the constructors Vr and Ap are free, hence injective (points (4) and (5) in the above proposition). On the other hand, the λ-constructor Lm is not free, since it introduces bindings – for example, $\mathsf{Lm}\ x\ (\mathsf{Vr}\ x) = \mathsf{Lm}\ y\ (\mathsf{Vr}\ y)$ for any variables x, y. Therefore, only a quasi-injectivity, i.e., injectivity up to a renaming, property holds for it (point (6)).

Points (I.6), (IV.3) and (VI.2) all reflect the fact that we work not with entirely free terms but with terms quotiented to alpha-equivalence. And so does the following proposition, expressing a strong version of structural induction, which is also generated by the **binder_datatype** command:

▶ **Prop 2** (Binding-aware structural induction)**.** Assume $\mathsf{Pvars} : {}'p \to {}'var\ \mathtt{set}$ and $\varphi : {}'p \to {}'var\ \mathtt{lterm} \to \mathtt{bool}$ are such that (1) $\forall p.\ |\mathsf{Pvars}\ p| <_o \mathsf{bound}_{\mathtt{lterm}}$, i.e., $\forall p.$ finite $(\mathsf{Pvars}\ p)$; (2) $\forall p, x.\ \varphi\ p\ (\mathsf{Vr}\ x)$; (3) $\forall p, t_1, t_2.\ (\forall q.\ \varphi\ q\ t_1) \wedge (\forall q.\ \varphi\ q\ t_2) \longrightarrow \varphi\ p\ (\mathsf{Ap}\ t_1\ t_2)$; and (4) $\forall p.\ \forall x, t.\ x \notin \mathsf{Pvars}\ p \wedge (\forall q.\ \varphi\ q\ t) \longrightarrow \varphi\ p\ (\mathsf{Lm}\ x\ t)$. Then $\forall p, t.\ \varphi\ p\ t$.

The above resembles standard structural induction (as available for the standard datatypes), except for the highlighted part, which allows one to assume during the induction process that the bound variables are disjoint from the variables coming from a designated type $'p$ of parameters – this enables the rigorous application of Barendregt's variable convention [6]. Taking $'p$ to be the unit type and $\mathsf{Pvars}\ p = \varnothing$, we obtain standard structural induction.

Given a type $'a$ together with operators resembling the free-variable and permutation operators, namely $afv : {}'a \to {}'var\ \mathtt{set}$ and $aprm : ({}'var \to {}'var) \to {}'a \to{}' a$, we say that they form a *loosely-supported pre-nominal structure*, written lspnom $afv\ aprm$, when the following holds:

- Compositionality (Prop. 1, V.2): $\mathsf{perm}\,\sigma \wedge \mathsf{perm}\,\sigma' \longrightarrow aprm\,\sigma\,(aprm\,\sigma'\,a) = aprm\,(\sigma \circ \sigma')\,a$.
- Congruence (Prop. 1, VI.2): $\mathsf{perm}\ \sigma \wedge (\forall x \in afv\ a.\ \sigma\ x = x) \longrightarrow aprm\ \sigma\ a = a$.

Moreover, for any cardinal κ, we say that they form a *κ-loosely-supported nominal structure*, written $\mathsf{lsnom}_\kappa\ afv\ aprm$, when lspnom $afv\ aprm$ holds and additionally the following holds:

- Smallness (Prop. 1, III.1 in case $\kappa = \mathsf{bound}_{\mathtt{lterm}}$): $|\mathsf{FV}\ a| <_o \kappa$.

Finally, given two types $'p$ and $'a$ and operators on them

- $pfv : {}'p \to {}'var\ \mathtt{set}$, $pprm : ({}'var \to {}'var) \to {}'p \to {}'p$,
- $afv : {}'a \to {}'var\ \mathtt{set}$, $aprm : ({}'var \to {}'var) \to {}'a \to {}'a$,
- $vr : {}'var \to ({}'p \to {}'a)$, $ap : {}'var\ \mathtt{lterm} \to ({}'p \to {}'a) \to {}'var\ \mathtt{lterm} \to ({}'p \to {}'a) \to ({}'p \to {}'a)$, $lm : {}'var \to {}'var\ \mathtt{lterm} \to ({}'p \to {}'a) \to ({}'p \to {}'a)$

(where vr, ap, lm have types resembling those of $\mathtt{lterm}$'s constructors Vr, Ap and Lm), we say that they form an $\mathtt{lterm}$*-model*, written $\mathsf{model}_{\mathtt{lterm}}\ pfv\ pprm\ afv\ aprm\ vr\ ap\ lm$, provided that: (1) $\mathsf{lsnom}_{\mathsf{bound}_{\mathtt{lterm}}}\ pfv\ pprm$ holds; (2) lspnom $afv\ aprm$ holds; and (3) the following properties, corresponding to properties of $\mathtt{lterm}$, hold, where $paprm\ \sigma\ f = aprm\ \sigma \circ f \circ pprm\ (\sigma^{-1})$:

1. equivariance of the constructors (Prop 1, II.1, II.2, II.3):
   - $\mathsf{perm}\ \sigma \longrightarrow paprm\ \sigma\ (vr\ x) = vr\ (\sigma\ x)$;
   - $\mathsf{perm}\ \sigma \longrightarrow paprm\ \sigma\ (ap\ t_1\ f_1\ t_2\ f_2) = ap\ (t_1[\sigma])\ (paprm\ \sigma\ f_1)\ (t_2[\sigma])\ (paprm\ \sigma\ f_2)$;
   - $\mathsf{perm}\ \sigma \longrightarrow paprm\ \sigma\ (lm\ x\ t\ f) = lm\ (\sigma\ x)\ (t[\sigma])\ (paprm\ \sigma\ f)$;
2. free-variables sub-distributing under constructors (weaker versions of Prop 1, IV.1, IV.2, IV.3, with inclusions instead of equalities):
   - $afv\,(vr\ x\ p) \subseteq \{x\} \cup pfv\ p$;
   - $afv\,(f_1\ p) \subseteq afv\ t_1 \cup pfv\ p\ \wedge\ afv\,(f_2\ p) \subseteq afv\ t_2 \cup pfv\ p \longrightarrow$ $afv\,(ap\ t_1\ f_1\ t_2\ f_2\ p) \subseteq afv\ t_1 \cup afv\ t_2 \cup pfv\ p$;
   - $x \notin pfv\ p \wedge afv\,(f\ p) \subseteq afv\ t \setminus \{x\} \cup pfv\ p \ \longrightarrow\ afv\,(lm\ x\ t\ f\ p) \subseteq afv\ t \setminus \{x\} \cup pfv\ p$.

We refer to the types $'p$ and $'a$ above as the *parameter type* and the *carrier type* of the *lterm*-model, respectively. Our binding-aware recursor operates on *lterm*-models, in that, given any *lterm*-model it returns a function from terms and parameters to carrier elements that (1) commutes with the constructors and permutation operators; and (2) preserves the free-variable operators. Moreover, commutation with the binding constructor happens in a binding-aware fashion, that is, avoiding clashes between the bound variables and the parameter variables – i.e., again obeying Barendregt's variable convention. This is expressed in the following proposition:

▶ **Prop 3** (Binding-aware recursion)**.** Assume $\mathsf{model}_{\mathit{lterm}}\ \mathit{pfv}\ \mathit{pprm}\ \mathit{afv}\ \mathit{aprm}\ \mathit{vr}\ \mathit{ap}\ \mathit{lm}$ holds and let $g : {'var}\ \mathit{lterm} \to {'p} \to {'a}$ denote $\mathsf{rec}_{\mathit{lterm}}\ \mathit{pfv}\ \mathit{pprm}\ \mathit{afv}\ \mathit{aprm}\ \mathit{vr}\ \mathit{ap}\ \mathit{lm}$. The following properties hold: (1) $g\ (\mathsf{Vr}\ x)\ p = \mathit{vr}\ x\ p$; (2) $g\ (\mathsf{Ap}\ t_1\ t_2)\ p = \mathit{ap}\ t_1\ (g\ t_1)\ t_2\ (g\ t_2)\ p$; (3) $x \notin \mathit{pfv}\ p \longrightarrow g\ (\mathsf{Lm}\ x\ t)\ p = \mathit{lm}\ x\ t\ (g\ t)\ p$; (4) $\mathsf{perm}\ \sigma \longrightarrow g\ (a[\sigma])\ p = \mathit{paprm}\ \sigma\ (g\ a)\ p$; and (5) $\mathit{afv}\ (g\ t\ p) \subseteq \mathsf{FV}\ t \cup \mathit{pfv}\ p$.

In the current implementation, we do not get a single recursor constant and the above recursion theorem, but rather given a model we define g and derive its properties on the fly. Our recursor definition follows Blanchette et al.'s design [10], which generalizes Norrish's nominal recursor [26] and removes one of the unnecessary assumptions [34].

Here is an example of applying the recursor. For any $\rho : {'var} \to {'var}\ \mathit{lterm}$, we let its *support* $\mathsf{Supp}\ \rho$ be $\{x : {'var} \mid \rho\ x \neq \mathsf{Vr}\ x\}$, and its *image-support* $\mathsf{ImSupp}\ \rho$ be $\mathsf{Supp}\ \rho \cup \bigcup_{t \in \mathsf{Supp}\ \rho} \mathsf{FV}\ t$. We let the type of substitution-functions $'var\ \mathsf{substFun}$ be the type of all functions ρ such that $|\mathsf{Supp}\ \rho| <_o \mathsf{bound}_{\mathit{lterm}}$ (obtained as a subtype of $\rho : {'var} \to {'var}\ \mathit{lterm}$); function application and composition are inherited to $'var\ \mathsf{substFun}$ from the function type and are denoted the same. To define term-for-variable substitution operator $\mathsf{subst} : {'var}\ \mathit{lterm} \to {'var}\ \mathsf{substFun} \to {'var}\ \mathit{lterm}$, we take $'p = {'var}\ \mathsf{substFun}$ and $'a = {'var}\ \mathit{lterm}$, and determine the model from the desired recursive clauses for the constructors and the desired behavior of substitution w.r.t. free variables and permutation:
(1) $\mathsf{subst}\ (\mathsf{Vr}\ x)\ \rho = \rho\ x$;
(2) $\mathsf{subst}\ (\mathsf{Ap}\ t_1\ t_2)\ \rho = \mathsf{Ap}\ (\mathsf{subst}\ t_1\ \rho)\ (\mathsf{subst}\ t_2\ \rho)$;
(3) $x \notin \mathsf{ImSupp}\ \rho \longrightarrow \mathsf{subst}\ (\mathsf{Lm}\ x\ t)\ \rho = \mathsf{Lm}\ x\ (\mathsf{subst}\ t\ \rho)$;
(4) $\mathsf{subst}\ (t[\sigma])\ \rho = \mathsf{subst}\ t\ ((\_[\sigma]) \circ \rho)$;
(5) $\mathsf{FV}\ (\mathsf{subst}\ t\ \rho) \subseteq \mathsf{FV}\ t \cup \mathsf{ImSupp}\ \rho$.
Namely, here is the *lterm*-model structure $(\mathit{pfv}, \mathit{pprm}, \mathit{afv}, \mathit{aprm}, \mathit{vr}, \mathit{ap}, \mathit{lm})$ corresponding to (and unambiguously determined from) the above:
(M1) $\mathit{vr}\ x\ \rho = \rho\ x$;
(M2) $\mathit{ap}\ t_1\ f_1\ t_2\ f_2\ \rho = \mathsf{Ap}\ (f_1\ \rho)\ (f_2\ \rho)$;
(M3) $\mathit{lm}\ x\ t\ f\ \rho = \mathsf{Lm}\ x\ (f\ \rho)$;
(M4) $\mathit{aprm}\ t\ \sigma = t[\sigma]$ and $\mathit{pprm}\ \rho\ \sigma = (\_[\sigma]) \circ \rho$;
(M5) $\mathit{afv}\ t = \mathsf{FV}\ t$ and $\mathit{pfv}\ \rho = \mathsf{ImSupp}\ \rho$.
Indeed, the (Mi) definitions are obtained by "fishing" the codomain operator behind the (i) recursive clause – e.g., (M2) turns (2) into $\mathsf{subst}\ (\mathsf{Ap}\ t_1\ t_2)\ \rho = \mathit{ap}\ t_1\ (\mathsf{subst}\ \rho\ t_1)\ t_2\ (\mathsf{subst}\ \rho\ t_2)\ \rho$. Currently this fishing process is not implemented in our package, so the user has to explicitly indicate these operators and then infer (1)–(5) from the recursion theorem.

3.3 Infinitary λ-calculus terms

Let $'a$ *stream* and $'a$ *dstream* be the polymorphic types of streams (i.e., countable sequences) and distinct (i.e., non-repetitive) streams, respectively. While streams exist in Isabelle's standard library, we introduce distinct streams as a subtype of streams that ensures that stream elements do not repeat. To simplify the exposition, we pretend that making a type non-repetitive (or linear) is performed automatically using the following command, while for now we are executing manually a uniform construction sketched by Blanchette et al. [10, §4].

$$\textbf{linear\_type}\ \ 'a\ \mathit{dstream} = {}'a\ \mathit{stream}\ \textbf{on}\ 'a$$

The type of infinitary λ-terms [22], where λ-abstraction binds a distinct stream of variables and application applies a term to a stream of terms, is introduced by the following command:

```
binder_datatype 'var iterm = iVr 'var | iAp "'var iterm" "'var iterm stream"
  | iLm "(xs::'var) dstream" t::"'var iterm" binds xs in t
```

This time (employing the same type-class mechanism explained in §3.2) when using the type $'var\ \mathit{iterm}$ we will implicitly assume that $'var$ has cardinality at least $\aleph_1$, i.e., is more than countable. Indeed, to accommodate the countable branching syntax while ensuring that no term can exhaust the entire supply of variables, we now have $\mathsf{bound}_{\mathit{iterm}} = \aleph_1$.

Our command produces again the familiar constants: the constructors iVr, iAp and iLm, free-variable operator iFV, permutation operator iPERM (written _[_]), a cardinal bound $\mathsf{bound}_{\mathit{iterm}}$ (here, $\aleph_1$), and a binding-aware recursion combinator $\mathsf{rec}_{\mathit{iterm}}$. Moreover, it generates similar properties as for lterm. We only show properties that differ in a major way from the lterm case (while keeping the numbering). We use an auxiliary predicate for a function that behaves as identity on a given set: $\mathsf{id\_on}\ A\ f = \forall x \in A.\ f\ x = x$.

▶ **Prop 4.**

(I) Distinctness and (quasi-)injectivity of the constructors: (6) $\mathsf{iLm}\ xs\ t = \mathsf{iLm}\ xs'\ t' \longleftrightarrow (\exists\sigma.\ \mathsf{perm}\ \sigma \wedge \mathsf{id\_on}\ (\mathsf{iFV}\ t \smallsetminus \mathsf{dsset}\ xs)\ \sigma \wedge \mathsf{dsmap}\ \sigma\ xs = xs' \wedge t[\sigma] = t')$;

(II) Equivariance of the constructors:
(2) $\mathsf{perm}\ \sigma \longrightarrow (\mathsf{iAp}\ t\ ts)[\sigma] = \mathsf{iAp}\ (t[\sigma])\ (\mathsf{smap}\ (\lambda t'.\ t'[\sigma])\ ts)$;
(3) $\mathsf{perm}\ \sigma \longrightarrow (\mathsf{iLm}\ xs\ t)[\sigma] = \mathsf{iLm}\ (\mathsf{dsmap}\ \sigma\ xs)\ (t[\sigma])$;

(III) Smallness (here, equivalently, at most countability) of the set of free variables:
(1) $|\mathsf{iFV}\ t| <_o \mathsf{bound}_{\mathit{iterm}}$;

(IV) Interaction between free variables and constructors:
(2) $\mathsf{iFV}\ (\mathsf{iAp}\ t\ ts) = \mathsf{iFV}\ t \cup \bigcup_{t' \in \mathsf{sset}\ ts} \mathsf{iFV}\ t'$; (3) $\mathsf{iFV}\ (\mathsf{iLm}\ xs\ t) = \mathsf{iFV}\ t \smallsetminus \mathsf{dsset}\ xs$;

(V) Permutation identity and compositionality;

(VI) Interaction between free variables and permutation.

Again, iVr and iAp are free constructors, hence injective, whereas the binding constructor iLm only satisfies quasi-injectivity, i.e., injectivity up to a permutation of the bound variables which leaves the term's free variables untouched (I.6) – note that the latter property uses the $\mathit{dstream}$-specific free variables (dsset) and permutation operators (dsmap). Similarly the recursive occurrences of iterm nested under stream in the iAp constructor are accessed via the stream's smap and sset functions in II.2 and IV.2, respectively. We obtain binding-aware structural induction and recursion principles, too, and highlight the main differences to Props. 2 and 3 (for a corresponding notion of iterm-model):

▶ **Prop 5** (Binding-aware structural induction)**.** Assume $\mathsf{Pvars} : {}'p \to {}'var\ \mathit{set}$ and $\varphi : {}'p \to {}'var\ \mathit{iterm} \to \mathit{bool}$ are such that (1) $\forall p.\ |\mathsf{Pvars}\ p| <_o \mathsf{bound}_{\mathit{iterm}}$, i.e., $\forall p.\ \mathsf{countable}\ (\mathsf{Pvars}\ p)$; (2) $\forall p, x.\ \varphi\ p\ (\mathsf{iVr}\ x)$; (3) $\forall p, t, ts.\ (\forall q.\ \varphi\ q\ t) \wedge (\forall t' \in \mathsf{sset}\ ts.\ \forall q.\ \varphi\ q\ t') \longrightarrow \varphi\ p\ (\mathsf{iAp}\ t\ ts)$; and (4) $\forall p, xs, t.\ \mathsf{dsset}\ xs \cap \mathsf{Pvars}\ p = \varnothing \wedge (\forall q.\ \varphi\ q\ t) \longrightarrow \varphi\ p\ (\mathsf{iLm}\ xs\ t)$. Then $\forall p, t.\ \varphi\ p\ t$.

▶ **Prop 6** (Binding-aware recursion). Assume $\mathsf{model}_{\mathit{iterm}}\ \mathit{pfv}\ \mathit{pprm}\ \mathit{afv}\ \mathit{aprm}\ \mathit{ivr}\ \mathit{iap}\ \mathit{ilm}$ holds, and let $g : {}'\mathit{var}\ \mathit{iterm} \to {}'p \to {}'a$ denote $\mathsf{rec}_{\mathit{iterm}}\ \mathit{pfv}\ \mathit{pprm}\ \mathit{afv}\ \mathit{aprm}\ \mathit{ivr}\ \mathit{iap}\ \mathit{ilm}$. Further let $\mathit{paprm}\ \sigma\ f = \mathit{aprm}\ \sigma \circ f \circ \mathit{pprm}\ (\sigma^{-1})$. The following hold: (1) $g\ (\mathsf{iVr}\ x)\ p = \mathit{ivr}\ x\ p$; (2) $g\ (\mathsf{iAp}\ t\ \mathit{ts})\ p = \mathit{iap}\ t\ (g\ t)\ \mathit{ts}\ (\mathsf{smap}\ g\ \mathit{ts})\ p$; (3) $\mathsf{dsset}\ \mathit{xs} \cap \mathit{pfv}\ p = \varnothing \longrightarrow g\ (\mathsf{iLm}\ \mathit{xs}\ t)\ p = \mathit{ilm}\ \mathit{xs}\ t\ (g\ t)\ p$; (4) $\mathsf{perm}\ \sigma \longrightarrow g\ (a[\sigma])\ p = \mathit{paprm}\ \sigma\ (g\ a)\ p$; and (5) $\mathit{afv}\ (g\ t\ p) \subseteq \mathsf{iFV}\ t \cup \mathit{pfv}\ p$.

3.4 Types and terms for System F$_{<:}$

We define the types and terms of System F$_{<:}$, which we will use in our solution to the POPLmark challenge (§6). Because we aim for the challenge 2B, we directly introduce the syntax that incorporates nested types and pattern matching. Compared to the previous subsections we will be much briefer regarding the output of our **binder_datatype** commands: the previous examples already cover many of the arising ingredients and phenomena.

We start by introducing a non-repetitive (in the keys) type of finite sets of key-value pairs that will be used to represent records (where we use strings as keys).

$$\begin{array}{lll}\textbf{type\_synonym} & \mathit{label} & = \mathit{string}\\ \textbf{linear\_type} & ({}'a, {}'b)\ \mathit{lfset} & = ({}'a \times {}'b)\ \mathit{fset}\ \textbf{on}\ {}'a\end{array}$$

The challenge description and all existing solutions favor ordered collections for records, and it would be easy for us to adjust our entire formalization to use lists instead of finite sets (fset). We chose to use fset as the basis for our records because in practical languages like Standard ML or JSON records are considered to be unordered collections. We also chose it because it displays the flexibility of our approach to work with nested recursion through non-datatypes:

```
binder_datatype 'tvar type = TVr 'tvar | Top | Arr "'tvar type" "'tvar type"
  | All X::'tvar "'tvar type" T::"'tvar type" binds X in T
  | TRec "(label, 'tvar type) lfset"
```

The above command defines POPLmark types. The only binding constructor is All and we obtain the following quasi-injectivity property for it (where $\mathsf{TFV} : {}'\mathit{tvar}\ \mathit{type} \to {}'\mathit{tvar}\ \mathit{set}$): $\mathsf{All}\ X\ T_1\ T_2 = \mathsf{All}\ X'\ T_1'\ T_2' \longleftrightarrow (T_1 = T_1' \wedge (X' \notin \mathsf{TFV}\ T_2 \vee X = X') \wedge T_2 = T_2'[X \leftrightarrow X'])$. Naturally, we also obtain binding-aware induction and recursion principles.

We continue with defining terms. For that purpose we introduce patterns as the non-repetitive subtype of the (non-binding) “pre-pattern” datatype that recurses through lfset.

$$\begin{array}{lll}\textbf{datatype} & ({}'\mathit{tvar}, {}'\mathit{var})\ \mathit{ppat} & = \mathsf{PPVr}\ {}'\mathit{var}\ ({}'\mathit{tvar}\ \mathit{type})\ \mid \mathsf{PPRec}\ (\mathit{label}, ({}'\mathit{tvar}, {}'\mathit{var})\ \mathit{ppat})\ \mathit{lfset}\\ \textbf{linear\_type} & ({}'\mathit{tvar}, {}'\mathit{var})\ \mathit{pat} & = ({}'\mathit{var}, {}'\mathit{tvar})\ \mathit{ppat}\ \textbf{on}\ {}'\mathit{var}\end{array}$$

We lift the pre-pattern constructors PPVr and PPRec to the pattern type as $\mathsf{PVr} : {}'\mathit{var} \to {}'\mathit{tvar}\ \mathit{type} \to ({}'\mathit{tvar}, {}'\mathit{var})\ \mathit{pat}$ and $\mathsf{PRec} : (\mathit{label}, ({}'\mathit{tvar}, {}'\mathit{var})\ \mathit{pat})\ \mathit{lfset} \to ({}'\mathit{tvar}, {}'\mathit{var})\ \mathit{pat}$. The latter operator is not a free constructor: its argument must satisfy a non-repetitiveness predicate ($\mathsf{nonrep}_{\mathsf{PRec}} : (\mathit{label}, ({}'\mathit{tvar}, {}'\mathit{var})\ \mathit{pat})\ \mathit{lfset} \to \mathit{bool}$). We are ready to define terms:

```
binder_datatype ('tvar, 'var) term = Vr 'var
| Ap "('tvar, 'var) term" "('tvar, 'var) term"
| Lm x::'var "'tvar typ" t::"('tvar, 'var) term" binds x in t
| ApT "('tvar, 'var) term" "'tvar typ"
| LmT X::'tvar "'tvar typ" t::"('tvar, 'var) term" binds X in t
| Rec "(label, ('tvar, 'var) term) lfset" | Proj "('tvar, 'var) term" label
| Let "('tvar, P::'var) pat" "('tvar, 'var) term" t::"('tvar, 'var) term" binds P in t
```

Of the eight constructors, three are binding. We show their quasi-injectivity properties:

$$
\begin{aligned}
&\mathsf{Lm}\ x\ T\ t = \mathsf{Lm}\ x'\ T'\ t' \longleftrightarrow (T = T' \wedge (x' \notin \mathsf{FV}\ t \vee x = x') \wedge t = t'[x \leftrightarrow x'])\\
&\mathsf{LmT}\ X\ T\ t = \mathsf{LmT}\ X'\ T'\ t' \longleftrightarrow (T = T' \wedge (X' \notin \mathsf{FTV}\ t \vee x = x') \wedge t = t'[X \leftrightarrow X'])\\
&\mathsf{Let}\ P\ t_1\ t_2 = \mathsf{Let}\ P'\ t_1'\ t_2' \longleftrightarrow (t_1 = t_1' \wedge\\
&\quad (\exists \sigma.\ \mathsf{perm}\ \sigma \wedge \mathsf{id\_on}\ (\mathsf{FV}\ t \setminus \mathsf{PV}\ P)\ \sigma \wedge P[\sigma] = P' \wedge t_2[\sigma] = t_2'))
\end{aligned}
$$

These hinge on our ability to refer to a term's free variables (FV) and its free type variables (FTV), as well as a pattern's free type variables (PV). Similarly, we obtain and make use of infrastructure to permute a pattern's variables, a term's variables, and a term's type variables. Again, we also obtain binding-aware induction and recursion principles, where e.g., the parameter p avoids a pattern P's free variables in the Let case of the induction by providing the assumption $\mathsf{PVars}\ p \cap \mathsf{PV}\ P = \varnothing$ to the user.

4 MrBNF's Internals: Construction of Datatypes with Bindings

Isabelle's definitional package for standard (co)datatypes [12], sometimes referred to as the BNF package, is based on bounded natural functors (BNFs) [41] – which are comprised of meta-information associated with well-behaved type constructors and are closed under composition and fixpoints (datatype and codatatype construction). The meta-information consists of a few constants and relations between them. Specifically, a BNF is an n-ary type constructor $\overline{\alpha}\ T$ (here we use the overlined notation as a shorthand for $(\alpha_1, \ldots, \alpha_n)\ T$ and similarly in the following for other types and terms) along with a mapper $\mathsf{map}_T : \overline{(\alpha \to \beta)} \to \overline{\alpha}\ T \to \overline{\beta}\ T$, the relator $\mathsf{rel}_T : \overline{(\alpha \to \beta \to \mathit{bool})} \to \overline{\alpha}\ T \to \overline{\beta}\ T \to \mathit{bool}$, several setters $\mathsf{set}_T^i : \overline{\alpha}\ T \to \alpha_i\ \mathit{set}$, and the cardinal bound bound_T satisfying a number of properties, e.g., $\mathsf{map}_T\ \overline{\mathsf{id}} = \mathsf{id}$ or $\mathsf{set}_T^i\ (\mathsf{map}\ \overline{f}\ x) = f_i\ `\ \mathsf{set}_T^i\ x$. For example, standard lists form a BNF with the standard map function, the relator $\mathsf{list\_all2}$, which relates two lists of the same length provided that lists' elements satisfy the given relation when zipped pair-wise, the set function that returns all the list's elements and the cardinality bound $\aleph_0$. Standard datatypes are least fixpoints of BNFs.

The BNF properties require map_T to behave well when applied to arbitrary functions. Blanchette et al. [10] observed that this is too restrictive when dealing with syntax with bindings and generalized BNFs to map-restricted bounded natural functions (MRBNFs). MRBNFs thus resemble BNFs but distinguish between three *modes* of type arguments: *bound* variables which can be mapped by permutations, *free* variables which can be mapped by small-support (endo)functions, and *live* variables which correspond to BNF's variables and can be mapped by arbitrary functions. (Both BNFs and MRBNFs also support variables that are ignored by map_T, which are called *dead*.) We write $(\overline{\alpha}, \overline{\beta})\ T$ for the MRBNF with $m = |\overline{\alpha}|$ free and bound variables and $|\overline{\beta}|$ live variables (dead variables are left implicit) with the mapper $\mathsf{map}_T : \overline{(\alpha \to \alpha)} \to \overline{(\beta \to \gamma)} \to (\overline{\alpha}, \overline{\beta})\ T \to (\overline{\alpha}, \overline{\gamma})\ T$, the relator $\mathsf{rel}_T : \overline{(\alpha \to \alpha)} \to \overline{(\beta \to \gamma \to \mathit{bool})} \to (\overline{\alpha}, \overline{\beta})\ T \to (\overline{\alpha}, \overline{\gamma})\ T \to \mathit{bool}$, several setters $\mathsf{set}_T^i : (\overline{\alpha}, \overline{\beta})\ T \to \gamma_i\ \mathit{set}$ where $\gamma_i = \alpha_i$ if $i < m$ and $\gamma_i = \beta_{i-m}$ otherwise, and the cardinal bound bound_T. Note that rel_T only acts on the bound and free variables using permutations or small-support functions, respectively. The MRBNF properties clarify which of the $\overline{\alpha}$ are bound or free. For example, distinct streams $'a\ \mathit{dstream}$ are an MRBNF with bound variable $'a$. Our MrBNF package implements Blanchette et al.'s [10] construction of binding-aware datatypes as least fixpoints of MRBNF type equations, and also customizes it to the high-level operators and theorems required by the users. In this section, we describe the construction's main milestones.

4.1 From User-Specifications to Fixpoint Equations

The **binder_datatype** command's syntax is inspired by Isabelle's standard **datatype** command. Bound and free variables in binder datatypes are always left polymorphic. A custom name can be provided for the free variable operators. A type class on the type argument ensures that the type chosen is large enough for the size of the binder datatype, e.g., that it is at least countably infinite for finite syntax like *lterm* and at least uncountably infinite for *iterm*. The "at least" makes nesting binder datatypes in other potentially larger (e.g. uncountable) types easier as the variable type can be increased to match the size of the surrounding type.

The other major addition to the command's syntax compared with standard datatypes are the binding annotations (inspired by Nominal Isabelle) and the subterm selectors. Normal datatypes allow to automatically define accessor functions using the `fun_name::type` syntax. For binder datatypes this syntax is repurposed and generalized to define the binding structure. A selector can not only appear on the top level (i.e. on a field of a constructor as in the Lm constructor in *term*) but also nested within other types (as in the Let constructor in *term*). Valid targets for the selectors are variable positions and (potentially mutually) recursive positions.

MrBNF translates the user specification into the *pre-datatype*, a non-recursive sum of products. Thereby, variable and recursive positions are separated based on whether they appear in a binding clause. Next to the free variables visible in the syntax (`'var`), the pre-datatype also has a bound variable position, and two recursive positions for recursive occurrences under a binder and not under a binder respectively. The *iterm* type's pre-datatype is:

```
type_synonym ('var, 'bvar, 'rec, 'brec) pre_iterm = 'var (* free occurrence *)
  + ('rec * 'rec stream) (* recursive non-binding occurrences *)
  + ('bbar dstream * 'brec) (* bound and recursive bound occurrence *)
```

Next, MrBNF defines a "raw" standard datatype with a single constructor (which is completely free, i.e., not yet quotiented to α-equivalence):

```
datatype 'var raw_iterm =
  ctor_raw_iterm "('var, 'var, 'var raw_iterm, 'var raw_iterm) pre_iterm"
```

For the above step as well as to prepare for the next steps in the construction of the binder datatype, the pre-datatype must form an MRBNF with free $'var$ (setter $\mathsf{set}^1_{\texttt{pre\_iterm}}$), bound $'bvar$ (setter $\mathsf{set}^2_{\texttt{pre\_iterm}}$), and live $'rec$ (setter $\mathsf{set}^3_{\texttt{pre\_iterm}}$) and $'brec$ (setter $\mathsf{set}^4_{\texttt{pre\_iterm}}$) positions – this is ensured by tracking the registered MRBNFs and automating their composition.

4.2 Composition of MRBNFs

The proof that a given type forms an MRBNF proceeds recursively via composition. For the individual components there are three cases to consider. If the type is a type-variable then return the *identity MRBNF* (live $'a$, $T_{ID} := 'a$, $\mathsf{map}_{ID} := \mathsf{id} : ('a \to 'a) \to 'a \to 'a$, $\mathsf{rel}_{ID} := \mathsf{id} : ('a \to 'a \to \mathit{bool}) \to 'a \to 'a \to \mathit{bool}$, and $\mathsf{set}_{ID} := \lambda x.\ \{x\} :\to 'a \to 'a\ \mathit{set}$). If the topmost type constructor is not known to be a (MR)BNF then return the *constant MRBNF* (dead $'d$, $T_{CST} := 'd$, $\mathsf{map}_{CST} := \mathsf{id} : 'd \to 'd$, $\mathsf{rel}_{CST} := (=) : 'd \to 'd \to \mathit{bool}$). Otherwise (i.e., the topmost type constructor *is* a MRBNF) recursively prove that its arguments are MRBNFs. Then do a composition step between the outer MRBNF and the inner MRBNFs.

Inspired by BNF composition [12], MRBNF composition is split into several phases. First step is *demoting*. All type-variables shared between the involved MRBNFs are demoted to the same mode. Given that modes can only become more specific (live > free > bound > dead), this will result in the lowest mode a variable is used at in any of the MRBNFs. If a type-variable appears under a type constructor that is not a (MR)BNF, it must be demoted to dead (using the constant MRBNF). The second step is *lifting*: new dummy type-variables are added to all MRBNFs to ensure that all involved MRBNFs have the same (modulo reordering) bound and free type-variables and all the inner MRBNFs have the same live type-variables (again modulo reordering). The third step is *permuting*: bring shared type variables into the same order in all involved MRBNFs. Finally, composition proceeds along the following definition:

▶ **Definition 1.** Given an outer MRBNF $(\overline{\alpha}, \overline{\beta})\ G$ where $|\overline{\alpha}| = m$ and $|\overline{\beta}| = n$ and inner MRBNFs $(\overline{\alpha}, \overline{\gamma})\ F_1 \ldots (\overline{\alpha}, \overline{\gamma})\ F_n$ where $|\overline{\gamma}| = k$, the composed MRBNF $(\overline{\alpha}, \overline{\gamma})\ H$ is given by:

$$\begin{array}{lcl}
(\overline{\alpha}, \overline{\gamma})\ \mathrm{T_H} & = & (\overline{\alpha}, (\overline{\alpha}, \overline{\gamma})\ F_1, \ldots, (\overline{\alpha}, \overline{\gamma})\ F_n)\ G \\
\mathsf{map}_{\mathrm{H}} & = & \lambda \overline{f}\ \overline{g}.\ \mathsf{map}_G\ \overline{f}\ (\mathsf{map}_{F_1}\ \overline{f}\ \overline{g}) \ldots (\mathsf{map}_{F_n}\ \overline{f}\ \overline{g}) \\
 & & \quad : \overline{(\alpha \to \alpha)} \to \overline{(\gamma \to \gamma')} \to (\overline{\alpha}, \overline{\gamma})\ T_H \to (\overline{\alpha}, \overline{\gamma'})\ T_H \\
\mathsf{set}_{\mathrm{H}}^{i \le m} & = & \lambda x.\ \mathsf{set}_G^i\ x \cup \bigcup \big(y \in \mathsf{set}_G^{m+1}\ x.\ \mathsf{set}_{F_1}^i\ y\big) \cup \cdots \cup \bigcup \big(y \in \mathsf{set}_G^{m+n}.\ \mathsf{set}_{F_n}^i\ y\big) \\
 & & \quad : (\overline{\alpha}, \overline{\gamma})\ T_H \to \alpha_i\ \mathsf{set} \\
\mathsf{set}_{\mathrm{H}}^{i > m} & = & \lambda x.\ \bigcup \big(y \in \mathsf{set}_G^{m+1}\ x.\ \mathsf{set}_{F_1}^i\ y\big) \cup \cdots \cup \bigcup \big(y \in \mathsf{set}_G^{m+n}.\ \mathsf{set}_{F_n}^i\ y\big) \\
 & & \quad : (\overline{\alpha}, \overline{\gamma})\ T_H \to \gamma_{i-m}\ \mathsf{set} \\
\mathsf{rel}_{\mathrm{H}} & = & \lambda \overline{f}\ \overline{R}.\ \mathsf{rel}_G\ \overline{f}\ (\mathsf{rel}_{F_1}\ \overline{f}\ \overline{R}) \ldots (\mathsf{rel}_{F_n}\ \overline{f}\ \overline{R}) \\
 & & \quad : \overline{(\alpha \to \alpha)} \to \overline{(\gamma \to \gamma' \to \mathrm{bool})} \to (\overline{\alpha}, \overline{\gamma})\ T_H \to (\overline{\alpha}, \overline{\gamma'})\ T_H \to \mathrm{bool}
\end{array}$$

4.3 Fixpoint and Quotienting Constructions

Next, MrBNF automates Blanchette et al. [10] definition of free variables, permutation and α-equivalence on the raw datatype. Free variables are defined via an inductive predicate free that specifies if a variable x is free in a term t, here shown on our running example *iterm*.

$$\frac{a \in \mathsf{set}^1_{\mathtt{pre\_iterm}}\ x}{\mathsf{free}\ a\ (\mathsf{ctor}_{\mathtt{raw\_iterm}}\ x)}\ \textsc{TopFree} \qquad \frac{z \in \mathsf{set}^3_{\mathtt{pre\_iterm}}\ x \qquad \mathsf{free}\ a\ z}{\mathsf{free}\ a\ (\mathsf{ctor}_{\mathtt{raw\_iterm}}\ x)}\ \textsc{RecFree}$$

$$\frac{z \in \mathsf{set}^4_{\mathtt{pre\_iterm}}\ x \qquad \mathsf{free}\ a\ z \qquad a \notin \mathsf{set}^2_{\mathtt{pre\_iterm}}\ x}{\mathsf{free}\ a\ (\mathsf{ctor}_{\mathtt{raw\_iterm}}\ x)}\ \textsc{RecBound}$$

The $\mathsf{FV}_{\mathtt{raw\_iterm}}$ function is then defined as $\lambda t.\ \{a.\ \mathsf{free}\ a\ x\}$. One also defines a primitive recursive function permute that takes an permutation on the variable position and applies it to all variables (bound and free). This function uses the map function of the pre-datatype: $\mathsf{permute}\ \sigma\ (\mathsf{ctor}_{\mathtt{raw\_iterm}}\ x) = \mathsf{ctor}_{\mathtt{raw\_iterm}}\ (\mathsf{map}_{\mathtt{pre\_iterm}}\ \sigma\ \sigma\ (\mathsf{permute}\ \sigma)\ (\mathsf{permute}\ \sigma)\ x)$. Equipped with these two functions, alpha-equivalence is defined inductively as follows:

$$\frac{\begin{array}{c}\mathsf{perm}\ \sigma \qquad \mathsf{id\_on}\ \Big(\Big(\bigcup_{z \in \mathsf{set}^4_{\mathtt{pre\_iterm}}\ x} \mathsf{FV}_{\mathtt{raw\_iterm}}\ z\Big) \setminus \mathsf{set}^2_{\mathtt{pre\_iterm}}\ x\Big)\ \sigma \\ \mathsf{rel}_{\mathtt{pre\_iterm}}\ \mathsf{id}\ \sigma\ \mathsf{alpha}_{\mathtt{iterm}}\ (\lambda z.\ \mathsf{alpha}_{\mathtt{iterm}}\ (\mathsf{permute}\ \sigma\ z))\ x\ y\end{array}}{\mathsf{alpha}_{\mathtt{iterm}}\ (\mathsf{ctor}_{\mathtt{raw\_iterm}}\ x)\ (\mathsf{ctor}_{\mathtt{raw\_iterm}}\ y)}$$

MrBNF proves $\mathsf{alpha}_{\mathtt{iterm}}$ to be an equivalence relation and uses it to define a quotient of the raw datatype. The constructor, free variable and permutation operators are lifted from the raw datatype to the quotient. The lifted constructor $\mathsf{ctor}_{\mathtt{iterm}}$ is used to define the high-level constructors iVr, iAp, and iLm, and to prove all the high-level theorems illustrated in §3.

MrBNF actually implements a mild generalization of Blanchette et al. [10]'s fixpoint construction, which only allows to bind variables in recursive subterms. However, sometimes it is necessary to bind variables that appear free in another (non-recursive) type occurrence. To solve this issue in the pre-datatype, instead of just having positions for free and bound type-variables, we also introduce a hybrid called *bfree* type-variables. Then, alpha-equivalence ensures that the portion of bfree variables that appear bound is appropriately renamed.

4.4 Recursion

MrBNF also implements Blanchette et al.'s binding-aware recursion principle [10]. To define a recursive function from a binding-aware datatype $\overline{\alpha}\,T$ of free type-variables $\overline{\alpha}$ to some other $\overline{\alpha}$-type $\overline{\alpha}\,U$, one needs: (1) a *parameter structure* consisting of a type $\overline{\alpha}\,P$, a permutation $Pmap : \overline{\alpha \to \alpha} \to \overline{\alpha}\ P \to \overline{\alpha}\ P$ and support operators $PVars_i : \overline{\alpha}\ P \to \alpha_i\ \mathsf{set}$, and (2) a *model* consisting of a type $\overline{\alpha}\,U$, permutation, and support operators similar to the parameter structure as well as an algebra structure encoding the recursive behavior of the all the constructors, $Uctor : \left(\overline{\alpha}, \overline{\overline{\alpha}\,T \times (\overline{\alpha}\,P \to \overline{\alpha}\,U)}\right)\ pre\_T \to \overline{\alpha}\,P \to \overline{\alpha}\,U$, where pre_T is the pre-datatype of T.

We introduce a *precursor* of our recursor on raw terms, which applies *Uctor* recursively while suitably permuting bound variables "out of the way" with regard to the parameter structure. Suitably means that the "out of the way" function f returns a permutation that does not change the frees and makes bounds disjoint from the frees. For *iterm*, suitable is defined as:

$$\begin{aligned}&\mathsf{suitable}_{\mathtt{iterm}}\ f = \forall x\,p.\ \mathsf{perm}\ (f\ x\ p) \land\\ &\quad \mathsf{imsupp}\ (f\ x\ p) \cap \left((\mathsf{FV}_{\mathtt{iterm}}\ (\mathsf{ctor}_{\mathtt{iterm}}\ x) \cup \mathsf{PVars}\ p) \setminus \mathsf{set}^2_{\mathtt{pre\_iterm}}\ x\right) = \varnothing \land\\ &\quad f\ x\ p\ `\ \mathsf{set}^2_{\mathtt{pre\_iterm}}\ x \cap (\mathsf{FV}_{\mathtt{iterm}}\ (\mathsf{ctor}_{\mathtt{iterm}}\ x) \cup \mathsf{PVars}\ p) = \varnothing\end{aligned}$$

Here, $\mathsf{imsupp}\ f = \mathsf{supp}\ f \cup f\ `\ \mathsf{supp}\ f$. As the precursor must permute the bound variables, it is not possible to define it using primitive recursion. Instead, we use well-founded recursion via an auxiliary $\mathsf{subshape}$ relation, which provides the necessary wiggle room:

$$\frac{\mathsf{perm}\ \sigma \qquad \mathsf{alpha}_{\mathtt{iterm}}\ (\mathsf{permute}_{\mathtt{raw\_iterm}}\ \sigma\ y)\ z \qquad z \in \mathsf{set}^3_{\mathtt{pre\_iterm}}\ x \cup \mathsf{set}^4_{\mathtt{pre\_iterm}}\ x}{\mathsf{subshape}\ y\ (\mathsf{ctor}_{\mathtt{raw\_iterm}}\ x)}$$

We obtain the definition of the precursor rec_U for a suitable "out of the way" function f:

$$\begin{aligned}&\mathsf{rec}_U\ f\ (\mathsf{ctor}_{\mathtt{raw\_iterm}}\ x)\ p = \text{if}\ \neg\mathsf{suitable}_{\mathtt{iterm}}\ f\ \text{then}\ \mathsf{undefined}\ \text{else}\ Uctor\\ &\quad (\mathsf{map}_{\mathtt{pre\_iterm}}\ \mathrm{id}\ (f\ x\ p)\ ((\lambda t.\ (t, \mathsf{rec}_U\ f\ t)) \circ \mathsf{permute}_{\mathtt{raw\_iterm}}\ (f\ x\ p))\ (\lambda t.\ (t, \mathsf{rec}_U\ f\ t))\ x)\ p\end{aligned}$$

The main lemma that the package proves is that the precursor commutes with permutation, it returns the same result for alpha-equivalent terms, and that the specific choice of the "out of the way" function is irrelevant. These properties must be proved simultaneously by induction on the binder datatype using the induction scheme associated with the subshape relation:

$$\begin{aligned}&\mathsf{suitable}_{\mathtt{iterm}}\ f \Longrightarrow \mathsf{suitable}_{\mathtt{iterm}}\ f' \Longrightarrow \mathsf{perm}\ \sigma \Longrightarrow \mathsf{alpha}_{\mathtt{iterm}}\ t\ t' \Longrightarrow\\ &\mathsf{rec}_U\ f\ (\mathsf{permute}_{\mathtt{raw\_iterm}}\ \sigma\ t)\ p = Umap\ \sigma\ \left(\mathsf{rec}_U\ f\ t\ (\mathsf{Pmap}\ \sigma^{-1}\ p)\right) \land \mathsf{rec}_U\ f\ t\ p = \mathsf{rec}_U\ f'\ t'\ p\end{aligned}$$

To move towards the binding-aware recursor, we use Hilbert Choice to hide the "out of the way" function by choosing an arbitrary suitable one $\mathsf{rrec}_U = \mathsf{rec}_U\ (\varepsilon f.\ \mathsf{suitable}_{\mathtt{iterm}}\ f)$.

The invariance of the precursor under alpha is needed to lift it from the raw type to the quotient type. The definition of the precursor is used to derive a better simplification rule that hides the permuting function. This rule requires that the top-level bound variables are disjoint from the parameter structure and from the free variables. We then use identity as the "out of the way" function on the top level and an arbitrary suitable function in the recursion.

$$\begin{aligned}&\mathsf{set}^2_{\texttt{pre\_iterm}}\ x \cap \Big(\mathsf{set}^1_{\texttt{pre\_iterm}}\ x \cup (\textstyle\bigcup_{z\in\mathsf{set}^3_{\texttt{pre\_iterm}}\ x}\ \mathsf{FV}_{\texttt{iterm}}\ z) \cup \mathsf{PVars}\ p\Big) = \varnothing \Longrightarrow\\ &\mathsf{rrec}_U\ (\mathsf{ctor}_{\texttt{raw\_iterm}}\ x)\ p = \mathsf{Uctor}\ \big(\mathsf{map}_{\texttt{pre\_iterm}}\ \mathsf{id}\ \mathsf{id}\ (\lambda t.\ (t, \mathsf{rrec}_U\ t))\ (\lambda t.\ (t, \mathsf{rrec}_U\ t))\ x\big)\ p\end{aligned}$$

Relativized Recursion. The above is a slightly simplified version of our recursion facilities. To accommodate situations where the domain of parameters or the target domain for the intended recursion function do not make up the entire types but only certain subsets, MrBNF allows the user to optionally provide predicates $\mathsf{valid}_P : \overline{\alpha}\ P \to \mathsf{bool}$ and $\mathsf{valid}_U : \overline{\alpha}\ U \to \mathsf{bool}$ that restrict these domains – while producing proof obligations that the user-provided parameter-structure and recursor-model operators preserve these predicates, and producing recursor clauses relativized to these predicates. For normal datatypes such a feature would be useless, as there are no proof obligations incurred for recursion. But for binding datatypes this is useful, since organizing the entire type as a parameter structure or a recursor model (so that the proof obligations can be discharged) is often difficult or awkward. An example of leveraging the valid_P flexibility is if the user prefers to define substitution using actual functions subject to the small-support requirement, as opposed to defining a subtype corresponding to this requirement (substFun at the end of §3.2). In §5, we show an example that leverages the valid_U flexibility.

5 Application I: Mazza's Isomorphism

In his work on connecting the meta-theory of λ-calculus with the notion of metric completion, Mazza [22] establishes an isomorphic translation between standard λ-calculus (using the *lterm* syntax in §3.2) and a (uniform affine) infinitary λ-calculus (the *iterm* syntax from §3.3). We show our formalization using MrBNF of some key constructions in his development.

Recall that the type-variable for the *lterm* type constructor must be infinite (since $\mathsf{bound}_{\mathit{lterm}} = \aleph_0$), and the one for *iterm* must be uncountably infinite (since $\mathsf{bound}_{\mathit{iterm}} = \aleph_1$). In what follows, we fix these type-variables, namely fix a countable type *var* (a copy of *nat*) and an uncountable type *ivar*; we will refer to the elements of *ivar* as *ivariables*. We will simply write *lterm* instead of *var lterm* and *iterm* instead of *ivar iterm*.

Following Mazza, we choose a countable set $\mathsf{Spr} : (\mathit{var\ dstream})\ \mathit{set}$ of distinct streams of variables called *supervariables*, having the property that any two are mutually disjoint: $\forall xs, ys \in \mathsf{Spr}.\ \mathsf{sset}\ xs \cap \mathsf{sset}\ ys = \emptyset$. The intention is restricting the λ-iterms to only use these as bindings. Moreover, we choose a function $\mathsf{spr} : \mathit{var} \to (\mathit{var\ dstream})\ \mathit{set}$ for which $\mathsf{bij\_betw}\ \mathsf{spr}\ (\mathsf{UNIV} : \mathit{var\ set})\ \mathsf{Spr}$ holds, i.e., spr is a bijection between variables and supervariables; we write $\mathsf{spr}^{-1} : (\mathit{var\ dstream})\ \mathit{set} \to \mathit{var}$ for its inverse. We refer to the elements of *nat list* as *positions*, and choose a bijection $\mathsf{natOf} : \mathit{nat\ list} \to \mathit{nat}$. For $p : \mathit{nat\ list}$ and $n : \mathit{nat}$, $p \cdot n$ denotes the concatenation of p and $[n]$. According to Mazza's definition, the finitary-to-infinitary translation should be a function $[\![\_]\!]_\_ : \mathit{lterm} \to \mathit{nat\ list} \to \mathit{iterm}$ given by: (1) $[\![\mathsf{Vr}\ x]\!]_p = \mathsf{iVr}\ ((\mathsf{spr}\ x)_{\mathsf{natOf}\ p})$; (2) $[\![\mathsf{Lm}\ x\ t]\!]_p = \mathsf{iLm}\ (\mathsf{spr}\ x)\ [\![t]\!]_p$; and (3) $[\![\mathsf{Ap}\ t_1\ t_2]\!]_p = \mathsf{iAp}\ [\![t_1]\!]_{p\cdot 0}\ ([\![t_2]\!]_{p\cdot 1}, [\![t_2]\!]_{p\cdot 2}, [\![t_2]\!]_{p\cdot 3}, \ldots)$.

The intuition is that every variable x in the original term is duplicated in the translation into countably many ivariable "copies" of it sourced from its corresponding supervariable, $\mathsf{spr}\ x$. The positions make sure that the copies located in different parts of the resulting iterm are distinct, thus ensuring that the iterm is affine. Indeed, in the recursive case for application, we see that the position p grows with different numbers appended to the

$$\frac{xs \in \mathsf{Spr} \quad \{x, x'\} \subseteq \mathsf{sset}\ xs}{\mathsf{iVr}\ x \ \approx\ \mathsf{iVr}\ x'}\ \textsc{iVr} \qquad\qquad \frac{xs \in \mathsf{Spr} \qquad t \approx t'}{\mathsf{iLm}\ xs\ t \ \approx\ \mathsf{iLm}\ xs\ t'}\ \textsc{iLm}$$

$$\frac{t \approx t' \qquad \forall t_1, t_2.\ \{t_1, t_2\} \subseteq \mathsf{sset}\ ts \cup \mathsf{sset}\ ts' \longrightarrow t_1 \approx t_2}{\mathsf{iAp}\ t\ ts \ \approx\ \mathsf{iAp}\ t'\ ts'}\ \textsc{iAp}$$

Figure 1 Renaming equivalence relation.

arguments of infinitary application, which ensures disjointness in conjunction with choosing the particular "copy" based on this position counter ($\mathsf{natOf}\ p$) when reaching the Vr-leaves. Correspondingly, abstraction over a variable is translated to abstraction over its supervariable, i.e., over all its "copies".

Moreover, to describe the image of this translation, Mazza defines the notion of *renaming equivalence* expressed as the relation $\approx$: $\mathit{iterm} \rightarrow \mathit{iterm} \rightarrow \mathit{bool}$ defined inductively in Fig. 1. It relates two λ-iterms t and t' just in case they **(i)** have the same ($\mathsf{iVr}, \mathsf{iLm}, \mathsf{iAp}$)-structure (as trees), **(ii)** only use supervariables in binders, **(iii)** at the leaves have variables appearing in the same supervariable, and **(iv)** for both t and t' all the subterms that form the righthand side of an application are mutually renaming equivalent. Then *uniformity* of an iterm, which will characterize the translation's image, is self-renaming-equivalence: $\mathsf{uniform}\ t = (t \approx t)$.

We aim to define a function satisfying clauses (1)–(3) above, with (2) formally written as $\mathsf{iAp}\ [\![t_1]\!]_{p\cdot 0}\ (\mathsf{smap}\ [\![t_2]\!]_{p\cdot\_}\ (\mathsf{natsFrom}\ 1))$ where, for any n, $\mathsf{natsFrom}\ n$ denotes the stream of naturals starting from n. To turn these clauses into a formal definition, we deploy our recursor for lterm, which requires also indicating the desired interaction between the to-be-defined function with permutation and free-variables. Upon analysis, we converge to (4) $[\![t[\sigma]]\!]_p = [\![t]\!]_p[\mathsf{v2iv}\,\sigma]$ and (5) spr^{-1} ` $\mathsf{touched}\ [\![t]\!]_p \subseteq \mathsf{FV}\ t$, where $\mathsf{v2iv}\,\sigma$ (read "variable to ivariable") converts $\sigma : \mathit{var} \rightarrow \mathit{var}$, via spr, into a supervariable-preserving function; namely, for any $y \in \mathit{ivar}$ such that y appears in some (necessarily unique) supervariable xs, we define $\mathsf{v2iv}\,\sigma\ y : \mathit{ivar}$ as $(\mathsf{spr}\ (\sigma\ (\mathsf{spr}^{-1}\ xs)))_i$ for the unique i such that $xs_i = y$; and $\mathsf{touched}\ t$ is the set of all supervariables that are touched by (the free variables of) t, namely $\{xs \in \mathsf{Spr} \mid \mathsf{sset}\ xs \cap \mathsf{FV}\ t \neq \emptyset\}$.

Equation (4) above is seen to be intuitive if we remember that the translation sends variables to supervariables, which means that bijections σ between variables naturally correspond to bijections between supervariables, hence (thanks to the supervariables being mutually disjoint) to supervariable-structure preserving bijections between ivariables; therefore indeed **(A)** applying a bijection on variables and then translating should be the same as **(B)** first translating and then applying this corresponding bijection of its ivariable "copies" in the translation. As for the above inclusion (5), we obtained it by adjunction from $\mathsf{touched}\ [\![t]\!]_p \subseteq \mathsf{spr}$ ` $\mathsf{FV}\ t$, which is again intuitive if we think in terms of the variable-supervariable correspondence.

Clauses (1)–(5) give us a structure on the intended codomain of $[\![\_]\!]\_$, $\mathit{nat\ list} \rightarrow \mathit{iterm}$, using the recipe sketched at the end of §3.2. However, for these to give us an lterm-model, we must restrict the codomain to include only those functions $f : \mathit{nat\ list} \rightarrow \mathit{iterm}$ whose image consists of renaming-equivalent iterms only – otherwise the model properties do not hold; this is not suprising, since Mazza's translation's goal is to produce uniform iterms. We therefore employ the codomain-relativized recursion discussed at the end of §4.4, obtaining:

▶ **Prop 7.** There exists a unique function $[\![\_]\!]\_$: $\mathit{lterm} \rightarrow \mathit{nat\ list} \rightarrow \mathit{iterm}$ such that clauses (1)–(5) hold, and in addition $\forall p, q.\ [\![t]\!]_p \approx [\![t]\!]_q$; in particular, $\forall p.\ \mathsf{uniform}\ [\![t]\!]_p$.

For the opposite translation $\llparenthesis\_\rrparenthesis$ (from infinitary back to finitary terms), Mazza writes equations that in our notation look as follows, restricting the domain to uniform iterms: (1) $\llparenthesis \mathsf{iVr}\ xs_i \rrparenthesis = \mathsf{Vr}\,(\mathsf{spr}^{-1}\ xs)$; (2) $\llparenthesis \mathsf{iLm}\ xs\ t \rrparenthesis = \mathsf{Lm}\ (\mathsf{spr}^{-1}\ xs)\ \llparenthesis t \rrparenthesis$; (3) $\llparenthesis \mathsf{iAp}\ t\ ts \rrparenthesis = \mathsf{Ap}\ \llparenthesis t \rrparenthesis\ \llparenthesis ts_0 \rrparenthesis$.

With the help of a custom recursor for a suitable superset of the uniform iterms, again adding clauses for permuation and free variables, we are able to prove:

▶ **Prop 8.** There exists a function $\llparenthesis\_\rrparenthesis : \mathtt{iterm} \to \mathtt{lterm}$ satisfying the above clauses (1)–(3) when resticted to uniform iterms (i.e., assuming $\mathsf{iVr}\ xs_i$, $\mathsf{iLm}\ xs\ t$ and $\mathsf{iAp}\ t\ ts$ are uniform), and such that $[\![\_]\!]$'s restriction to uniform iterms is uniquely determined by these properties.

Mazza's main result consists of a sequence of five statements, three of which refer to the syntactic component of the finitary-infinitary isomorphism.

▶ **Prop 9.** The following hold: **(1)** (Lemma 16 from [22]) $t \approx t \longrightarrow \llparenthesis t \rrparenthesis = \llparenthesis t' \rrparenthesis$.
(2) (Thm. 19(1) from [22]) $\llparenthesis [\![s]\!]_p \rrparenthesis = s$. **(3)** (Thm. 19(2) from [22]) $\mathsf{uniform}\ t \longrightarrow [\![\llparenthesis t \rrparenthesis]\!]_p \approx t$.

The theorem states that, for any position p, $[\![\_]\!]$ and $\llparenthesis\_\rrparenthesis_p$ give mutually inverse bijections between terms and equivalence classes of uniform iterms w.r.t. renaming equivalence. An additional lemma (omitted here) shows that the $\approx$-representative produced by $\llparenthesis\_\rrparenthesis_p$ is affine (i.e., has no repeated variables). Thus the result establishes a *syntactic* isomorphism, up to renaming equivalence, between terms and uniform affine iterms. Mazza's isomorphism also has an operational-semantics component, given by a theorem stating that $[\![\_]\!]$ and $\llparenthesis\_\rrparenthesis_p$ preserve β-reduction in both calculi in a manner that matches the number of reduction steps [22, Thm. 19(3,4)]. We omit this result here, but details can be found in our formalization [44].

6 Application II: POPLmark Challenge

We report on our solution to the Part 2 of the POPLmark challenge [4], which is concerned with the type soundness of System $F_{<:}$; our formalization also solves Part 1, which is concerned with subtyping. We work with the System $F_{<:}$ types and terms we have introduced in §3.4. With our setup that enforces the variable convention in all induction proofs, the formalization becomes a routine exercise: we can follow the formalization document and transcribe auxiliary lemmas and their proofs. We show our core definitions of Part 2. Naturally, they rely on some definitions of Part 1 (notably the subtyping relation) and other basic infrastructure (notably contexts modeled as lists); we refer to our formalization for full details [44].

We start with the typing judgments for patterns and terms:

```
inductive pat_typing ("⊢ _ : _ → _" [30,29,30] 30) where
  PTPVr: "⊢ PVr x T : T → ∅ , Inr x <: T"
| PTPRec: "nonrep_PRec PP ⟹ labels PP = labels TT ⟹
  (∀l P T. (l, P) ∈∈ PP ⟶ (l, T) ∈∈ TT ⟶ ⊢ P : T → Δ l) ⟹
  ⊢ PRec PP : TRec TT → concat (map Δ (labelist TT))"

inductive typing ("_ ⊢ _ : _" [30,29,30] 30) where
  TVr: "⊢ Γ OK ⟹ (Inr x, T) ∈ set Γ ⟹ Γ ⊢ Vr x : T"
| TLm: "Γ , Inr x <: T1 ⊢ t : T2 ⟹ Γ ⊢ Lm x T1 t : Arr T1 T2"
| TAp: "Γ ⊢ t1 : Arr T11 T12 ⟹ Γ ⊢ t2 : T11 ⟹ Γ ⊢ App t1 t2 : T12"
| TLmT: "Γ , Inl X <: T1 ⊢ t : T2 ⟹ Γ ⊢ LmT X T1 t : All X T1 T2"
| TApT: "Γ ⊢ t1 : All X T11 T12 ⟹ proj_ctxt Γ ⊢ T2 <: T11 ⟹
   Γ ⊢ ApT t1 T2 : substT (TVr(X := T2)) T12"
| TSub: "Γ ⊢ t : S ⟹ proj_ctxt Γ ⊢ S <: T ⟹ Γ ⊢ t : T"
| TRec: "⊢ Γ OK ⟹ rel_lfset id (λt T. Γ ⊢ t : T) XX TT ⟹ Γ ⊢ Rec XX : TRec TT"
| TProj: "Γ ⊢ t : TRec TT ⟹ (l, T) ∈∈ TT ⟹ Γ ⊢ Proj t l : T"
| TLet: "Γ ⊢ t : T ⟹ ⊢ P : T → Δ ⟹ Γ , Δ ⊢ u : U ⟹ Γ ⊢ Let P t u : U"
```

All rules follow closely the challenge description [4]. A few rules deserve some explanation. Rules `TVr` and `TRec` assume that the context is well-scoped ($\vdash$ Γ `OK`); other rules preserve this invariant inductively. Rule `TRec` uses the relator **`rel_lfset`** to relate the values in two *`lfset`*s pairwise grouped by label. Rule `TApT` uses the parallel substitution function on System $F_{<:}$ types (`substT`), which we define using our recursor. Rule `PTPRec` assumes that the destructed record pattern is nonrepetitive (`nonrep_PRec`); it also writes ∈∈ for membership in `lfset` and constructs the resulting context by sorting the finite set of labels lexicographically (`labelist`).

We next define matching and the evaluation function for the terms.

```
inductive match for σ where
  MPVr: "σ X = v ⟹ match σ (PVr X T) v"
| MPRec: "nonrep_PRec PP ⟹ labels PP ⊆ labels VV ⟹
    (∀l P v. (l, P) ∈∈ PP ⟶ (l, v) ∈∈ VV ⟶ match σ P v) ⟹
    match σ (PRec PP) (Rec VV)"

definition "restrict σ A x = (if x ∈ A then σ x else Vr x)"

inductive step where
  ApLm: "value v ⟹ step (Ap (Lm x T t) v) (subst (Vr(x := v)) TVr t)"
| ApTLmT: "step (ApT (LmT X T t) T2) (subst Vr (TVr(X := T2)) t)"
| LetV: "value v ⟹ match σ P v ⟹ step (Let P v u) (subst (restrict σ (PV p)) TVr u)"
| ProjRec: "∀v ∈ values VV. value v ⟹ (l, v) ∈∈ VV ⟹ step (Proj (Rec VV) l) v"
| ApCong1: "step t t' ⟹ step (Ap t u) (Ap t' u)"
| ApCong2: "value v ⟹ step t t' ⟹ step (Ap v t) (Ap v t')"
| ApTCong: "step t t' ⟹ step (ApT t T) (ApT t' T)"
| ProjCong: "step t t' ⟹ step (Proj t l) (Proj t' l)"
| RecCong: "step t t' ⟹ (l, t) ∈∈ XX ⟹ step (Rec XX) (Rec (XX⟨l := t'⟩))"
| LetCong: "step t t' ⟹ step (Let P t u) (Let P t' u)"
```

Similar to Berghofer's solution [8], we use a matching predicate rather than a (partial) function that computes the matching substitution. The rules `ApLm`, `ApTLmT`, `LetV`, and `ProjRec` implement actual transitions; the remaining rules of `step` are congruence rules navigating to allowed redexes. We prefer this formulation over an equivalent context-based one, because the congruence steps are in all cases the easy cases of the involved induction proofs. The rules `ApLm`, `ApTLmT`, `LetV` use the parallel substitution `subst`, which we again define using our recursor. This substitution function acts both on term variables (first argument) and type variables (second argument). We then prove the main results: progress and preservation.

```
lemma progress: "∅ ⊢ t : T ⟹ value t ∨ (∃t'. step t t')"

lemma preservation: : "Γ ⊢ t : T ⟹ step t t' ⟹ Γ ⊢ t' : T"
```

The proofs are canonical following the challenge description [4]. We pervasively use binding-aware induction on our datatypes but also on the shown inductive predicates, which has recently been developed in Isabelle by van Brügge et al. [43]. Occasionally we use induction even in places where a case distinction would have sufficed: this is because our tool lacks case distinction theorems following the variable convention. One omission in the challenge's pen-and-paper proof, which has been also noted by Berghofer [8], is the following lemma, crucial for progress, about the existence of matching substitutions for well-typed patterns.

```
lemma pat_typing_ex_match: ⊢ P : T → Δ ⟹ ∅ ⊢ v : T ⟹ value v ⟹ ∃σ. match σ P v
```

7 Conclusion

MrBNF is a new definitional package in Isabelle/HOL for defining binding-aware datatypes. It follows a modular approach to datatypes relying on the notion of MRBNF as infrastructure to refer to free, bound, and recursive occurrences in a syntax declaration. It comprises 20 000 lines of Standard ML. While some of its usability edges are still rough, our case studies suggest that MrBNF can be a cornerstone in mechanized developments, pushing the boundaries of nominal techniques. We are currently proceeding to polish MrBNF's rough edges, which involves providing a high-level interface to the recursor, automating the non-repetitiveness construction (**linear_type**), and providing variable-avoiding case distinction rules and a proof method to apply them effectively. At the same time, we are extending MrBNF's scope to binding-aware codatatypes, which will have applications such as Böhm trees [6].

References

1 Submitted solutions to the POPLmark challenge. `https://www.seas.upenn.edu/~plclub/poplmark/`, accessed March 22, 2025, 2005.

2 Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt. HOπ in Coq. *J. Autom. Reason.*, 65(1):75–124, 2021. `doi:10.1007/S10817-020-09553-0`.

3 Michael Ashley-Rollman, Karl Crary, and Robert Harper. Group from CMU's solution to the POPLmark challenge. `https://www.seas.upenn.edu/~plclub/poplmark/cmu.html`, accessed March 22, 2025, 2005.

4 Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMark challenge. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs 2005*, volume 3603 of *LNCS*, pages 50–65. Springer, 2005. `doi:10.1007/11541868_4`.

5 David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *J. Formaliz. Reason.*, 7(2):1–89, 2014. `doi:10.6092/ISSN.1972-5787/4650`.

6 Hendrik Pieter Barendregt. *The lambda calculus – its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

7 Jesper Bengtson and Joachim Parrow. Formalising the pi-calculus using nominal logic. *Log. Methods Comput. Sci.*, 5(2), 2009. URL: `http://arxiv.org/abs/0809.3960`.

8 Stefan Berghofer. A solution to the POPLMark challenge using de Bruijn indices in Isabelle/HOL. *J. Autom. Reason.*, 49(3):303–326, 2012. `doi:10.1007/S10817-011-9231-4`.

9 Stefan Berghofer and Christian Urban. A head-to-head comparison of de Bruijn indices and names. In Alberto Momigliano and Brigitte Pientka, editors, *LFMTP 2006*, volume 174 of *ENTCS*, pages 53–67. Elsevier, 2006. `doi:10.1016/J.ENTCS.2007.01.018`.

10 Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. Bindings as bounded natural functors. *Proc. ACM Program. Lang.*, 3(POPL):22:1–22:34, 2019. `doi:10.1145/3290335`.

11 Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014. `doi:10.1007/978-3-319-08970-6_7`.

12 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Cardinals in Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 111–127. Springer, 2014. `doi:10.1007/978-3-319-08970-6_8`.

13 Joachim Breitner. Formally proving a compiler transformation safe. In Ben Lippmeier, editor, *Haskell Symposium 2015*, pages 35–46. ACM, 2015. `doi:10.1145/2804302.2804312`.

14 Matthias Brun and Dmitriy Traytel. Generic authenticated data structures, formally. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *ITP 2019*, volume 141 of *LIPIcs*, pages 10:1–10:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ITP.2019.10.

15 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi:10.1007/s10817-011-9225-2.

16 Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In Nachum Dershowitz and Andrei Voronkov, editors, *LPAR 2007*, volume 4790 of *LNCS*, pages 211–225. Springer, 2007. doi:10.1007/978-3-540-75560-9_17.

17 N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

18 Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects Comput.*, 13(3-5):341–363, 2002. doi:10.1007/s001650200016.

19 Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. In *LICS 1987*, pages 194–204. IEEE Computer Society, 1987.

20 Gérard P. Huet. Residual theory in lambda-calculus: A formal development. *J. Funct. Program.*, 4(3):371–394, 1994. doi:10.1017/S0956796800001106.

21 Brian Huffman and Christian Urban. A new foundation for Nominal Isabelle. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 35–50. Springer, 2010. doi:10.1007/978-3-642-14052-5_5.

22 Damiano Mazza. An infinitary affine lambda-calculus isomorphic to the full lambda-calculus. In *LICS 2012*, pages 471–480. IEEE Computer Society, 2012. doi:10.1109/LICS.2012.57.

23 Conor McBride and James McKinna. Functional pearl: I am not a number—I am a free variable. In Henrik Nilsson, editor, *Haskell Workshop 2004*, pages 1–9. ACM, 2004. doi:10.1145/1017472.1017477.

24 Alberto Momigliano, S.J. Ambler, and R.L. Crole. A comparison of formalizations of the meta-theory of a language with variable bindings in Isabelle. In *TPHOLs 2001, Supplemental Proceedings*, pages 267–282, 2001. URL: https://www.inf.ed.ac.uk/publications/online/0046/b267.pdf.

25 Tobias Nipkow. More Church-Rosser proofs. *J. Autom. Reason.*, 26(1):51–66, 2001. doi:10.1023/A:1006496715975.

26 Michael Norrish. Recursive function definition for types with binders. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *TPHOLs 2004*, volume 3223 of *LNCS*, pages 241–256. Springer, 2004. doi:10.1007/978-3-540-30142-4_18.

27 Michael Norrish and René Vestergaard. Proof pearl: De Bruijn terms really do work. In Klaus Schneider and Jens Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 207–222. Springer, 2007. doi:10.1007/978-3-540-74591-4_16.

28 Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, 1989. doi:10.1007/BF00248324.

29 Lawrence C. Paulson. A mechanised proof of Gödel's incompleteness theorems using Nominal Isabelle. *J. Autom. Reason.*, 55(1):1–37, 2015. doi:10.1007/s10817-015-9322-8.

30 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *PLDI 1988*, pages 199–208. ACM, 1988. doi:10.1145/53990.54010.

31 Frank Pfenning and Carsten Schürmann. System description: Twelf – A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *CADE 1999*, volume 1632 of *LNCS*, pages 202–206. Springer, 1999. doi:10.1007/3-540-48660-7_14.

32 Brigitte Pientka and Jana Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNCS*, pages 15–21. Springer, 2010. doi:10.1007/978-3-642-14203-1_2.

33 Andrew M. Pitts. Locally nameless sets. *Proc. ACM Program. Lang.*, 7(POPL):488–514, 2023. doi:10.1145/3571210.

34 Andrei Popescu. Nominal recursors as epi-recursors. *Proc. ACM Program. Lang.*, 8(POPL):425–456, 2024. doi:10.1145/3632857.

35 Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs 2005*, volume 3603 of *LNCS*, pages 294–309. Springer, 2005. doi:10.1007/11541868_19.

36 Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In Christian Urban and Xingyuan Zhang, editors, *ITP 2015*, volume 9236 of *LNCS*, pages 359–374. Springer, 2015. doi:10.1007/978-3-319-22102-1_24.

37 Natarajan Shankar. A mechanical proof of the Church-Rosser theorem. *J. ACM*, 35(3):475–522, 1988. doi:10.1145/44483.44484.

38 Kathrin Stark. *Mechanising syntax with binders in Coq.* PhD thesis, Saarland University, Saarbrücken, Germany, 2020. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/28822.

39 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *CPP 2019*, pages 166–180. ACM, 2019. doi:10.1145/3293880.3294101.

40 Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. A sound and complete projection for global types. In Adam Naumowicz and René Thiemann, editors, *ITP 2023*, volume 268 of *LIPIcs*, pages 28:1–28:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ITP.2023.28.

41 Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *LICS 2012*, pages 596–605. IEEE Computer Society, 2012. doi:10.1109/LICS.2012.75.

42 Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in Nominal Isabelle. *Log. Methods Comput. Sci.*, 8(2), 2012. doi:10.2168/LMCS-8(2:14)2012.

43 Jan van Brügge, James McKinna, Andrei Popescu, and Dmitriy Traytel. Barendregt convenes with Knaster and Tarski: Strong rule induction for syntax with bindings. *Proc. ACM Program. Lang.*, 9(POPL):57:1–57:32, 2025. doi:10.1145/3704893.

44 Jan van Brügge, Andrei Popescu, and Dmitriy Traytel. Implementation of MrBNF and case studies presented in this paper, 2025. doi:10.5281/zenodo.15756655.

45 Jérôme Vouillon. A solution to the POPLMark challenge based on de Bruijn indices. *J. Autom. Reason.*, 49(3):327–362, 2012. doi:10.1007/S10817-011-9230-5.