A new framework for Syntax with Binders

by

Jan van Brügge

Submitted for the degree of

Doctor of Philosophy

School of Mathematical and Computer Sciences
Heriot-Watt University

June 2026

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.

It also cannot provide a strengthened induction theorem for inductive definitions like standard β-reduction, as it relies on a syntactic criterion to ensure soundness which rules out many definitions for which such a strengthening would be sound.

Taking advantage of recent theoretical advancements that overcome these limitations through a modular approach using the concept of a Map-Restricted Bounded Natural Functor (MRBNF), we develop and implement a new definitional package for binding-aware datatypes and codatatypes in Isabelle/HOL. We also provide semantic criterion for the strengthening of inductive definitions that is able to capture e.g., standard β-reduction in the untyped λ-calculus without the need to add extra side conditions.

Finally, through the new notion of a Multi-Nominal Monad (MN-Monad), we are able to carry the substitutive structure of datatypes through composition and least fixpoint construction, which enables the automatic definition of substitution operators that compose across types.

Acknowledgements

First, I'd like to thank my supervisors at Heriot-Watt, Jurriaan Hage and James McKinna. They helped with my research, navigating university bureaucracy and encouraged me to widen my horizon with several summer schools.

Second, thank you to Heriot-Watt university for giving me the opportunity to persue this PhD by awarding me the James Watt Scholarship to fund my research.

Third, I am immensely grateful to have had a close collaboration with Dmitriy Traytel from the University of Copenhagen and Andrei Popescu from the University of Sheffield. During my PhD we have written several papers together that were published at international conferences.

Finally, I want to thank Jurriaan Hage and Dmitriy Traytel specifically again for reading early drafts of this thesis and providing valuable feedback that made this a much better work.

Inclusion of Published Works Form

Declaration

This thesis contains one or more multi-author published works. I hereby declare that the contributions of each author to these publications is as follows:

Citation details	Jan van Brügge, James McKinna, Andrei Popescu, and Dmitriy Traytel. “Barendregt Convenes with Knaster and Tarski: Strong Rule Induction for Syntax with Bindings”. In: Proc. ACM Program. Lang. 9.POPL (Jan. 2025). doi: 10.1145/3704893
Author 1	Design, Implementation
Author 2, 3 & 4	Design, Case Studies

Citation details	Jan van Brügge, Andrei Popescu, and Dmitriy Traytel. “Animating MRB-NFs: Truly Modular Binding-Aware Datatypes in Isabelle/HOL”. In: 16th International Conference on Interactive Theorem Proving (ITP 2025). Ed. by Yannick Forster and Chantal Keller. Vol. 352. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 11:1–11:20. isbn: 978-3-95977-396-6. doi: 10.4230/LIPIcs.ITP.2025.11
Author 1	Design, Implementation
Author 2 & 3	Design, Case Studies

Table of Contents

1 Introduction **1**

2 Exploring the problem **6**
2.1 Simple syntax . . . 6
2.2 Nested and mutual recursion . . . 9
2.3 Complex binders . . . 10
2.4 Nesting syntax . . . 12
2.5 Going beyond finite syntax . . . 13

3 Constructing binder types **15**
3.1 Preliminaries . . . 16
3.1.1 Higher-order Logic (HOL) . . . 17
3.1.2 Datatypes and Codatatypes . . . 18
3.1.3 Functors . . . 19
3.1.4 Monads . . . 20
3.2 Parsing the user-facing datatype syntax . . . 21
3.3 Datatypes and codatatypes . . . 21
3.3.1 Map-Restricted Bounded Natural Functors (MRBNFs) . . . 22
3.3.2 Closure under composition . . . 27
3.3.3 Binding fixpoint and α-equivalence . . . 30
3.3.4 Binding-aware recursor . . . 40
3.3.5 Renaming in datatypes . . . 49
3.3.6 Binding-aware corecursor . . . 50
3.3.7 Renaming in codatatypes . . . 55
3.4 Defining Substitution . . . 56
3.4.1 Multi-Nominal (MN) Monads . . . 57
3.4.2 Closure under composition . . . 62
3.4.3 Map-Restricted Substitutive BNFs (MRSBNFs) . . . 68
3.4.4 Closure under least fixpoint . . . 70
3.5 High-level syntax sugar . . . 72

4 Strong rule induction for inductive predicates **74**
4.1 Preliminaries . . . 75
4.1.1 Nominal sets . . . 75
4.1.2 The Knaster-Tarski fixpoint theorem . . . 77
4.2 Strengthening induction theorems for inductive definitions . . . 77

5 Implementation **82**
5.1 Manually constructing example proofs . . . 83
5.2 Generalizing the example . . . 84
5.3 Automating the construction with Isabelle/ML . . . 85
5.4 Major refactorings during the development . . . 88
5.4.1 Variable type class registry . . . 88
5.4.2 Using locales for the recursor . . . 89
5.5 Statistics . . . 90

6 Case Studies **91**
6.1 Type safety of the simply typed λ-calculus 91
6.2 The POPLmark challenge 93
6.2.1 Transitivity of subtyping 93
6.2.2 Type safety of System $F_{<:}$ 96
6.3 Mazza's infinite affine λ-calculus 100

7 Related Work **106**
7.1 Nameless representations 106
7.2 Higher-order abstract syntax 107
7.3 Nameful representations 107
7.4 Other comparisons between paradigms in the literature 108
7.5 Comparison to Nominal2 108

8 Conclusion and Future Work **110**

A Command syntax **113**
A.1 Binder (Co)Datatypes 113
A.2 Binder Inductives 113

Glossary

API Application Programming Interface.

BNF Bounded Natural Functor.

HOAS Higher-Order Abstract Syntax.

HOL Higher-Order Logic.

MN-Monad Multi-Nominal Monad.

MRBNF Map-Restricted Bounded Natural Functor.

MRSBNF Map-Restricted Substitutive Bounded Natural Functor.

ZF Zermelo-Fraenkel.

Chapter 1

Introduction

Most programming languages involve variable-binding constructs, or simply binders, such as the λ-abstraction or recursive and non-recursive let operators. For example, in the syntax of the untyped λ-calculus (see definition 1.1) we assume that the variable x in the λ-abstraction is bound in the body.

Definition 1.1. *Syntax of the untyped λ-calculus* $\quad t ::= x \mid t_1 \cdot t_2 \mid \lambda x.\, t$

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 λ-calculus expressions $\lambda x.\, x$ and $\lambda y.\, y$ as syntactically equal and to choose bound variables in a way that avoids name clashes with surrounding free variables – this is known as the *variable convention* popularized by Barendregt [5].

Furthermore, this convention not only applies to the syntax itself, but is also commonly used in inductively defined relations on syntax such as operational semantics. In an informal setting, it is often simply assumed that applying this convention to any given relation is sound. In reality, it is very easy to construct example where such an assumption is unsound:

Definition 1.2. *Unbinding of variables*

$$\frac{}{x \hookrightarrow [], x}\ \textsc{Var} \qquad \frac{}{t_1 \cdot t_2 \hookrightarrow [], t_1 \cdot t_2}\ \textsc{App} \qquad \frac{t \hookrightarrow xs, t'}{\lambda x.\, t \hookrightarrow x :: xs, t'}\ \textsc{Abs}$$

This example relation from Urban et al. [63, p. 38] strips the topmost binders from the syntax of the untyped λ-calculus. This relation is not functional, as multiple syntactically equal terms may have different bound variables. Nevertheless, it is is a well-defined relation, and not trivial. For example we have $\lambda x.\, (\lambda y.\, (x \cdot z)) \hookrightarrow [x, y], x \cdot z$, but also $\lambda y.\, (\lambda x.\, (y \cdot z)) \hookrightarrow [y, x], y \cdot z$. Note how the left hand sides of the relation are equal (they only differ in the choice of bound variables) while the right hand sides are different.

When performing induction on this relation, in the Abs case, we would like to be able to assume that the bound variable x is distinct from the free variables in the rule, in particular that x is not an element of the residual list xs. This however, would allow us to prove the following faulty lemma, where $\lambda xs.\, t$ is a shorthand for the iterated binding of all variables in the list xs:

Theorem 1.1 (False claim)**.** *If* $t \hookrightarrow (x :: xs), t'$ *and* $x \in \mathsf{FVars}\ t'$ *then* $x \in \mathsf{FVars}\ (\lambda xs.\, t')$

This theorem is false as can be seen by the counterexample $\lambda x.\, (\lambda x.\, x)$. So beyond just supporting the variable convention, any formal system also needs to be able to precisely know *when* it is sound to apply the convention at all. This is not only a problem in contrived examples; Bengtson [7] notes a similar issue with the classic presentation of the reduction relation in the π-calculus.

The mechanization of programming language metatheory in proof assistants struggles to keep up with this informal convention. The POPLmark challenge [3] initiated a flurry of approaches to mechanized binders, each with its own strengths and weaknesses (see chapter 7). 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 [18, 37]; (2) the nameful or nominal representation that includes bound variable names but identifies terms modulo α-equivalence, i.e., up to bound variable renaming [22]; and (3) the reductive representation that embeds the programming language's binders into the metalogic's binders [25, 48, 47].

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 [28] implements the nominal representation in the Isabelle/HOL proof assistant with successful applications ranging from Gödel's incompleteness theorems [46] to verifying the correctness of one of Haskell's compiler optimizations [11] and the security of authenticated data structures [14]. All these developments use the standard binder constructs: λ-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

examples like Mazza's infinitary affine λ calculus [36]. 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 the most flexible variant provided by Nominal2 [64], cannot directly incorporate linear patterns, which are required in most of the complex binder structures including those of the second part of the POPLmark challenge.

For inductive definitions, Nominal2 provides a strong induction rule if the rules that make up the definition follow a certain syntactic format [63]. While this criterion rules out definitions where strong induction would be unsound, it also requires extra side conditions on many common definitions, e.g. for β-reduction in the untyped λ-calculus. The user then has to prove that the relation with the extra side conditions defines the same relation as the definition from the literature. To quote the authors themselves [63, p. 48]: "This is annoying because both versions can be shown to define the same relation, but we have no general, and automatable, method for determining this."

To combat these limitations, we build upon Blanchette et al.'s work on Map-Restricted Bounded Natural Functors (MRBNFs) [9] as the foundation for a new framework for syntax with binders. MRBNFs are a generalization of Bounded Natural Functors (BNFs) [61], which underlie Isabelle's datatypes and codatatypes [10] (see section 3.1.2 for an overview of the differences between the two). Similar to these (co-)datatypes, it is not necessary for the user of our framework to know about its internals. In particular, they do not need to know what MRBNFs are. At the same time, advanced users and library authors can reach behind the abstraction to integrate their custom types by proving the MRBNF axioms (see section 3.3.1). Our framework is implemented as a definitional package for Isabelle/HOL and available on GitHub:

`https://github.com/jvanbruegge/binder_datatypes/tree/v0.1`

The main contributions in this thesis can be split into theoretical and technical contributions. The former are:

1. An algorithm for composing arbitrary MRBNFs (see section 3.3.2)

2. The notion of a Multi-Nominal Monad (MN-Monad) to capture the substitutive structure of datatypes that automatically composes across types (see section 3.4) which is a joint development with Andrei Popescu from the University of Sheffield and Dmitriy Traytel from the University of Copenhagen.

3. A semantic criterion for the rules that make up an inductive definition to provide a strong induction theorem that avoids bound variables (see chapter 4), a joint development with Andrei Popescu, Dmitriy Traytel and my second supervisor from Heriot-Watt University, James McKinna.

The technical contributions in the Isabelle implementation are:

4. An implementation of the core MRBNF structure (see section 3.3.1)

5. Automated proofs for the closure of MRBNFs under composition (see section 3.3.2)

6. The binding fixpoint construction including support for binding variables across types (see section 3.3.3)

7. For datatypes, a binding-aware recursor (see section 3.3.4) that is automatically instantiated to obtain a renaming operator (see section 3.3.5)

8. For codatatypes, the same renaming operator (see section 3.3.7) is defined with a binding-aware corecursor (see section 3.3.6). This corecursor is a joint development with my Master student Berk Özkütük [45].

9. An implementation of the core MN-Monad structure, including automated proofs for closure under composition and least fixed point (see section 3.4)

10. Automation to provide strengthened induction principles for inductively defined predicates (see section 4)

Additionally, we also provide an overview over the different axes along which simple syntax can be generalized (see chapter 2) and present our approach to implementing the framework in Isabelle/HOL (see chapter 5). We verify our framework with several case studies (see chapter 6) and show a comparison between the different approaches to syntax with binders in the literature (see chapter 7).

Throughout this thesis, we will omit most proofs for the theorems we present. Instead, we want to point to our Isabelle formalization, which contains full example proofs alongside the automation code.

This thesis is built upon the following two referenced conference publications as well as one unpublished draft:

- "Barendregt Convenes with Knaster and Tarski: Strong Rule Induction for Syntax with Bindings" by Jan van Brügge, James McKinna, Andrei Popescu and Dmitriy Traytel [12]
- "Animating MRBNFs: Truly Modular Binding-Aware Datatypes in Isabelle/HOL" by Jan van Brügge, Andrei Popescu and Dmitiry Traytel [13]

We reuse material from these publications with the permission of the coauthors.

Chapter 2

Exploring the problem

Contents

2.1 Simple syntax 6

2.2 Nested and mutual recursion 9

2.3 Complex binders 10

2.4 Nesting syntax 12

2.5 Going beyond finite syntax 13

2.1 Simple syntax

Our framework aims to provide a full-fledged solution to defining syntax with binders in Isabelle. What does that mean exactly? For this let us start with the simplest commonly used syntax: the untyped λ-calculus.

Definition 2.1. *Syntax of the untyped λ-calculus*

$$t ::= Vr\, x \mid t_1\, t_2 \mid \lambda x.\, t$$

Given an infinite set *Var* of variables – ranged over by x, y, z etc. – a λ-term is either (the injection of) a variable, or an application, or a λ-abstraction of a variable in a term. The terms are equated modulo the induced notion of α-equivalence, e.g. $\lambda x.\, (Vr\, x) = \lambda y.\, (Vr\, y)$.

Two terms are called α-equivalent if they only differ in the choice of their bound variables. Expressed differently, there has to exists a bijection between variables that is applied to all bound variables to make the terms equal. In the example above, the bijection that sends x to y, y to x and all other variables to themselves is such a bijection. On the other hand, the terms $\lambda x.\, Vr\, x$ and $\lambda y.\, Vr\, z$ are not α-equivalent, because no bijection on the bound variables exists that would make the terms equal.

In the literature, the explicit injection Vr of variables into terms is usually omitted, so we will write $\lambda x.\, x$ instead of $\lambda x.\, (Vr\, x)$. However, we will come back to this injection as it is vital for the existence of a term-for-variable substitution operation on terms (see section 3.4).

Aside from providing a type that models the terms of the syntax, the framework should also provide several common operations. The first is the *free variable* operator. Given a term t, $FVars\, t$ returns all the variables in t that are not bound by a λ-abstraction. It can be (uniquely) characterized by the following equations:

Definition 2.2. *Free variables of a λ-term*

$$\begin{aligned} FVars\, (Vr\, x) &= \{x\} \\ FVars\, (t_1\ t_2) &= FVars\, t_1 \cup FVars\, t_2 \\ FVars\, (\lambda x.\, t) &= FVars\, t \setminus \{x\} \end{aligned}$$

Second, we want a *renaming* function that allows us to substitute a free variable for another variable in a term (written as $t[x/y]$, see definition 2.3). More generally, we want a *parallel renaming* function that can rename multiple free variables at once (see definition 2.4). To ensure that there are always new fresh variables available, we have to restrict the function to rename only a set of variables with a cardinality strictly smaller than the type used as variables. We call such functions "small". Singular renaming then is just a special instance of parallel renaming with a function that sends x to y and all other variables to themselves.

Definition 2.3. *Singular renaming*

$$\begin{aligned} (Vr\, z)[x/y] &= \begin{cases} Vr\, y, & \text{if } x = z \\ Vr\, z, & \text{otherwise} \end{cases} \\ (t_1\ t_2)[x/y] &= (t_1[x/y])\ (t_2[x/y]) \\ (\lambda z.\, t)[x/y] &= \lambda z.\, (t[x/y]), \text{if } x, y \neq z \end{aligned}$$

Definition 2.4. *Parallel renaming*

$$\begin{aligned} (Vr\, z)[f] &= Vr\, (f\ z) \\ (t_1\ t_2)[f] &= (t_1[f])\ (t_2[f]) \\ (\lambda z.\, t)[f] &= \lambda z.\, (t[f]), \text{if } z \notin imsupp\ f \end{aligned}$$

Note that in both cases the variables being renamed may not clash with the binder in the abstraction case. For parallel renaming, $imsupp f$ is the union of the set of all variables that are not sent to themselves and the image of the function on those variables, i.e. $\bigcup\{\{a, f\, a\} \mid a.\, f\, a \neq a\}$. Throughout this thesis, we will use a slightly modified set comprehension notation that explicitly states which variables are bound

and thus are local to the comprehension. This makes complex set comprehensions easier to read. We also use the shorthand $\{a.\ P\ a\}$ to stand for $\{a \mid a.\ P\ a\}$, i.e. if the bound variable is not transformed.

Given that terms are α-equivalent classes and not freely constructed types, excluding the set $imsupp f$ as a side condition does not under-specify the function. As long as the set is small, the bound variable can be replaced by a fresh variable to fulfill this condition. There are several properties that we expect from such a renaming function:

1. renaming with the identity function should return the original term, $t[id] = t$
2. renaming twice should be the same as composing the renamings, $t[f][g] = t[g \circ f]$
3. renaming should only depend on the (values of the function on the) free variables, $(\forall x \in FVars\, t.\ f\ x = g\ x) \longrightarrow t[f] = t[g]$
4. extracting the set of free variables after renaming should be the same as applying the function to all elements in the set of free variables before renaming, $FVars\ (t[f]) = \{f\ a \mid a.\ a \in FVars\, t\}$

The last of the operations we expect the framework to provide is a term-for-variable substitution. Again, there exist singular and parallel variants similar to renaming, but we will directly focus on the parallel variety (we overload the syntax for renaming, using $\rho :: \mathsf{Var} \to \mathsf{Term}$ for the function; see definition 2.5).

Definition 2.5. *Parallel substitution*

$$\begin{aligned}
(Vr\ z)[\rho] &= \rho\ z \\
(t_1\ t_2)[\rho] &= (t_1[\rho])\ (t_2[\rho]) \\
(\lambda z.\ t)[\rho] &= \lambda z.\ (t[\rho]),\ \mathit{if}\ z \notin \mathit{IImsupp}\ \rho
\end{aligned}$$

Here the injection Vr of free variables becomes relevant again as it marks the spot where new terms can be substituted in. For example, the syntax of the π-calculus [38] (see definition 2.6) does have variables – called *channel names* – but it does not have such an injection. This means that we can only define parallel renaming for the syntax, but not substitution. Note that in the π-calculus, the forms $a(x).\ P$ and $v(x).\ P$ bind the variable x in the process P.

Definition 2.6. *Syntax of the π-calculus*

$$P, Q \quad ::= \quad 0 \mid P + Q \mid P \parallel Q \mid !P \mid [x = y]P \mid [x \neq y]P \mid \overline{a}\, x.P \mid a(x).P \mid v(x).P$$

Given that (parallel) substitution expects a function from variables to terms, the injection also acts as the analog to the identity function. So the set of variables that the function does not send to "themselves" – $IImsupp\rho$ – is defined as $\bigcup\{FVars(\rho a) \cup \{a\} \mid a.\ \rho\, a \neq Vr\, a\}$. Some properties of substitution mirror those of renaming, but there are also some additional ones. Given that our framework should be able to handle arbitrary syntax, we will only fix as much structure as necessary. For substitution, this means the existence of an injection Vr:

1. substituting with the injection Vr should return the original term, $t[Vr] = t$
2. for the injection Vr, substitution should return the result of the function, $(Vr\, x)[\rho] = \rho\, x$
3. performing two substitutions in sequence should be the same as substitution with the composition of the two functions[1], $t[\rho][\rho'] = t[\rho \,\square\, \rho']$
4. the free variables of the injection Vr is the singleton set, $FVars\,(Vr\, x) = \{x\}$
5. the free variables after substitution are the same set as the free variables after applying the function to all free variables before substitution, $FVars\,(t[\rho]) = \bigcup_{x \in FVars\, t} FVars\,(\rho\, x)$
6. substitution should only depend on the (values of the function on the) free variables, $(\forall x \in FVars\, t.\ \rho\, x = \rho'\, x) \longrightarrow t[\rho] = t[\rho']$

Here, properties 1, 3, 5 and 6 mirror those of renaming. Properties 1 through 3 are also the monad axioms of a *Kleisli triple* with the variable injection as *return* and substitution as *bind* (see section 3.1.4). We also expect renaming and substitution to be related by composing the function with the injection: $t[f] = t[Vr \circ f]$.

2.2 Nested and mutual recursion

The first natural generalization of our simple syntax extends the recursive structure of the syntax. Taking the untyped λ-calculus from the section 2.1, we might

[1] where the composition $(\rho \,\square\, \rho')\, x := (\rho\, x)[\rho']$

want to extend its application from binary to n-ary:

Definition 2.7. *Syntax of the untyped λ-calculus with n-ary application*

$$t ::= Vr\, x \mid t\, \overline{t_i} \mid \lambda x.\, t$$

The most direct way to define this type is to represent n-ary application as a pair of a term and a list of terms. However, this requires the framework to allow recursion through other types (*list* in this case). Ideally it should not only allow recursion through freely constructed types like lists, but also non-free datatypes like (finite) sets or Isabelle's subtypes (see section 3.1.1). Of course, the free variable, renaming and substitution operators should also generalize to this syntax. This hints at the necessity of some compositional construction in the framework (see section 3.3.2).

Furthermore, some calculi like Levy's call-by-push-value [32] distinguish between values and computations that mutually depend on each other (see definition 2.8 for the terms of an example calculus in the call-by-push-value style):

Definition 2.8. *Terms in a calculus using call-by-push-value semantics*

$$\begin{array}{lrcl} \textit{value terms} & v & ::= & Vr\, x \mid U\, \underline{c} \\ \textit{computation terms} & \underline{c} & ::= & \lambda x.\, \underline{c} \mid \underline{c}\, v \mid \textbf{\textit{return}}\, v \end{array}$$

Here, computations are embedded in values via the explicit *thunk* constructor U (short for unevaluated), and values are embedded in computations via the return constructor and the argument of application. Obviously this also requires that our operators on the terms are now mutually recursive families of functions. Furthermore we want to be able to freely combine the two, for example to introduce n-ary applications in our call-by-push-value terms.

2.3 Complex binders

Another axis of generalization is the binders themselves. The λ-abstraction only binds a single variable x. However, similar to the n-ary application in section 2.2, we may want to allow it to bind several variables at once. While for application, choosing a list-like structure was the obvious choice, for a multi-λ-function construct

there are several sensible options:

- a list, the most direct option. This ensures that the order of the bound variables is retained, however it would not prevent the same variable to be bound multiple times which may not be want we want.
- a subtype of list that ensures that all elements are pairwise distinct. Similar to a plain list, this also ensures that the order of binding matters, but it no longer allows duplicates.
- a (finite) set, in case we do not care about the exact ordering of bound variables.

Note that the second and third options rely on non-free datatypes for their binders, so the framework has to support that. Additionally, not only the binders themselves may be complex, the binding site is also of interest. For example, if we extend the untyped λ-calculus with a *letrec* construct (see definition 2.9), the binders are simultaneously bound in all of the right hand sides of the letrec expression as well as in the body.

Definition 2.9. *Syntax of the untyped λ-calculus with a letrec construct*

$$t ::= Vr\, x \mid t_1\, t_2 \mid \lambda x.\, t \mid \textbf{\textit{letrec}}\, \overline{x_i = t_i}\, \textbf{\textit{in}}\, t$$

Again, if we want to ensure that all the binders of such expressions are unique, we could choose a subtype of a list – or set – of pairs that ensures that the first components of the pairs are pairwise distinct. Such a representation combines the nested recursion through a non-free datatype from section 2.2 with a more complex binding structure.

However, not only flat structures are useful as binder, tree-like binders also appear in many calculi. For example if we extend the untyped λ-calculus with record terms and a case expression which scrutinizes such records, we may want to allow the pattern in the case expression to be arbitrarily nested:

Definition 2.10. *Syntax of the untyped λ-calculus with records and case-expressions*

$$\begin{array}{llll} \textit{terms} & t & ::= & Vr\, x \mid t_1\, t_2 \mid \lambda x.\, t \mid \{l_1 = t_1, \ldots, l_n = t_n\} \mid \textbf{\textit{case}}\, t\, \textbf{\textit{of}}\, p \to t \\ \textit{patterns} & p & ::= & PVr\, x \mid \{l_1 = p_1, \ldots, l_n = p_n\} \end{array}$$

Here, in the case expression, all of the variables in the pattern – no matter how deeply nested – are bound in the right hand side. Similar to the *letrec* and multi-λ terms, we can use a subtype of the pattern type that ensures uniqueness of variables.

2.4 Nesting syntax

Binding across different syntaxes like in section 2.3 gets even more interesting if they have substitution operators. For example System F [24], a polymorphic extension of the simply typed λ-calculus, has types containing type variables from an infinite set $TVar$ – ranged over by $a, b, \ldots$ – and terms that contain term variables as well as type variables (see definition 2.11).

Definition 2.11. *Types and terms of System F*

$$\begin{array}{llll} \textit{types} & \tau & ::= & TVr\, a \mid \tau_1\, \tau_2 \mid \forall a.\, \tau \\ \textit{terms} & t & ::= & Vr\, x \mid t_1\, t_2 \mid \lambda x : \tau.\, t \mid t\, \tau \mid \Lambda a.\, t \end{array}$$

The types in System F are isomorphic to the terms in the untyped λ-calculus. Thus they obviously have the relevant free (type) variable, renaming and substitution operations. The terms however nest the types in their syntax and also feature a type variable binder that binds type variables in a term (and thus in its nested types). The important detail here is that while terms have an injection of term variables into terms, they do not have their own injection of type variables. Using the naive approach from section 2.1 where only these injections give rise to substitutions would thus not result in the desired operator. The other operators, i.e. the free variable operator – here called $FTVars$ for free type variables – and renaming, do not have this issue.

What we want here is not an entirely new substitution operator, we would like to automatically *lift* the substitution operator for types to also work on terms. We also want to transfer the properties of substitution mentioned in section 2.1 to the new lifted operator as far as they make sense. In particular we want the following properties for the type-in-term substitution (written as $t[\rho]_\tau : t \to (TVar \to \tau) \to t$):

1. $t[TVr]_\tau = t$
2. $t[\rho]_\tau[\rho']_\tau = t[\rho \,\square_\tau \rho']_\tau$

3. $FTVars\,(t[\rho]_\tau) = \bigcup_{a \in FTVars\,t} FVars_\tau(\rho\,a)$

4. $(\forall a \in FTVars\,t.\ \rho\,a = \rho'\,a) \longrightarrow t[\rho]_\tau = t[\rho']_\tau$

The only two properties that do not make sense for the lifted version are those that depend on an injection that returns the term type, i.e. the free variables of the injection and the left identity (properties 2 and 4 in section 2.1).

The key question here is how to do this lifting of substitution and its properties automatically and generically so it works for any syntax the user might want to define. Our solution is to treat the substitution monad as *relative* to other monads and maintain this structure under composition and least fixpoint (see section 3.4).

2.5 Going beyond finite syntax

While most commonly used calculi use finite syntax, especially in the study of the semantics of programming languages, infinite syntax is a useful tool. There are two different axes along which a syntax may be infinite: A syntax tree may have finite depth but infinite width, it may have finite width but infinite depth, or it may be infinite along both axes.

An example of the first kind from the literature is Mazza's infinite affine λ-calculus [36]:

Definition 2.12. *Syntax of Mazza's infinite affine λ-calculus*

$$t ::= Vr\,x \mid t\,ts \mid \lambda xs.\,t$$

Here, $ts = (ts_i)_{i \in \mathbb{N}}$ is a stream of λ-terms and $xs = (xs_i)_{i \in \mathbb{N}}$ is a non-repetitive stream of variables. Within $\lambda xs.\,t$ all variables in the stream xs are bound simultaneously in t. Note that this syntax features not only infinitely many bound variables, but also infinitely many terms (and thus potentially infinitely many free variables).

To define the renaming and substitution operators we must ensure that there are always enough fresh variables available for the binders. Indeed, if we assumed that Var is only countably infinite, unary substitution would already be problematic. For example, let $(x_i)_{i \in \mathbb{N}}$ be an enumeration of Var, and let $t = (Vr\,x_1)(Vr\,x_3)\ldots$ and $s = (Vr\,x_1)(Vr\,x_2)\ldots$ (i.e. infinite application terms consisting of the odd-indexed

variables and all variables, respectively). Then the result of $\lambda(x_{2*i})_{i\in\mathbb{N}}.\, t)[s/x_1]$ is impossible to compute, because this result would need to have all variables in *Var* as free, and thus we would have no (fresh) variables left for the λ-binder.

Therefore we have to assume *Var* has at least uncountable cardinality, which ensures that we can produce fresh variables to avoid capture from the (countably infinite) stream binders.

The other interesting case of infinitary syntax comes from allowing terms that are not necessarily well-founded, i.e. may have infinite depth. The classic example are the (possibly) infinite-depth λ-terms (which we will refer to as λ-coterms, see definition 2.13) that form the basis of the concepts of Böhm, Lévy-Longo and Berarducci trees, used for studying the semantics of the λ-calculus [6, 29].

Definition 2.13. *λ-coterms*

$$t ::^{\infty}= \bot \mid Vr\, x \mid t_1\, t_2 \mid \lambda x.\, t$$

This syntax is coinductively interpreted and not inductively, i.e. it is allowed to form coterms from an infinite number of constructor applications. For example the (necessarily unique) coterm $t = \lambda x.\, t\, (Vr\, x)$ is an inhabitant of the λ-coterms.

One may of course argue whether the λ-coterms should even be called "syntax". The reason why they can be fruitfully regarded as syntax is that they can be identified up to their natural notion of α-equivalence (defined coinductively rather than inductively) and, importantly, admit a (corecursively defined) notion of substitution that is again as well-behaved as that on normal λ-terms.

Our framework can accommodate all of these generalizations of simple syntax. We show how to construct types that represent syntax in chapter 3.

Chapter 3

Constructing binder types

Contents

3.1 Preliminaries 16

3.1.1 Higher-order Logic (HOL) 17

3.1.2 Datatypes and Codatatypes 18

3.1.3 Functors 19

3.1.4 Monads 20

3.2 Parsing the user-facing datatype syntax 21

3.3 Datatypes and codatatypes 21

3.3.1 Map-Restricted Bounded Natural Functors (MRBNFs) . . 22

3.3.2 Closure under composition 27

3.3.3 Binding fixpoint and α-equivalence 30

3.3.4 Binding-aware recursor 40

3.3.5 Renaming in datatypes 49

3.3.6 Binding-aware corecursor 50

3.3.7 Renaming in codatatypes 55

3.4 Defining Substitution 56

3.4.1 Multi-Nominal (MN) Monads 57

3.4.2 Closure under composition 62

3.4.3 Map-Restricted Substitutive BNFs (MRSBNFs) 68

3.4.4 Closure under least fixpoint 70

3.5 High-level syntax sugar 72

The construction of the binder (co)datatypes manages to share most of the proofs and definitions between datatypes and codatatypes until the definitions of the recursor and corecursor (see figure 3.1 for a rough overview of the steps involved).

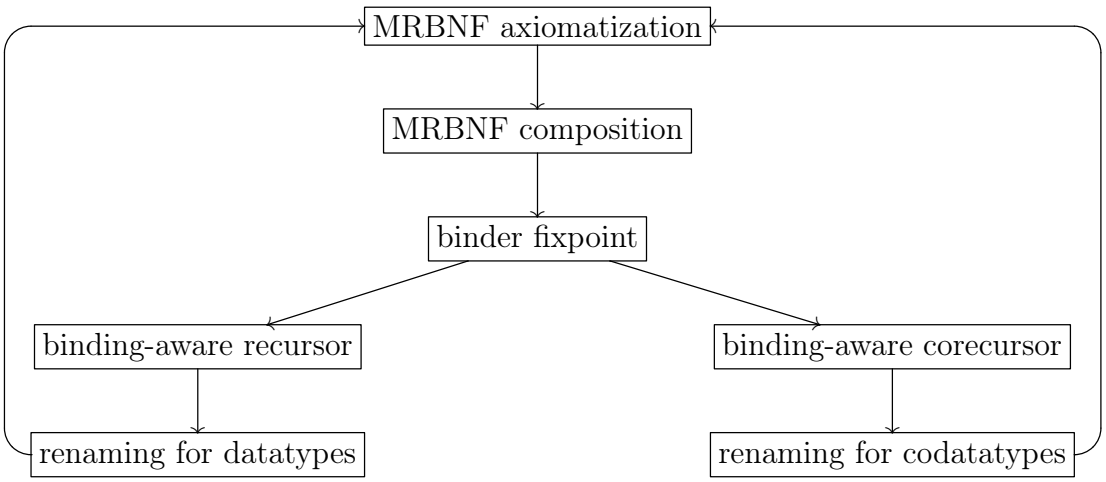


Figure 3.1: Pipeline for binder (co)datatype creation

The main reason is that a coinductive definition of α-equivalence can also be used for an inductively defined syntax while the reverse would not be possible.

At the core of the construction is a special class of functors, the Map-Restricted Bounded Natural Functor (MRBNF) (see section 3.3.1). We can prove that any type is a MRBNF by composition (see section 3.3.2). Given a suitable MRBNF we can define its least (for datatypes) or greatest (for codatatypes) binder fixpoint and define its quotient with respect to α-equivalence (see section 3.3.3).

For datatypes, we then define the binding-aware recursor (see section 3.3.4). By instantiating it, we obtain a renaming function for the datatype (see section 3.3.5). With this renaming function as *map* function and the free variable operators as *set* functions, we can prove the MRBNF axioms for the fixpoint datatype.

For codatatypes, we instead define the binding-aware corecursor (see section 3.3.6). Again, by instantiating it, we obtain a renaming function (see section 3.3.7) that also allows us to prove the MRBNF axioms for the codatatype.

3.1 Preliminaries

We have implemented our framework in Isabelle/HOL. We give a short overview over this logic in section 3.1.1 as well of the differences between datatypes and codatatypes (see section 3.1.2). Our construction is based on the categorical notions of functors and monads which we will summarize in sections 3.1.3 and 3.1.4.

$$
\begin{array}{lll}
T_{\text{prim}} ::= & & \\
\quad | & \alpha & \text{Variables} \\
\quad | & \mathit{bool} & \text{Booleans} \\
\quad | & \mathit{ind} & \text{Infinite type} \\
\quad | & T \to T & \text{Function Types}
\end{array}
$$

Figure 3.2: Primitive types of HOL

$$
\begin{array}{lllll}
t & ::= & & & \\
& | & (=) & : \alpha \to \alpha \to \mathit{bool} & \text{Equality} \\
& | & 0 & : \mathit{ind} & \text{Zero} \\
& | & \mathit{Succ} & : \mathit{ind} \to \mathit{ind} & \text{Successor} \\
& | & \epsilon & : (\alpha \to \mathit{bool}) \to \alpha & \text{Hilbert Choice operator}
\end{array}
$$

Figure 3.3: Primitive terms of HOL, including their types

3.1.1 Higher-order Logic (HOL)

The default and most widely used object logic for the Isabelle proof assistant is Higher-Order Logic (HOL), written as Isabelle/HOL. The core of HOL is based on Church's simple type theory [15] with rank-1 polymorphism where terms are implicitly α-, β- and η-equivalent. It thus is a weaker logic than other object logics supported by Isabelle like Zermelo-Fraenkel (ZF) set theory that work with arbitrary sets as opposed to HOL's types. In HOL, primitive types (see figure 3.2) can be used to define more complex types with a *typedef* ($T' = \{x :: T.\, P\, x\}$), which carves out a non-empty subset of a type T with a predicate P. Terms can be constructed from primitive terms (see figure 3.3) and λ-abstraction and application. The primitive terms witness two of the main axioms of the logic, the axiom of infinity[1] (through 0 and *Succ*) and the axiom of choice[2] (through Hilbert's choice operator).

The HOL library defines many commonly used types. From these types, the most relevant for this thesis are:

- $\alpha\ \mathit{set}$, the type of sets of elements of type α (i.e., the powertype of α)
- $\alpha + \beta$, the disjoint union of α and β
- $\alpha \times \beta$, the cartesian product of α and β
- $\alpha \Rightarrow \beta$, the type of functions from α to β

[1] There exists at least one infinite set: The set of the natural numbers
[2] Given an infinite set, it is possible to choose exactly one element from the set

3.1.2 Datatypes and Codatatypes

Most types commonly used in theorem proving are datatypes. For example lists, natural numbers or products fall in this category. In a datatype, constructors may only be a applied a finite number of times. This means that e.g. every list will always contain a finite number of elements.

To transform datatypes, *recursive* functions are used. These destruct the type layer by layer to return a final result, i.e., the input of the function is the datatype and the result is an arbitrary type. For example, to count the length of the list, we pattern match on the constructors of the type and recurse on the subcomponents:

```
datatype 'a list = Nil | Cons 'a "'a list"

fun len :: "nat list => nat" where
  "len Nil = 0"
| "len (Cons _ xs) = 1 + len xs"
```

The most important property for the validity of this definition is its *termination*. In Isabelle, the `fun` command automatically proves termination in many cases. Termination is required for the soundness of the logical system. Take for example a slightly adapted definition of `len`:

```
fun len :: "nat list => nat" where
  "len Nil = 0"
| "len (Cons x xs) = 1 + len (Cons x xs)"
```

If this definition were allowed, we could use simple arithmetic to derive a contradiction: If you subtract `len (Cons x xs)` from both sides of the equation in the second case of `len`, you are left with $0 = 1$.

Codatatypes on the other hand do allow an infinite repetition of constructors. For example the type of infinite streams only has a single constructor:

```
codatatype 'a stream = Elem 'a "'a stream"
```

Contrary to recursive functions that consume datatypes, *corecursive* functions build up a codatatype. For example we can define a function that builds an (infinite) stream of every other natural number from a given starting point:

```
primcorec every_other :: "nat => nat stream" where
  "every_other n = Elem n (every_other (n + 2))"
```

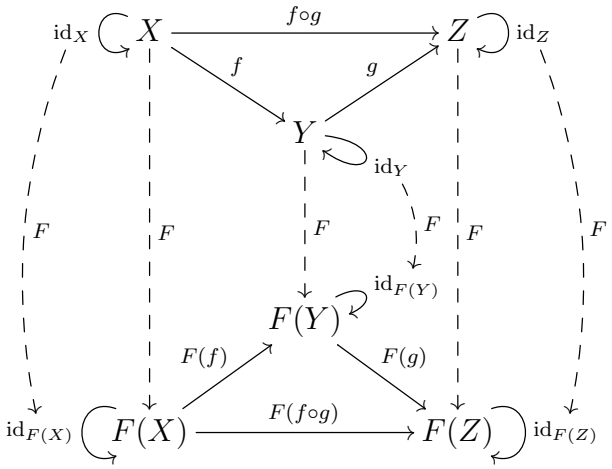


Figure 3.4: Preservation of composition and identity

While for recursive functions termination is required, corecursive functions require *productivity*. This means that the function needs to guard its recursive call behind at least one constructor application. This allows to evaluate the function "up to a point", i.e, only as far as needed. For this reason, codatatypes also model lazy types like those found in the Haskell programming language [35].

3.1.3 Functors

In category theory, a functor F is a mapping between two categories $\mathcal{C}$ and $\mathcal{D}$. It associates every object X in $\mathcal{C}$ with an object $F(X)$ in $\mathcal{D}$ and every morphism $f : X \to Y$ in $\mathcal{C}$ with a morphism $F(f) : F(X) \to F(Y)$ in $\mathcal{D}$. The transformation has to preserve identity and composition:

- $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for every object X in $\mathcal{C}$
- $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f : X \to Y$ and $g : Y \to Z$ in $\mathcal{C}$

In other words, a functor F has to make the diagram in figure 3.4 commute. Two of the most basic functors are the *identity* and *constant* functors. The former, usually written as $1_{\mathcal{C}}$, maps every object X and every morphism f in a category $\mathcal{C}$ to iself. It thus is an endofunctor on $\mathcal{C}$. The constant functor maps every object in a category $\mathcal{C}$ to a fixed object X in a category $\mathcal{D}$, and every morphism in $\mathcal{C}$ to the identity morphism on X.

It is also useful to talk about mappings between functors. Given two functors F and G between categories $\mathcal{C}$ and $\mathcal{D}$, a family of morphisms $(\eta_X)_{X \in \mathcal{C}}$, where $\eta_X : F(X) \to G(X)$ is called a *natural transformation* if it respects the structure of

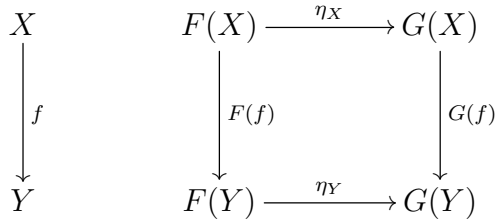


Figure 3.5: Natural transformations respect the structure of functors

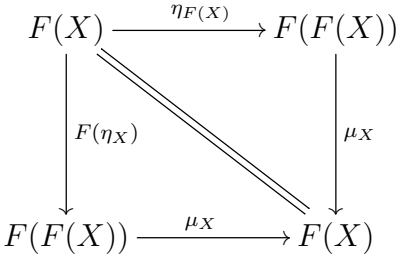


Figure 3.6: Monad unit law

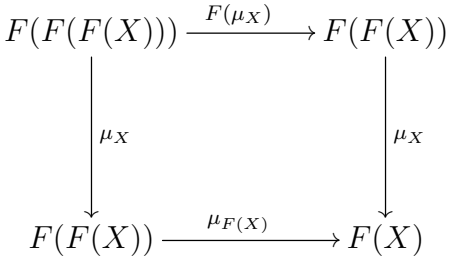


Figure 3.7: Monad associativity law

the functors: for all morphisms $f : X \to Y$ in $\mathcal{C}$, $\eta_Y \circ F(f) = G(f) \circ \eta_X$. Again this property can be expressed as a commutative diagram (see figure 3.5). Natural transformations can be considered "morphisms of functors", and indeed form the basis of the category of functors.

For this thesis we only use (endo)functors where both categories have (HOL) types as objects, and functions as morphisms. For example, the *list* functor maps types to lists of types, and maps functions to functions on lists.

3.1.4 Monads

A monad is a specific kind of endofunctor $F : \mathcal{C} \to \mathcal{C}$, together with two natural transformations $\eta_X : 1_{\mathcal{C}}(X) \to F(X)$ and $\mu_X : F(F(X)) \to F(X)$ that fulfill:

- the unit law, $\mu_X \circ F(\eta_X) = \mu_X \circ \eta_{F(X)} = 1_{F(X)}$ (see figure 3.6)
- the associativity law, $\mu_X \circ F(\mu_X) = \mu_X \circ \mu_{F(X)}$ (see figure 3.7)

Equivalently, we can also define a monad via a *Kleisli triple* [41] consisting of an endofunctor F, a morphism $return_X : X \to F(X)$, and a morphism $bind\ f : F(X) \to F(Y)$ for every morphism $f : X \to F(Y)$, such that:

- $bind\ f \circ return_X = f$ (right identity)
- $bind\ return_X = 1_{F(X)}$ (left identity)
- $bind\ (bind\ g \circ f) = bind\ g \circ bind\ f$ (associativity)

In this thesis we will use this second definition when presenting our work on substitutions (see section 3.4) because the parallel substitution operator is precisely this monadic *bind* operator.

3.2 Parsing the user-facing datatype syntax

Before any of the core construction can happen, the user needs to specify the exact syntax they want to use. To do so, the framework implements a custom syntax that leans heavily on the syntax for Isabelle datatypes [10] (the precise syntax is specified in appendix A.1). For example, to define the syntax of the untyped λ-calculus, a datatype with three constructors and a single binding annotation is sufficient:

```
binder_datatype (FVars: 'a) term =
  Var 'a
| App "'a term" "'a term"
| Lam x::'a t::"'a term" binds x in t
```

Internally, this specification is parsed and converted into a *pre-datatype*. This is a non-recursive type with "holes" instead of the variable positions and the recursive occurrences. Different holes are used whether a variable or a recursive occurrence appears in a binding clause or not. The constructors are represented as a sum of products. For example, the specification above would translate into the following pre-datatype where the infix product type constructor binds stronger than the sum type constructor:

```
type_synonym ('a, 'b, 'rec, 'brec) term_pre = 'a + 'rec * 'rec + 'b * 'brec
```

3.3 Datatypes and codatatypes

With the parsed specification the framework proves that the pre-datatype is a MRBNF (see section 3.3.1) by composition (see section 3.3.2), constructs the recursive type as least or greatest fixpoint of the pre-datatype (see section 3.3.3) and defines the binding-aware (co)recursor for the datatype (see sections 3.3.4 and 3.3.6).

3.3.1 Map-Restricted Bounded Natural Functors (MRBNFs)

One fundamental design decision behind our framework is that most parts of the construction do not need to know the exact shape of the syntax the user wants to define. For this reason, all operations are defined abstractly on type constructors and not on concrete types.

At the core of the binder datatype construction sits a class of functors: the Map-Restricted Bounded Natural Functor (MRBNF)[3], first introduced by Blanchette et al. [9] (see definition 3.1). It is a generalization of the Bounded Natural Functor (BNF) which forms the foundation of Isabelle's datatypes [61]. A type constructor is a BNF if it is a functor in both Set and Rel, i.e., it has a *map* function that lifts functions on elements to functions on the type constructor and a *relator* that lifts relations on elements to relations on the type constructor. It also has to have a *set* function that extracts all the elements into a set that has to be bounded by an infinite cardinal. This cardinal may not depend on the element type.

In general, most container-like types like lists, trees, sums and products fall into this class of functor. Sums and products are also examples of BNFs that feature more than one position on the type constructor as both the left and right injection into a sum as well as the first and second element of a tuple may have different types. Some types are only BNFs in some of their positions, for example the function arrow. While we cannot define a function to "extract" all elements in the domain of a function, we can enumerate its codomain. This set is bounded by the cardinality of the domain type, but given that the domain type is not part of the functorial structure, this is not an issue. We call the functorial positions *live* while all other positions are called *dead*.

BNFs admit and are closed under both initial algebra and final coalgebra as well as composition. More importantly, these algebras are expressible in HOL and for example do not require higher kinded polymorphism. This makes them a perfect basis for datatypes in Isabelle as we can build arbitrarily complex types from simple blocks by composing and iterating them into recursive datatypes.

However, not all positions on types are BNFs, most of them due to the lack of a bounded number of elements like seen with the function space earlier. The powerset type also is not a BNF for this reason (and neither is it a MRBNF as that

[3] pronounced "Mister BNF"

still requires boundedness, see axiom 5). Another blocker may be the map function itself. BNFs require that the map function accepts arbitrary functions. If we take the type of distinct lists, i.e., lists where all the elements are pairwise disjoint, we do not have such a map function as we would be able to map all elements to a constant, breaking the disjointness invariant.

For defining syntax, this distinct list type and other similar types are very common. For example, most languages do not allow repeated variable names in the arguments of function definitions. A natural way to represent this would be via such a distinct list type. This is where MRBNFs add more granularity to the kinds of positions. Instead of just having live and dead positions, MRBNFs also feature positions we called *free* and *bound*. As the name suggests, in our construction we will use those positions for free and bound variables respectively. These positions are part of the functorial structure, but contrary to live positions they do not have to accept arbitrary functions. Instead, both only have to accept small endofunctions[4] with bound positions further restricting this requirement to bijections.

This means the type of distinct lists is a MRBNF that has one bound position. Thus, the map function only has to accept endobijections which make it impossible to break the invariant of the type. Going back to the example of function arguments, given that we use bound positions to represent bound variables, we can now use distinct lists to represent the arguments in our syntax.

In the following sections, a $(m_1,\ m_2,\ n)$-ary MRBNF has m_1 free positions, m_2 bound positions and n live positions. In definitions and theorems we will use α for free positions, β for bound positions, γ for live positions and δ for dead positions. For ease of presentation, the type constructors will always have all the free positions first, followed by the bound, live and dead positions. In practice they can be arbitrarily permuted as long as the relevant functions have their arguments in the same order. We also use m as the sum of the number of bound and free positions.

Throughout this thesis we use the notation $\overline{x}_k$ to denote a repetition of x. The subscript k denotes the number of repetitions. We omit this number if it is clear from context or irrelevant. For example, as dead positions do not contribute to the functorial structure of a MRBNF, the exact number of dead type variables δ is

[4] functions that are the identity on all but a bounded number of values

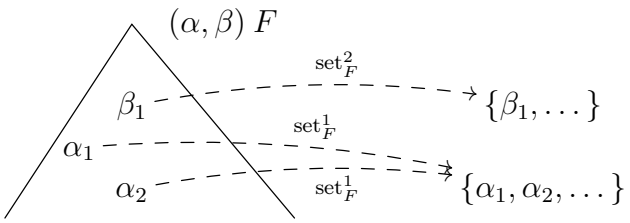


Figure 3.8: Set functions of a MRBNF

irrelevant.

Definition 3.1 (MRBNF). $\mathcal{F} = \left(\left(\overline{\alpha}_{m_1}, \overline{\beta}_{m_2}, \overline{\gamma}_n, \overline{\delta}\right) F,\ map_F,\ \overline{set_F},\ rel_F,\ bd_F\right)$ *where*

$$
\begin{aligned}
map_F \quad &: \quad \overline{(\alpha \to \alpha)} \to \overline{(\beta \to \beta)} \to \overline{(\gamma \to \gamma')} \to (\overline{\alpha}, \overline{\beta}, \overline{\gamma}, \overline{\delta})\ F \to (\overline{\alpha}, \overline{\beta}, \overline{\gamma'}, \overline{\delta})\ F \\
set^i_F \quad &: \quad \begin{cases} (\overline{\alpha}, \overline{\beta}, \overline{\gamma}, \overline{\delta})\ F \to \alpha_i\ set, & \text{if } i \leq m_1 \\ (\overline{\alpha}, \overline{\beta}, \overline{\gamma}, \overline{\delta})\ F \to \beta_{(i-m_1)}\ set, & \text{if } m_1 < i \leq m \\ (\overline{\alpha}, \overline{\beta}, \overline{\gamma}, \overline{\delta})\ F \to \gamma_{(i-m)}\ set, & \text{otherwise} \end{cases} \\
rel_F \quad &: \quad \overline{(\alpha \to \alpha)} \to \overline{(\beta \to \beta)} \to \overline{(\gamma \to \gamma' \to bool)} \\
&\quad\ \to (\overline{\alpha}, \overline{\beta}, \overline{\gamma}, \overline{\delta})\ F \to (\overline{\alpha}, \overline{\beta}, \overline{\gamma'}, \overline{\delta})\ F \to bool \\
bd_F \quad &: \quad \text{is an infinite, regular cardinal}
\end{aligned}
$$

such that $\forall i \in \{1 \dots m_1\}.\ |\alpha_i| \geq_o bd_F$ *and* $\forall j \in \{1 \dots m_2\}.\ |\beta_i| \geq_o bd_F$ *is called a* (m_1, m_2, n)*-MRBNF if the following axioms hold:*

Axiom 1 (map_id).

$$\mathrm{map}_F\ \overline{\mathrm{id}} = \mathrm{id}$$

Axiom 2 (map_comp).

$$\frac{\begin{array}{c}\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall j \in \{1 \dots m_2\}.\ |\mathbf{supp}\ f_{j+m_1}| <_o |\beta_j| \wedge \mathbf{bij}\ f_{j+m_1} \\ \forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ g_i| <_o |\alpha_i| \\ \forall j \in \{1 \dots m_2\}.\ |\mathbf{supp}\ g_{j+m_1}| <_o |\beta_j| \wedge \mathbf{bij}\ g_{j+m_1}\end{array}}{\mathrm{map}_F\ \overline{f} \circ \mathrm{map}_F\ \overline{g} = \mathrm{map}_F\ \overline{(f \circ g)}}$$

As a functor, every MRBNF needs to have a *map* function which preserves identity and composition (see axioms 1 and 2). In general, this map function requires small-support[5] endofunctions for the free positions and small-support endobijections

[5] $\mathbf{supp}\ f := \{a.\ f\ a \neq a\}$

for the bound positions. This requirement for small-support functions is what we call the map restriction of the MRBNF. As the identity function has an empty support and is a bijection it fulfils this restriction.

Axiom 3 (map_cong).

$$\frac{\begin{array}{c}\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall j \in \{1 \dots m_2\}.\ |\mathbf{supp}\ f_{j+m_1}| <_o |\beta_j| \wedge \mathbf{bij}\ f_{j+m_1} \\ \forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ g_i| <_o |\alpha_i| \\ \forall j \in \{1 \dots m_2\}.\ |\mathbf{supp}\ g_{j+m_1}| <_o |\beta_j| \wedge \mathbf{bij}\ g_{j+m_1} \\ \forall k \in \{1 \dots (m+n)\}.\ \forall a.\ a \in set^k_F\ x \longrightarrow f_k\ a = g_k\ a\end{array}}{\mathrm{map}_F\ \overline{f}\ x = \mathrm{map}_F\ \overline{g}\ x}$$

Axiom 4 (set_map).

$$\frac{\begin{array}{c}\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall j \in \{1 \dots m_2\}.\ |\mathbf{supp}\ f_{j+m_1}| <_o |\beta_j| \wedge \mathbf{bij}\ f_{j+m_1}\end{array}}{\forall k \in \{1 \dots (m+n)\}.\ \mathrm{set}^k_F\ \left(\mathrm{map}_F\ \overline{f}\ x\right) = \{f_k\ a \mid a.\ x \in \mathrm{set}^k_F\ x\}}$$

Axioms 3 and 4 are about the interaction of *map* with the so-called *set* functions. These functions extract all elements in the "holes" of one position into a set (see figure 3.8). The *map_cong* axiom ensures that the map function only depends on the elements of the sets returned by the set functions. The other axiom states that the set functions are natural transformations from the MRBNF to the powerset functor. Viewed differently, it states that it does not matter if we first map over a term x and then extract the set set^k_F or if we first extract the set and then apply the function f_k to all elements in the set.

Axiom 5 (set_bd).

$$\forall k \in \{1 \dots (m+n)\}.\ |\mathrm{set}^k_F\ x| <_o \mathrm{bd}_F$$

Axiom 5 ensures that all the set functions return bounded sets. The bound has to be an infinite regular cardinal. For our application of syntax with binders, this axiom – combined with the requirement that the types used for bound and free positions is at least as large as the cardinal bound and thus at least countable – ensures that we always have enough fresh variables. The regularity of the cardinal bound is needed for nested sets: if every set in a bounded set of sets is itself bounded, the union of all sets is bounded by the same cardinal.

Axiom 6 (rel_eq).

$$\mathrm{rel}_F\ \overline{\mathrm{id}}_m\ \overline{(=)}_n = (=)$$

Axiom 7 (rel_comp).

$$\frac{\begin{array}{c}\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall j \in \{1 \dots m_2\}.\ |\mathbf{supp}\ f_{j+m_1}| <_o |\beta_j| \wedge \mathbf{bij}\ f_{j+m_1} \\ \forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ g_i| <_o |\alpha_i| \\ \forall j \in \{1 \dots m_2\}.\ |\mathbf{supp}\ g_{j+m_1}| <_o |\beta_j| \wedge \mathbf{bij}\ g_{j+m_1}\end{array}}{\mathrm{rel}_F\ \overline{f}\ \overline{R} \diamond \mathrm{rel}_F\ \overline{g}\ \overline{S} \leq \mathrm{rel}_F\ \overline{(f \circ g)}\ \overline{(R \diamond S)}}$$

A MRBNF is not only a functor in Set, but also a functor in Rel, the category of relations. Axioms 6 and 7 are the identity and composition axioms for this functor. Here, relation composition is defined as $(R \diamond S)\ x\ y := \exists z.\ R\ x\ z \wedge S\ z\ y$.

As the bound and free positions are limited to endofunctions, the relator rel_F uses the graph of the function f_i[6] to relate these positions.

Axiom 8 (rel_def).

$$\frac{\begin{array}{c}\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall j \in \{1 \dots m_2\}.\ |\mathbf{supp}\ f_{j+m_1}| <_o |\beta_j| \wedge \mathbf{bij}\ f_{j+m_1}\end{array}}{\begin{array}{l}\mathrm{rel}_F\ \overline{f}\ \ \overline{R}\ \ x\ \ y = \exists z.\ (\forall k \in \{1 \dots n\}.\ \mathrm{set}_F^{k+m}\ z \subseteq \{(a, b) \mid R_k\ a\ b\}) \\ \qquad \wedge \mathrm{map}_F\ \overline{\mathrm{id}}\ \ \overline{\mathrm{fst}}_n\ z = x \wedge \mathrm{map}_F\ \overline{f}\ \ \overline{\mathrm{snd}}_n\ z = y\end{array}}$$

The last axiom ensures the relator works pairwise on the elements of the type. It requires that there exists a term z that has pairs of elements for the live positions. Using the structure-preserving *map* we can obtain the original terms x and y.

The binary sum and product types are typical examples for a MRBNF (see definitions 3.2 and 3.3).

[6] $\mathrm{Grp}\ f := \lambda x\ y.\ fx = y$

Definition 3.2 (Sum MRBNF).

$$
\begin{array}{lcl}
(\gamma_1, \gamma_2)\ F & := & \gamma_1 + \gamma_2 \\
map_{sum} & := & \lambda f_1\ f_2\ x.\ \textbf{\textit{case}}\ x\ \textbf{\textit{of}} \\
 & & \mid Inl\ y \Rightarrow Inl\ (f_1\ y) \\
 & & \mid Inr\ y \Rightarrow Inr\ (f_2\ y) \\
set^1_{sum} & := & \lambda x.\ \textbf{\textit{case}}\ x\ \textbf{\textit{of}} \\
 & & \mid Inl\ y \Rightarrow \{y\} \\
 & & \mid Inr\ \_ \Rightarrow \{\} \\
set^2_{sum} & := & \lambda x.\ \textbf{\textit{case}}\ x\ \textbf{\textit{of}} \\
 & & \mid Inl\ \_ \Rightarrow \{\} \\
 & & \mid Inr\ y \Rightarrow \{y\} \\
rel_{sum} & := & \lambda R_1\ R_2\ \ x\ y.\ \textbf{\textit{case}}\ (x, y)\ \textbf{\textit{of}} \\
 & & \mid (Inl\ z_1, Inl\ z_2) \Rightarrow R_1\ z_1\ z_2 \\
 & & \mid (Inr\ z_1, Inr\ z_2) \Rightarrow R_2\ z_1\ z_2 \\
 & & \mid \_ \Rightarrow \textbf{\textit{false}} \\
bd_{sum} & := & \aleph_0
\end{array}
$$

Definition 3.3 (Product MRBNF).

$$
\begin{array}{lcl}
(\gamma_1, \gamma_2)\ F & := & \gamma_1 \times \gamma_2 \\
map_{prod} & := & \lambda f_1\ f_2\ (x, y).(f_1\ x, f_2\ y) \\
set^1_{prod} & := & \lambda(x, y).\{x\} \\
set^2_{prod} & := & \lambda(x, y).\{y\} \\
rel_{prod} & := & \lambda R_1\ R_2\ \ (x_1, x_2)\ (y_1, y_2). \\
 & & \quad R_1\ x_1\ y_1 \wedge R_2\ x_2\ y_2 \\
bd_{prod} & := & \aleph_0
\end{array}
$$

3.3.2 Closure under composition

While MRBNFs are fully defined by Blanchette et al. [9], several other properties like closure under composition and least-fixed point are only proved for select examples. For the rest of section 3.3, our contribution is the algorithms to derive these properties for arbitrary MRBNFs.

MRBNFs can be composed by nesting n inner MRBNFs in the live positions of an outer MRBNF. This property allows the framework to handle a wide range of possible syntaxes generically. As the fixpoint of a datatype is also created in a subset of the live positions of a type constructor (see section 3.3.3), compositionality also allows us to nest recursion through other types as seen in section 2.4.

To make the composition easier, instead of composing arbitrary MRBNFs, it is possible to first normalize them with regard to each other. After normalization, all inner MRBNFs will have the same free, bound and live positions, in the same order. The outer MRBNF will have the same free and bound positions as the inners, also in the same order. This normalization can be achieved with a three step pipeline:

Demoting is the process of bringing positions that are shared between multiple MRBNFs to the same level. The different levels get more specific, from fully unrestricted functions for live positions, to small endofunctions for free positions, to small endobijections for bound positions to no function at all for dead positions. This means we can only *demote* down, i.e., make requirements more specific. After demoting, all positions will be at their lowest common level. To derive a demoted MRBNF, the *map* and *rel* functions are applied to their identities[7] for newly dead positions. In the *rel* function, live positions that are demoted to bound or free are instantiated to the graph of the function. The *set* functions of newly dead positions are removed. The type itself and the bound remain unchanged.

Lifting adds missing positions to a MRBNF. These k new dummy positions are always added to the front of the type constructor. The *map* and *rel* functions are wrapped in a λ-expression with k dummy arguments. The *set* functions of the new positions are the constant function that returns the empty set. The bound does not change.

Permuting changes the order of positions on a MRBNF. For this, the order of arguments in the *map* and *rel* functions are changed as well as the order of *set* functions. Again, the bound does not change.

After these normalization steps (first demoting, then lifting and finally permuting), the actual composition is easier to implement compared to composing arbitrary types directly. For this, the *map* and *rel* functions of the inner MRBNFs are nested in the respective functions of the outer MRBNF. The composed *set* functions are defined as the union of all of the sets returned by the elements of the inner MRBNFs. The bound is the sum of the inner cardinals multiplied with the outer cardinal (see definition 3.4). Recall that we use m as the sum $m_1 + m_2$. Every inner type may have different dead positions $\overline{\delta}$ from any other inner type.

Definition 3.4 (Composed MRBNF)**.** *Given n inner MRBNFs* $\overline{\left(\overline{\alpha}_{m_1}, \overline{\beta}_{m_2}, \overline{\gamma}_{n'}, \overline{\delta}\right) F}$ *and one outer MRBNF* $\left(\overline{\alpha}_{m_1}, \overline{\beta}_{m_2}, \overline{\gamma'}_n, \overline{\delta'}\right) G$*, the composed MRBNF H can be con-*

[7] *id* and (=)

structed as:

$$
\begin{array}{lcl}
\left(\overline{\alpha}, \overline{\beta}, \overline{\gamma}, \overline{\delta}, \overline{\delta'}\right) H & := & \left(\overline{\alpha}, \overline{\beta}, \overline{(\overline{\alpha}, \overline{\beta}, \overline{\gamma}, \overline{\delta})\ F}, \overline{\delta'}\right) G \\
map_H & := & \lambda \overline{f}_m\ \overline{g}_{n'}.\ map_G\ \overline{f}\ \left(map_{F_1}\ \overline{f}\ \overline{g}\right) \ldots \left(map_{F_n}\ \overline{f}\ \overline{g}\right) \\
set_H^{i \leq m} & := & \lambda x.\ set_G^i\ x \cup \bigcup_{y \in set_G^{m+1}\ x} \left(set_{F_1}^i\ y\right) \cup \cdots \cup \bigcup_{y \in set_G^{m+n}\ x} \left(set_{F_n}^i\ y\right) \\
set_H^{m<i} & := & \lambda x.\ \bigcup_{y \in set_G^{m+1}\ x} \left(set_{F_1}^i\ y\right) \cup \cdots \cup \bigcup_{y \in set_G^{m+n}\ x} \left(set_{F_n}^i\ y\right) \\
rel_H & := & \lambda \overline{f}_m\ \overline{R}_{n'}.\ rel_G\ \overline{f}\ \left(rel_{F_1}\ \overline{f}\ \overline{R}\right) \ldots \left(rel_{F_n}\ \overline{f}\ \overline{R}\right) \\
bd_H & := & (bd_{F_1} +_c \cdots +_c\ bd_{F_n}) \times_c\ bd_G
\end{array}
$$

From this "single-step" composition, it is possible to prove that an arbitrary type is a MRBNF with a simple recursive algorithm: if the type is a type variable, return the identity MRBNF (see definition 3.5). If the type is not a MRBNF, return the constant MRBNF (see definition 3.6). Otherwise (if the outer type constructor is already a MRBNF) recursively prove that all arguments of the type constructor are also MRBNFs, normalize and then do a single-step composition.

Definition 3.5 (Identity MRBNF)**.**

$$
\begin{array}{lcl}
\alpha\ F & := & \alpha \\
map_{ID} & := & id : (\alpha \to \alpha) \to \alpha \to \alpha \\
set_{ID} & := & \lambda x.\ \{x\} : \alpha \to \alpha\ set \\
rel_{ID} & := & id : (\alpha \to \alpha \to bool) \\
 & & \quad \to \alpha \to \alpha \to bool \\
bd_{ID} & := & \aleph_0
\end{array}
$$

Definition 3.6 (Constant MRBNF)**.**

$$
\begin{array}{lcl}
\delta\ F & := & \delta \\
map_K & := & id : \delta \to \delta \\
rel_K & := & (=) : \delta \to \delta \to bool \\
bd_K & := & \aleph_0
\end{array}
$$

Together with manual proofs of the MRBNF axioms for the binary sum and product types (as seen in section 3.3.1, definitions 3.2 and 3.3), this allows us to prove that the pre-datatype of an arbitrary syntax forms a MRBNF. The framework also allows the user to register their own types as MRBNF by providing the relevant constants and proving the MRBNF axioms for them. As we will allow recursion through any live position no matter what the concrete type is, these manual registrations make it possible to recurse and bind through non-free datatypes.

3.3.3 Binding fixpoint and α-equivalence

Once a given pre-datatype is a MRBNF, it is used to create a *name-carrying* "raw" datatype. This is a normal, freely constructed datatype using the datatypes already present in Isabelle. It is the least (for datatypes) or greatest (for codatatypes) solution to the induced fixpoint equation on the pre-datatype (see definition 3.8). The properties of MRBNFs guarantee that such a solution always exists.

Definition 3.7 (Raw (co)datatype)**.**

$$\overline{\alpha}\ term_{raw} \simeq (\overline{\alpha}, \overline{\alpha}, \overline{\alpha}\ term_{raw}, \overline{\alpha}\ term_{raw})\ term_{pre}$$

For the example of the untyped λ-calculus from section 3.2, $|\overline{\alpha}| = 1$, so the specific raw datatype has the following shape:

Definition 3.8 (Raw datatype for the untyped λ-calculus)**.**

$$\begin{aligned} &\alpha\ term_{raw} \simeq (\alpha, \alpha, \alpha\ term_{raw}, \alpha\ term_{raw})\ term_{pre} \\ &\quad \textbf{\textit{where}}\ (\alpha, \beta, \gamma, \gamma')\ term_{pre} = \alpha + \gamma \times \gamma + \beta \times \gamma' \end{aligned}$$

From the definition of the datatype we will obtain a constructor and a destructor constant (see definition 3.9). The latter is only used for codatatypes. The destructor and constructor are inverses of each other, i.e., $\mathrm{ctor}_{\mathrm{term\_raw}}(\mathrm{dtor}_{\mathrm{term\_raw}}\ x) = x$.

Definition 3.9 (Constructor and destructor of the raw (co)datatype)**.**

$$\begin{aligned} ctor_{term\_raw} &: (\overline{\alpha}, \overline{\alpha}, \overline{\alpha}\ term_{raw}, \overline{\alpha}\ term_{raw})\ term_{pre} \to \overline{\alpha}\ term_{raw} \\ dtor_{term\_raw} &: \overline{\alpha}\ term_{raw} \to (\overline{\alpha}, \overline{\alpha}, \overline{\alpha}\ term_{raw}, \overline{\alpha}\ term_{raw})\ term_{pre} \end{aligned}$$

In order to define α-equivalence on the datatype, first a function is needed to act on the variables in the type. For this, a primitive (co)recursive function *permute* is defined that applies a function to each variable in the type, no matter if bound or free (see definition 3.10). To prevent capture of variables, this function may only be used with bijections.

Definition 3.10 (Permute functions for least and greatest fixpoint)**.**

$$\begin{aligned}
&\mathit{permute}\ \overline{f}\ (\mathit{ctor}_{\mathit{term_raw}}\ x) = \\
&\quad \mathit{ctor}_{\mathit{term_raw}}\ \Big(\mathit{map}_{\mathit{term_pre}}\ \overline{f}\ \overline{f}\ \Big(\mathit{permute}\ \overline{f}\Big)\ \Big(\mathit{permute}\ \overline{f}\Big)\ x\Big) \\
&\mathit{dtor}_{\mathit{term_raw}}\ \Big(\mathit{permute}\ \overline{f}\ x\Big) \overset{\infty}{=} \\
&\quad \mathit{map}_{\mathit{term_pre}}\ \overline{f}\ \overline{f}\ \Big(\mathit{permute}\ \overline{f}\Big)\ \Big(\mathit{permute}\ \overline{f}\Big)\ (\mathit{dtor}_{\mathit{term_raw}}\ x)
\end{aligned}$$

This function preserves identity and composition if restricted to small bijections. It inherits these properties directly from the map function of the pre-datatype.

The other component needed for α-equivalence is the set of free variables in the type. The set is defined using an auxiliary inductive predicate *free* that specifies when a variable a is free in a term t (see definition 3.11). This is always an inductive predicate even for codatatypes as for a variable to be free, it has to be located at some arbitrary but finite depth in the type. Note that in our presentation, the first m_1 set functions of the pre-datatype will contain the top-level free variables, while the next m_2 set functions contain the top-level bound variables. For the fixpoint construction in particular, we require that $m_1 = m_2$. The function $\mathit{set\_rec}$ contains the recursive occurrences of the type that do not appear under a binder (e.g. the ones in the *App* constructor) and the function $\mathit{set\_brec}$ contains the recursive occurrences that do (i.e., the body of the λ-function).

Definition 3.11 (Free predicate)**.** *For all $i \leq m_1$, the predicate free_i is the smallest predicate closed under the rules:*

$$\frac{a \in \mathit{set}^i_{\mathit{term\_pre}}\ x}{\mathit{free}_i\ a\ (\mathit{ctor}_{\mathit{term\_raw}}\ x)}\ \textsc{Top Free} \qquad \frac{t \in \mathit{set\_rec}_{\mathit{term\_pre}}\ x \qquad \mathit{free}_i\ a\ t}{\mathit{free}_i\ a\ (\mathit{ctor}_{\mathit{term\_raw}}\ x)}\ \textsc{Rec}$$

$$\frac{t \in \mathit{set\_brec}_{\mathit{term\_pre}}\ x \qquad \mathit{free}_i\ a\ t \qquad a \notin \mathit{set}^{m_1+i}_{\mathit{term\_pre}}\ x}{\mathit{free}_i\ a\ (\mathit{ctor}_{\mathit{term\_raw}}\ x)}\ \textsc{BRec}$$

The set of free variables can then be defined as the set of all variables that are free:

Definition 3.12 (Free variables)**.** *For all $i \leq m_1$, $\boldsymbol{FVars}^i_{raw}\ t := \{a.\ \mathit{free}_i\ a\ t\}$*

One important property of the set of free variables is that it is bounded by either the same cardinal bound that the pre-datatype had in the case of a datatype, or the

successor cardinal in the case of a codatatype. After proving a simplification rule for the free variables function (see theorem 3.1), the proof for the datatype follows by straightforward induction.

Theorem 3.1 (Simplification rule for FVars)**.** *For all* $i \leq m_1$*,*

$$FVars^i_{raw}(ctor_{term\_raw}\, x) =$$

$$set^i_{term\_pre}\, x \cup \left(\bigcup_{t \in set\_rec\, x} FVars^i_{raw}\, t\right) \cup \left(\left(\bigcup_{t \in set\_brec\, x} FVars^i_{raw}\, t\right) \setminus set^{m_1+i}_{term\_pre}\, x\right)$$

For codatatypes the proof is a bit more involved. First, we define an auxiliary, primitive recursive function *set_level* that over-approximates the free variables up to a level n (see definition 3.13).

Definition 3.13 (set_level)**.**

$$\begin{aligned} set\_level^i\ 0\, t &= \{\} \\ set\_level^i\ n\, (ctor_{term\_raw}\, x) &= set^i_{term\_pre}\, x \\ &\quad \cup \left(\bigcup_{t \in set\_rec\, x \cup set\_brec\, x} set\_level^i\ (n-1)\, t\right) \end{aligned}$$

For this definition it is trivial to prove (by induction on n) that the function has the same bound as the pre-datatype. Together with a proof that the union for all levels is an over-approximation of the free variables (see theorem 3.2), the free variables have to be bounded by the successor cardinal of the pre-datatype.

Theorem 3.2 (set_level over-approximates FVars)**.**

$$FVars^i_{raw}\, t \subseteq \bigcup_{n=0}^{\infty} set\_level\, n\, t$$

With this, α-equivalence is defined coinductively on the datatype (see definition 3.14). This way, the definition works for both datatypes and codatatypes and all the automation for the proofs and definitions can be shared between the two. The definition makes use of the auxiliary definition $id\_on\ A\, f := \forall a.\, a \in A \longrightarrow f\, a = a$. In general, we use the double bar notation for coinductive definitions compared to the single bar in inductive definitions and theorems.

Definition 3.14 (Alpha equivalence)**.**

$$\frac{\begin{array}{c}\forall i \in \{1 \ldots m_1\}.\ |\boldsymbol{supp}\ f_i| <_o |\alpha_i| \wedge \boldsymbol{bij}\ f_i \\ \wedge\ \boldsymbol{id\_on}\left(\left(\bigcup_{t \in set\_brec\ x} FVars^i_{raw}\ t\right) \setminus set^{m_1+i}_{term\_pre}\ x\right)\ f_i \\ rel_{term\_pre}\ \overline{id}\ \overline{f}\ (\equiv_\alpha)\ \left(\lambda t\ u.\ permute\ \overline{f}\ t \equiv_\alpha u\right)\ x\ y\end{array}}{ctor_{term\_raw}\ x \equiv_\alpha ctor_{term\_raw}\ y}$$

To define the final term quotient type, it is necessary to prove that this relation is an equivalence relation:

Theorem 3.3 (Alpha equivalence is an equivalence relation)**.**

$$\frac{}{x \equiv_\alpha x}\ \textsc{Refl} \qquad \frac{x \equiv_\alpha y}{y \equiv_\alpha x}\ \textsc{Sym} \qquad \frac{x \equiv_\alpha y \qquad y \equiv_\alpha z}{x \equiv_\alpha z}\ \textsc{Trans}$$

While reflexivity and transitivity can be proven by rather straightforward coinduction on the relation, symmetry needs the additional fact that α-equivalent terms have the same free variables (see theorem 3.4) and a theorem that relates α-equivalence with *permute* (see theorem 3.6):

Theorem 3.4 (Free variables of α-equivalent terms)**.**

$$\frac{x \equiv_\alpha y}{\forall i \in \{1 \ldots m_1\}.\ FVars^i_{raw}\ x = FVars^i_{raw}\ y}$$

While we could prove this theorem by induction on x for datatypes, this would not work for codatatypes. To share the proof automation, we use induction on the free variables to prove both directions of the equality separately (see theorem 3.5).

Theorem 3.5 (Auxiliary free variable theorem)**.**

$$\frac{free_i\ a\ x}{\forall y.\ x \equiv_\alpha y \longrightarrow a \in FVars^i_{raw}\ y} \qquad \frac{free_i\ a\ y}{\forall x.\ x \equiv_\alpha y \longrightarrow a \in FVars^i_{raw}\ x}$$

Theorem 3.6 (Alpha-equivalence under permutation)**.**

$$\frac{\begin{array}{c}\forall i \in \{1 \ldots m_1\}.\ |\boldsymbol{supp}\ f_i| <_o |\alpha_i| \wedge \boldsymbol{bij}\ f_i \\ \forall i \in \{1 \ldots m_1\}.\ |\boldsymbol{supp}\ g_i| <_o |\alpha_i| \wedge \boldsymbol{bij}\ g_i \\ \forall i \in \{1 \ldots m_1\}. \forall a \in FVars^i_{raw}\ x.\ f\ a = g\ a \qquad x \equiv_\alpha y\end{array}}{permute\ \overline{f}\ x \equiv_\alpha permute\ \overline{g}\ y}$$

At this point the final term type can be defined as the quotient of the raw term type and α-equivalence. The free variable functions, the permute function and the constructor are lifted to the new quotient using the functions Abs_{term} – that maps a raw term to its quotient representative – and Rep_{term} – that maps a term to a raw term using choice (see definitions 3.15 through 3.18).

Definition 3.15. $\overline{\alpha}\ \boldsymbol{term} := \overline{\alpha}\ term_{raw}\ //\ \{(x, y).\ x \equiv_\alpha y\}$

Definition 3.16. $\boldsymbol{FVars}^i_{term} := \lambda t.\ FVars_{raw}\ (Rep_{term}\ t)$

Definition 3.17. $\boldsymbol{permute}_{term} := \lambda \overline{f}\ t.\ Abs_{term}\ (permute_{raw}\ \overline{f}\ (Rep_{term}\ t))$

Definition 3.18. $\boldsymbol{ctor}_{term} := \lambda x.\ Abs_{term}\ (ctor_{term\_raw}\ (map_{term\_pre}\ \overline{id}\ \overline{Rep_{term}}\ x))$

With the quotient, it is possible to provide some theorems about freshness of bound variables. The first step is to prove that there always exists a choice of bound variable that is disjoint from an arbitrary (bounded) set (see theorem 3.7). With this fact we can prove a strong cases rule that provides a choice of bound variables disjoint from an arbitrary bounded set as well as the free variables (witnessed by the *noclash* predicate[8]) (see theorem 3.8).

Theorem 3.7 (Existence of fresh bound variables)**.**

$$\frac{\forall i \in \{1 \ldots m_1\}.\ |A_i| <_o |\alpha_i|}{\exists y.\ \left(\forall i \in \{1 \ldots m_1\}.\ set^{m_1+i}_{term\_pre}\ y \cap A_i = \{\}\right) \wedge ctor_{term\_raw}\ x \equiv_\alpha ctor_{term\_raw}\ y}$$

Theorem 3.8 (Strong cases)**.**

$$\frac{\begin{array}{c}\forall i \in \{1 \ldots m_1\}.\ |A_i| <_o |\alpha_i| \\ \forall x.\ t = ctor_{term}\ x \longrightarrow (\forall i \in \{1 \ldots m_1\}.\ set^{i+m_1}_{term\_pre}\ x \cap A_i = \{\}) \longrightarrow noclash\ x \longrightarrow P\end{array}}{P}$$

[8] $noclash\ x := \forall i \in \{1 \ldots m_1\}.\ \text{set}^{m_1+i}_{\text{term\_pre}}\ x \cap \left(\text{set}^{i}_{\text{term\_pre}}\ x \cup \bigcup_{t \in \text{set\_rec}\ x} \text{FVars}^i\ t\right) = \{\}$

We will also need a theorem about (quasi-)injectivity. For normal datatypes (e.g. the raw datatype), equality of constructors is the same as equality of the pre-datatypes, i.e., $(\mathrm{ctor}_{\mathrm{term\_raw}}\ x = \mathrm{ctor}_{\mathrm{term\_raw}}\ y) \longleftrightarrow (x = y)$. However, for binding datatypes this is no longer the case. We can only say that they are equal up to permuting of the top-level bound variables (see theorem 3.9).

Theorem 3.9 (Quasi-injectivity of terms)**.**

$$(ctor_{term}\ x = ctor_{term}\ y) \longleftrightarrow \exists \overline{f}. \Bigg(\forall i \in \{1 \dots m_1\}.\ |\boldsymbol{supp}\ f_i| <_o |\alpha_i| \wedge \boldsymbol{bij}\ f_i \wedge\ \boldsymbol{id\_\ on}\ \left(\left(\bigcup_{z \in set_{brec}\ x} FVars^i\ z \right) \setminus set^{i+m_1}\ x \right)\ f_i \Bigg)$$
$$\wedge\ map_{term\_pre}\ \overline{id}\ \overline{f}\ \left(permute\ \overline{f}\right)\ id\ x = y$$

If the bound variables of both sides are already disjoint from arbitrary bounded sets $\overline{A}$, then we can even provide a stronger injection theorem whose bijection also avoids the given set (see theorem 3.10). The difference in the conclusion of the theorem compared with the normal injectivity is highlighted in grey.

Theorem 3.10 (Fresh quasi-injectivity of terms)**.**

$$\frac{\forall i \in \{1 \dots m_1\}.\ |A_i| <_o |\alpha_i| \wedge\ set^{i+m_1}\ x\ \cap\ A_i = \{\} \wedge\ set^{i+m_1}\ y\ \cap\ A_i = \{\}}{(ctor_{term}\ x = ctor_{term}\ y) \longleftrightarrow \exists \overline{f}. \Bigg(\forall i \in \{1 \dots m_1\}.\ |\boldsymbol{supp}\ f_i| <_o |\alpha_i| \wedge \boldsymbol{bij}\ f_i \wedge\ \boldsymbol{id\_\ on}\ \left(\left(\bigcup_{z \in set_{brec}\ x} FVars^i\ z \right) \setminus set^{i+m_1}\ x \right)\ f_i \wedge\ \colorbox{lightgray}{$imsupp\ f_i\ \cap\ A_i = \{\}$} \Bigg)}$$
$$\wedge\ map_{term\_pre}\ \overline{id}\ \overline{f}\ \left(permute\ \overline{f}\right)\ id\ x = y$$

While so far we were able to the same construction for datatypes as well as codatatypes, now these will diverge. We will first show how to derive the strong induction theorem for datatypes: to obtain the necessary "wiggle" room to permute bound variables in between induction steps, an auxiliary *subshape* relation has to be defined:

Definition 3.19 (Subshape)**.**

$$\frac{\forall i \in \{1 \ldots m_1\}.\, |\boldsymbol{supp}\ f_i| <_o |\alpha_i| \wedge \boldsymbol{bij}\ f_i \qquad \overline{permute\ f}\ y \equiv_\alpha z \qquad z \in set\_rec_{term\_pre}\ x \cup set\_brec_{term\_pre}\ x}{y \prec: ctor_{term\_raw}\ x}$$

From this definition follows that any subshape z is also the subshape of all other α-equivalent terms (see theorem 3.11). With this fact we can prove an induction theorem using this subshape definition (see theorem 3.12). The proof first generalizes the theorem to show the property for all possible permutations of x by induction on the raw datatype, then instantiates the renaming to identity.

Theorem 3.11 (Equivalence of subshapes)**.**

$$\frac{z \prec: x \qquad x \equiv_\alpha y}{z \prec: y}$$

Theorem 3.12 (Subshape induction)**.**

$$\frac{\forall y.\, y \prec: x \longrightarrow P\ y}{P\ x}$$

This induction theorems also proves that the subshape relation is a well-founded relation. Finally we can prove the first of the induction theorems, the most general existential one (see theorem 3.13).

Theorem 3.13 (Existential induction)**.**

$$\frac{\begin{array}{l}\rho \in \mathcal{P} \\ \exists y.\, ctor\_term\ y = ctor\_term\ x \\ \quad \wedge \left(\forall z.\, z \in set\_rec_{term\_pre}\ y \cup set\_brec_{term\_pre}\ y \longrightarrow \forall \rho' \in \mathcal{P}.\, P\ z\ \rho'\right) \\ \quad \longrightarrow P\ (ctor\_term\ y)\ \rho\end{array}}{\forall \rho' \in \mathcal{P}.\, P\ t\ \rho'}$$

From this slightly unusual looking theorem, the main induction theorem can be derived using the fresh cases theorem from earlier (see theorem 3.14). Similar to the existential induction theorem, this one explicitly uses some parameter structure $\mathcal{P}$ to contain everything the induction should avoid. With the parameter changing in the induction steps, this allows the user to avoid dynamically changing sets of free variables. If this power is not needed, a straightforward corollary replaces the parameter structure with fixed sets (see theorem 3.15).

Theorem 3.14 (Strong induction)**.**

$$\frac{\begin{array}{c}\forall i \in \{1 \ldots m_1\}.\, \forall \rho.\, \rho \in \mathcal{P} \longrightarrow |K_i\, \rho| <_o |\alpha_i| \\ \forall x\, \rho.\, (\forall z\, \rho'.\, z \in set\_rec\, x \cup set\_brec\, x \longrightarrow \rho' \in \mathcal{P} \longrightarrow P\, z\, \rho') \\ \longrightarrow \left(\forall i \in \{1 \ldots m_1\}.\, set^{i+m_1}\, x \cap K_i\, \rho = \{\}\right) \\ \longrightarrow noclash\, x \longrightarrow \rho \in \mathcal{P} \longrightarrow P\, (ctor\_term\, x)\, \rho\end{array}}{\forall \rho' \in \mathcal{P}.\, P\, t\, \rho'}$$

Theorem 3.15 (Fresh induction)**.**

$$\frac{\begin{array}{ll}\forall i \in \{1 \ldots m_1\}.\, |A_i| <_o |\alpha_i| & \forall x.\, (\forall z.\, z \in set\_rec\, x \cup set\_brec\, x \longrightarrow P\, z) \\ & \longrightarrow \left(\forall i \in \{1 \ldots m_1\}.\, set^{i+m_1}\, x \cap A_i = \{\}\right) \\ & \longrightarrow noclash\, x \longrightarrow P\, (ctor\_term\, x)\end{array}}{P\, t}$$

For codatatypes, deriving a useful coinduction theorem is more involved. The main issue is that the bijections in the normal α-equivalence relation only permute the bound variables of the left operand to the variables on the right. In a coinductive proof, it is useful to be able to work on either side directly. For this reason we define a second coinductive relation for a version of α-equivalence that uses two sets of bijections:

Definition 3.20 (Symmetric α-equivalence)**.**

$$\frac{\begin{array}{l}\forall i \in \{1 \ldots m_1\}.\, |\boldsymbol{supp}\, f_i| <_o |\alpha_i| \wedge \boldsymbol{bij}\, f_i \\ \qquad \wedge \boldsymbol{id\_on} \left(\left(\bigcup_{t \in set\_brec\, x} FVars^i_{raw}\ t\right) \setminus set^{m_1+i}_{term\_pre}\, x\right)\ f_i \\ \forall i \in \{1 \ldots m_1\}.\, |\boldsymbol{supp}\, g_i| <_o |\alpha_i| \wedge \boldsymbol{bij}\, g_i \\ \qquad \wedge \boldsymbol{id\_on} \left(\left(\bigcup_{t \in set\_brec\, y} FVars^i_{raw}\ t\right) \setminus set^{m_1+i}_{term\_pre}\, y\right)\ g_i \\ rel_{term\_pre}\ \overline{id}\ \overline{(g^{-1} \circ f)}\ (\equiv_{\alpha'})\ \left(\lambda t\ u.\, permute\ \overline{f}\ t \equiv_{\alpha'} permute\ \overline{g}\ u\right)\ x\ y\end{array}}{ctor_{term\_raw}\, x \equiv_{\alpha'} ctor_{term\_raw}\, y}$$

Normal α-equivalence obviously implies symmetric α-equivalence by choosing all g_i to be identity. To prove the other direction and thus that both actually define the same relation is trickier. It requires redoing the proofs that relate permutation

with α-equivalence (see theorem 3.16) and one direction of the theorem that states that α-equivalent terms have the same free variables (see theorem 3.17).

Theorem 3.16 (Permutation of α-equivalent terms)**.**

$$\frac{\forall i \in \{1 \ldots m_1\}.\, |\boldsymbol{supp}\ f_i| <_o |\alpha_i| \wedge \boldsymbol{bij}\ f_i \qquad \mathit{permute}\ \overline{f}\ x \equiv_{\alpha'} \mathit{permute}\ \overline{f}\ y}{x \equiv_{\alpha'} y}$$

Theorem 3.17 (Alpha equivalent terms have the same free variables)**.**

$$\frac{\mathit{free}_i\ \ a\ \ x}{\forall y.\, x \equiv_{\alpha'} y \longrightarrow a \in \mathit{FVars}\, y}$$

After proving that both definitions are the same relation, we can use them interchangeably. This means we can now make use of the different coinduction rule associated with the symmetrical version to prove properties about normal α-equivalence. The first theorem that requires this is the existential coinduction theorem (see theorem 3.18)

Theorem 3.18 (Existential coinduction)**.**

$$\frac{\begin{array}{l}\forall x\ y.\, R\ (\mathit{ctor}_{\mathit{term}}\ x)\ (\mathit{ctor}_{\mathit{term}}\ y) \longrightarrow \\ \quad \exists z\ w.\, \mathit{ctor}_{\mathit{term}}\ z = \mathit{ctor}_{\mathit{term}}\ x \wedge \mathit{ctor}_{\mathit{term}}\ w = \mathit{ctor}_{\mathit{term}}\ y \\ \quad \wedge\ \mathit{rel}_{\mathit{term\_pre}}\ \overline{id}\ \ \overline{id}\ \ (\lambda l\ r.\, R\ l\ r \vee l = r)(\lambda l\ r.\, R\ l\ r \vee l = r)\ z\ w\end{array}}{R\ t\ u \longrightarrow t = u}$$

Similar to datatypes, we can also derive another coinduction theorem using the fresh cases theorem (see theorem 3.19).

Theorem 3.19 (Strong coinduction)**.**

$$\frac{\begin{array}{l}\exists \rho \in \mathcal{P}.\, R\ t\ u\ \rho \qquad \forall i \in \{1 \ldots m_1\}.\, \forall \rho.\, \rho \in \mathcal{P} \longrightarrow |K_i\ \rho| <_o |\alpha_i| \\ \forall x\ y\ \rho.\, R\ (\mathit{ctor}_{\mathit{term}}\ x)\ (\mathit{ctor}_{\mathit{term}}\ y)\rho \longrightarrow \\ \quad \big(\forall i \in \{1 \ldots m_1\}.\, \mathit{set}^{i+m_1}\ x\ \cap\ K_i\ \rho = \{\}\big) \longrightarrow \rho \in \mathcal{P} \longrightarrow \\ \quad \mathit{rel}_{\mathit{term\_pre}}\ \overline{id}\ \ \overline{id}\ \ \overline{(\lambda l\ r.\, (\exists \rho' \in \mathcal{P}.R\ l\ r\ \rho') \vee l = r)}\ z\ w\end{array}}{t = u}$$

Given that coinduction can only be used when α-equivalence is the conclusion

of a theorem, it is also helpful to retain access to some form of induction even for codatatypes. In our case, given that free variables are defined inductively, we can lift the induction theorem from the internal *free* predicate (see definition 3.11) to the free variables of the quotient (see theorem 3.20). These theorems of course are also available for normal datatypes, but usually induction on the type itself is much easier to use.

Theorem 3.20 (Induction on the free variables)**.** *For all* $i \leq m_1$:

$$\frac{\begin{array}{c} a \in FVars^i\ t \qquad \forall a\ x.\ a \in set^i\ x \longrightarrow P\ a\ (ctor_{term}\ x) \\ \forall a\ z\ x.\ z \in set\_brec\ x \longrightarrow a \in FVars^i\ z \longrightarrow P\ a\ z \\ \longrightarrow a \notin set^{i+m_1}\ x \longrightarrow P\ a\ (ctor_{term}\ x) \\ \forall a\ z\ x.\ z \in set\_rec\ x \longrightarrow a \in FVars^i\ z \longrightarrow P\ a\ z \longrightarrow P\ a\ (ctor_{term}\ x) \end{array}}{P\ a\ t}$$

However, similar to the normal induction theorem on datatypes, we often need to avoid variables with the induction. Using the fresh cases theorem (theorem 3.8) and the fresh injectivity (see theorem 3.10), we can strengthen the induction theorem:

Theorem 3.21 (Strong induction on the free variables)**.** *For all* $i \leq m_1$:

$$\frac{\begin{array}{c} a \in FVars^i\ t \qquad \forall \rho.\ \rho \in \mathcal{P} \longrightarrow |K\ \rho| <_o |\alpha_i| \\ \forall a\ x\ \rho.\ a \in set^i\ x \longrightarrow \rho \in \mathcal{P} \longrightarrow set^{i+m_1}\ x \cap K\ \rho = \{\} \\ \longrightarrow noclash\ x \longrightarrow P\ a\ (ctor\_term\ x)\ \rho \\ \forall a\ z\ x\ \rho.\ z \in set\_brec\ x \longrightarrow a \in FVars^i\ z \longrightarrow (\forall \rho \in \mathcal{P}.\ P\ a\ z\ \rho) \longrightarrow a \notin set^{i+m_1}\ x \\ \longrightarrow \rho \in \mathcal{P} \longrightarrow set^{i+m_1}\ x \cap K\ \rho = \{\} \longrightarrow noclash\ x \longrightarrow P\ a\ (ctor\_term\ x)\ \rho \\ \forall a\ z\ x\ \rho.\ z \in set\_rec\ x \longrightarrow a \in FVars^i\ z \longrightarrow (\forall \rho \in \mathcal{P}.\ P\ a\ z\ \rho) \\ \longrightarrow \rho \in \mathcal{P} \longrightarrow set^{i+m_1}\ x \cap K\ \rho = \{\} \longrightarrow noclash\ x \longrightarrow P\ a\ (ctor\_term\ x)\ \rho \end{array}}{\forall \rho \in \mathcal{P}.\ P\ a\ t\ \rho}$$

From this we can easily derive a version that avoids a static set A if the full generality of the strong induction is not needed:

Theorem 3.22 (Fresh induction on the free variables)**.** *For all* $i \leq m_1$:

$$\frac{\begin{array}{c} a \in FVars^i\ t \qquad |A| <_o |\alpha_i| \\ \forall a\ x.\ a \in set^i\ x \longrightarrow set^{i+m_1}\ x \cap A = \{\} \longrightarrow noclash\ x \longrightarrow P\ a\ (ctor\_term\ x) \\ \forall a\ z\ x.\ z \in set\_brec\ x \longrightarrow a \in FVars^i\ z \longrightarrow P\ a\ z \longrightarrow a \notin set^{i+m_1}\ x \\ \longrightarrow set^{i+m_1}\ x \cap A = \{\} \longrightarrow noclash\ x \longrightarrow P\ a\ (ctor\_term\ x) \\ \forall a\ z\ x.\ z \in set\_rec\ x \longrightarrow a \in FVars^i\ z \longrightarrow P\ a\ z \\ \longrightarrow set^{i+m_1}\ x \cap A = \{\} \longrightarrow noclash\ x \longrightarrow P\ a\ (ctor\_term\ x) \end{array}}{P\ a\ t}$$

3.3.4 Binding-aware recursor

To define functions on the binder datatypes, the framework provides a binding-aware recursor. It is implemented using *locales*, Isabelle's module system. A locale allows us to fix some constants and assume some axioms about them in a local context. Within this context, new theorems can be proven based on the local axioms. A user can then later instantiate a locale by providing the constants and proving the axioms for them. They then have access to all the derived theorems and definitions.

The recursor in particular is available as three distinct locales. The main one – where the actual construction of the recursive function takes place (see definition 3.21) – and two derived locales that fix some of the constants (see definitions 3.29 and 3.30). The axioms make use of the auxiliary definitions $\text{pred}_{\text{term\_pre}} \overline{P} x :=$ $\text{rel}_{\text{term\_pre}}\ \overline{\text{id}}\ \overline{(\lambda y\ \_.\ P\ y)}\ x\ x$ and $\text{pred}_{\text{fun}}\ P\ Q\ f := \forall x.\ P\ x \longrightarrow Q\ (f\ x)$.

Definition 3.21 (Recursor locale)**.** *For a given type* $\overline{\alpha}$ *term, with* ρ *and* υ *as local type parameters on the locale, the recursor fixes:*

$Pmap$	:	$\overline{(\alpha \to \alpha)} \to \rho \to \rho$	
$\overline{PFVars}$	:	$\rho \to \alpha_i\ set$	***for*** $i \in \{1 \ldots m_1\}$
$validP$	:	$\rho \to \textbf{\textit{bool}}$	
$\overline{A}$	:	$\alpha_i\ set$	***for*** $i \in \{1 \ldots m_1\}$
$Umap$	:	$\overline{(\alpha \to \alpha)} \to \overline{\alpha}\ term \to \upsilon \to \upsilon$	
$\overline{UFVars}$	:	$\overline{\alpha}\ term \to \upsilon \to \alpha_i\ set$	***for*** $i \in \{1 \ldots m_1\}$

$$\begin{aligned} \mathit{validU} &: \upsilon \to \textbf{\textit{bool}} \\ \mathit{Uctor} &: \left(\overline{\alpha}, \overline{\alpha}, \overline{\alpha}\ \mathit{term} \times (\rho \to \upsilon), \overline{\alpha}\ \mathit{term} \times (\rho \to \upsilon)\right)\ \mathit{term\_pre} \to \rho \to \upsilon \end{aligned}$$

such that the recursor axioms hold. Then, there exists a recursive function $\mathit{REC} :: \overline{\alpha}\ \mathit{term} \to \rho \to \upsilon$ *that is characterized by theorems 3.28 through 3.30.*

The first set of fixed constants forms the *parameter structure*. It is composed of a dynamic part ρ and a list of static sets $\overline{A}$. The dynamic part needs an operator to apply bijections to all variables in ρ and one to extract its free variables. Additionally a user-chosen predicate *validP* allows the user to define the function only on a subset of the type without the need to introduce explicit subtypes. The parameter structure is meant to contain all the variables in the domain of the resulting function that the recursor needs to avoid. While it would have been possible to omit the static sets by instantiating ρ with a singleton type, explicitly carrying them makes the construction easier to use. As many functions do not need a dynamically changing set of variables to be avoided, the first derived locale (see definition 3.29) instantiates ρ to unit and only leaves the sets $\overline{A}$ for the user.

The second set of constants are the *model* of the function. It also has operators to apply bijections and extract free variables, similar to the parameter structure. Again, it is possible to define the function only on a subset of codomain by supplying a predicate *validU*. The main constant of the locale however is *Uctor* that we call the *blueprint* of the function. Similar to a recursion scheme, it defines the reduction of a single level of the datatype that is then iterated to form the final function. For this, the recursive positions contain functions that can be used to obtain the reduced value of the subcomponents as well as their unreduced terms. For a detailed example on how to instantiate this constant see section 3.3.5.

Axiom 1 (Pmap_comp).

$$\frac{\begin{array}{c} \forall i \in \{1 \ldots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \wedge\ \mathbf{bij}\ f_i \\ \forall i \in \{1 \ldots m_1\}.\ |\mathbf{supp}\ g_i| <_o |\alpha_i| \wedge\ \mathbf{bij}\ g_i \qquad \mathrm{validP}\ \rho \end{array}}{\mathrm{Pmap}\ \overline{f}\ (\mathrm{Pmap}\ \overline{g}\ \rho) = \mathrm{Pmap}\ \overline{(f \circ g)}\ \rho}$$

Similar to the composition axiom of MRBNFs, axiom 1 ensures that the *Pmap* operator preserves composition of functions. The corresponding identity axiom is omitted on purpose, as it can be derived from axiom 2. As mentioned earlier, in all of the axioms the *Pmap* operator only works with bijections on valid parameters.

Axiom 2 (Pmap_cong_id).

$$\frac{\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \wedge \mathbf{bij}\ f_i \qquad \forall i \in \{1 \dots m_1\}.\ \forall a.\ a \in \mathrm{PFVars}_i\ \rho \longrightarrow f_i\ a = a \qquad \mathrm{validP}\ \rho}{\mathrm{Pmap}\ \overline{f}\ \rho = \rho}$$

Axiom 3 (PFVars_Pmap).

$$\frac{\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \wedge \mathbf{bij}\ f_i \qquad \mathrm{validP}\ \rho}{\forall i \in \{1 \dots m_1\}.\ \mathrm{PFVars}_i \left(\mathrm{Pmap}\ \overline{f}\ \rho\right) = \{f_i\ x \mid x.\ x \in \mathrm{PFVars}_i\ \rho\}}$$

Again similar to MRBNFs, axioms 2 and 3 define the interaction of the mapping and free variable operators. One difference is that *Pmap_cong_id* only requires congruence with regard to identity, not arbitrary functions.

Axiom 4 (small_PFVars).

$$\frac{\mathrm{validP}\ \rho}{\forall i \in \{1 \dots m_1\}.\ |\mathrm{PFVars}_i\ \rho| <_o |\alpha_i| \wedge |A_i| <_o |\alpha_i|}$$

As the parameters contain all the variables that the recursor should avoid, axiom 4 ensures that the sets have a smaller cardinality than the type used as variables. Thus there are always new fresh variables available.

Axiom 5 (valid_Pmap).

$$\frac{\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \wedge \mathbf{bij}\ f_i \qquad \mathrm{validP}\ \rho}{\mathrm{validP}\ \left(\mathrm{Pmap}\ \overline{f}\ \rho\right)}$$

Axiom 5 states that *Pmap* can not leave the valid fragment of the parameters. In other words, the validity predicate is closed under mapping.

Axiom 6 (Umap_comp).

$$\frac{\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \wedge \mathbf{bij}\ f_i \qquad \forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ g_i| <_o |\alpha_i| \wedge \mathbf{bij}\ g_i \qquad \mathrm{validU}\ \upsilon}{\mathrm{Umap}\ \overline{f}\ t\ (\mathrm{Umap}\ \overline{g}\ t\ \upsilon) = \mathrm{Umap}\ \overline{(f \circ g)}\ t\ \upsilon}$$

Axiom 7 (Umap_cong_id).

$$\frac{\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \wedge \mathbf{bij}\ f_i \qquad \forall i \in \{1 \dots m_1\}.\ \forall a.\ a \in \mathrm{UFVars}_i\ t\ \upsilon \longrightarrow f_i\ a = a \qquad \mathrm{validU}\ \upsilon}{\mathrm{Umap}\ \overline{f}\ t\ \upsilon = \upsilon}$$

Similar to the parameter structure, axioms 6 and 7 require the *Umap* operator to preserve composition and congruence with regard to identity.

Axiom 8 (Umap_Uctor).

$$\frac{\begin{array}{c}\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \wedge\ \mathbf{bij}\ f_i \qquad \mathrm{validP}\ \rho \\ \mathrm{pred}_{\mathrm{term\_pre}}\ (\mathrm{pred}_{\mathrm{fun}}\ \mathrm{validP}\ \ \mathrm{validU} \circ \mathbf{snd})\ (\mathrm{pred}_{\mathrm{fun}}\ \mathrm{validP}\ \ \mathrm{validU} \circ \mathbf{snd})\ y \\ \forall i \in \{1 \dots m_1\}.\ \mathrm{imsupp}\ f_i\ \cap\ A_i = \{\}\end{array}}{\begin{array}{l}\mathrm{Umap}\ \overline{f}\ \left(\mathrm{ctor\_term}\ \left(\mathrm{map}_{\mathrm{term\_pre}}\ \overline{\mathrm{id}}\ \overline{\mathrm{id}}\ \ \mathbf{fst}\ \ \mathbf{fst}\ y\right)\right)\ (\mathrm{Uctor}\ y\ \rho) = \mathrm{Uctor} \\ \quad \left(\mathrm{map}_{\mathrm{term\_pre}}\ \overline{f}\ \overline{f}\ \overline{\left(\lambda(t, g).\ \left(\mathrm{permute}\ \overline{f}\ t, \mathrm{Umap}\ \overline{f}\ t\ (g\ (\mathrm{Pmap}\ \overline{f^{-1}}\ \rho))\right)\right)}\ y\right) \\ \quad \left(\mathrm{Pmap}\ \overline{f}\ \rho\right)\end{array}}$$

The *Umap* operator has to act as a permutation action on our codomain type. Axiom 8 ensures that this operator commutes with the blueprint of the function. In many cases – those where the codomain is the same as the domain – this operator will just be the standard permutation operator from section 3.3.3. See our second case study in section 6.3 for a more complex choice for *Umap*.

Axiom 9 (UFVars_subset).

$$\frac{\begin{array}{c}\mathrm{validP}\ \rho \\ \mathrm{pred}_{\mathrm{term\_pre}}\ (\mathrm{pred}_{\mathrm{fun}}\ \mathrm{validP}\ \ \mathrm{validU} \circ \mathbf{snd})\ (\mathrm{pred}_{\mathrm{fun}}\ \mathrm{validP}\ \ \mathrm{validU} \circ \mathbf{snd})\ y \\ \forall i \in \{1 \dots m_1\}.\ \forall t\ f\ p.\ \mathrm{validP}\ p \longrightarrow (t, f) \in \mathrm{set\_brec}\ y \cup \mathrm{set\_rec}\ y \\ \longrightarrow \mathrm{UFVars}_i\ t\ (f\ p) \subseteq \mathrm{FVars}\ t \cup \mathrm{PFVars}_i\ p \cup A_i\end{array}}{\begin{array}{l}\forall i \in \{1 \dots m_1\}.\ \mathrm{UFVars}_i\ \left(\mathrm{ctor\_term}\ \left(\mathrm{map}_{\mathrm{term\_pre}}\ \overline{\mathrm{id}}\ \mathrm{fst}\ \mathrm{fst}\ y\right)\right)\ (\mathrm{Uctor}\ y\ \rho) \\ \quad \subseteq \mathrm{FVars}\ \left(\mathrm{ctor\_term}\ \left(\mathrm{map}_{\mathrm{term\_pre}}\ \overline{\mathrm{id}}\ \mathrm{fst}\ \mathrm{fst}\ y\right)\right) \cup \mathrm{PFVars}_i\ \rho \cup A_i\end{array}}$$

As the recursor needs to avoid all free variables – be it in the term itself or in the parameter structure – we need to be sure that those are the only free variables. Axioms 9 requires that the free variables of the blueprint are a subset of the free variables of the original term and the parameter structure.

Axiom 10 (valid_Umap).

$$\frac{\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \wedge\ \mathbf{bij}\ f_i \qquad \mathrm{validU}\ \upsilon}{\mathrm{validU}\ \left(\mathrm{Umap}\ \overline{f}\ t\ \upsilon\right)}$$

Axiom 11 (valid_Uctor).

$$\frac{\mathrm{pred}_{\mathrm{term\_pre}}\ (\mathrm{pred}_{\mathrm{fun}}\ \mathrm{validP}\ \ \mathrm{validU} \circ \mathbf{snd})\ (\mathrm{pred}_{\mathrm{fun}}\ \mathrm{validP}\ \ \mathrm{validU} \circ \mathbf{snd})\ y \quad \mathrm{validP}\ \rho}{\mathrm{validU}\ \ (\mathrm{Uctor}\ y\ \rho)}$$

Again, similar to the parameter structure, axioms 10 and 11 ensure that the validity predicate is closed under the relevant operators. This way we can never leave the valid subset of the codomain.

To construct the final recursive function *REC*, first some auxiliary definitions in the locale context are needed. The first three lower the *Umap*, *UFVars* and *Uctor* operators to work on the raw datatype instead of the quotient term type. Here $\mathrm{Abs}_{\mathrm{term}}$ is the function that maps every raw term to its representative in the quotient type.

Definition 3.22. ***raw_Umap*** $\overline{f}\ x := Umap\ \overline{f}\ \ (Abs_{term}\ x)$

Definition 3.23. ***raw_UFVars*** $x := UFVars\ \ (Abs_{term}\ x)$

Definition 3.24. ***raw_Uctor*** $y := Uctor\ \left(map_{term\_pre}\ \overline{id}\ \overline{\left(map_{prod}\ \ Abs_{term}\ \ id\right)}\ y\right)$

To permute bound variables "out of the way" of the parameter structure, the recursor will need to *pick* a bijection that is *suitable*. In this case this means a function that takes a raw datatype and a parameter structure and returns a bijection (recall the definition of $imsupp\ f := \bigcup\{\{a, f\ a\} \mid a.\ f\ a \neq a\}$):

Definition 3.25 (suitable)**.** ***For*** $i \in \{1 \ldots m_1\}$*:*

$$\begin{aligned}
&\boldsymbol{suitable}_i\ \ pick := \forall x\ p.\ validP\ p \longrightarrow \\
&\quad \boldsymbol{bij}\ (pick\ x\ p) \wedge |\boldsymbol{supp}\ (pick\ x\ p)| <_o |\alpha_i| \wedge \\
&\quad \boldsymbol{imsupp}\ (pick\ x\ p) \cap ((FVars_{raw}\ (ctor_{raw}\ x) \cup PFVars_i\ p \cup A_i) \setminus set^{i+m_1}\ x) = \{\} \\
&\quad \wedge \{pick\ x\ p\ a \mid a.\ a \in set^{i+m_1}\ x\} \cap (FVars_{raw}\ (ctorraw\ x) \cup PFVars_i\ p \cup A_i) = \{\}
\end{aligned}$$

Intuitively, this definition says that the function returned by *pick* has the following properties:

- it is a small-support bijection
- it is not touching any of the variables in the term or parameter structure aside from the top-level bound ones

- applying the function to the bound variables makes them disjoint from the variables of the term and parameter structure

Using this definition we can then define the actual recursor. Given that we will need to permute the bound variables at every step of the recursion, this function needs to use well-founded recursion, not primitive recursion.

Definition 3.26 (Raw recursor)**.**

$$\begin{aligned}&\boldsymbol{H}\,\overline{pick}\,(ctor\_raw\_term\,x)\,p = \boldsymbol{if}\ \forall i \in \{1 \ldots m_1\}.\ suitable_i\ \ pick_i\ \ \boldsymbol{then}\\ &raw\_Uctor\ \Big(map_{term\_pre}\ \overline{id}\ \overline{(pick\,x\,p)}\\ &\quad\Big(\lambda t.\ \Big(t, H\ \overline{pick}\ t\Big)\Big)\ \Big(\lambda t.\ \Big(permute\ \overline{(pick\,x\,p)}\ t, H\ \overline{pick}\ \Big(permute\ \overline{(pick\,x\,p)}\ t\Big)\Big)\Big)\\ &\quad x\Big)\ p\ \ \boldsymbol{else}\ \ undefined\end{aligned}$$

To prove termination of this function, we can reuse the *subshape* relation from earlier. It was defined with exactly this use case in mind: primitive recursion up to permuting of bound variables. We already know that this relation is well-founded from section 3.3.3.

Additionally we can already define the final function *REC* which will be the raw recursor lifted to the quotient type. For this we obtain a suitable picking function using Hilbert's choice operator. The existence of such functions can be proven using the smallness of all sets involved. Thus it is always possible to obtain variables that are fresh.

Definition 3.27 (pick0)**.** ***For*** $i \in \{1 \ldots m_1\}$: $\boldsymbol{pick0}_i := \epsilon\ suitable_i$

Definition 3.28 (Recursor)**.** $\boldsymbol{REC}\,t := H\ \overline{pick0}\,(Rep_{term}\,t)$

The rest of the construction will be concerned with proving that the raw recursor H is well behaved with regard to α-equivalence and that it is invariant under the exact choice of suitable pick functions.

The first step is to lower all of the axioms of the locale to work on the raw datatype and the raw operators defined earlier (definitions 3.22, 3.23 and 3.24). These proofs follow from the properties of the *Abs* and *Rep* functions (theorems 3.23 and 3.24), as well as the compatibility of α-equivalence and permutation (theorem 3.6).

Theorem 3.23 (Abs_inverse)**.**

$$alpha_{term}\ (Rep_{term}\ (Abs_{term}\ x))\ x$$

Theorem 3.24 (Rep_inverse)**.**

$$Abs_{term}\ (Rep_{term}\ x) = x$$

The fact that the recursor does not introduce any new free variables (see theorem 3.25) follows directly by subshape induction and the UFVars_subset axiom of the locale.

Theorem 3.25 (Free variables of raw recursor)**.**

$$\frac{validP\ p \qquad \forall i \in \{1 \ldots m_1\}.\ suitable_i\ \ pick_i}{\forall i \in \{1 \ldots m_1\}.\ UFVars_i\ t\ \left(H'\ \overline{pick}\ t\ p\right) \subseteq FVars_i\ t \cup PFVars_i\ p \cup A_i}$$

The main theorem of the construction proves three facts in tandem: the raw recursor commutes with permutation, it returns the same result for α-equivalent terms and it is invariant under the specific choice of pick functions (see theorem 3.26). The theorem uses the auxiliary definition *raw_PUmap*[9].

Theorem 3.26 (Swapping, α-equivalence and pick-irrelevance of the raw recursor)**.**

$$\frac{\begin{array}{c} validP\ p \qquad \forall i \in \{1 \ldots m_1\}.\ suitable_i\ \ pick_i \wedge\ suitable_i\ \ pick'_i \\ \forall i \in \{1 \ldots m_1\}.\ |\boldsymbol{supp}\ f_i| <_o |\alpha_i| \wedge\ \boldsymbol{bij}\ f_i \\ \forall i \in \{1 \ldots m_1\}.\ imsupp\ f_i \cap A_i = \{\} \qquad t \equiv_\alpha t' \end{array}}{\begin{array}{l} H\ \overline{pick}\ \left(permute\ \overline{f}\ t\right)\ p = raw\_PUmap\ \overline{f}\ t\ \left(H\ \overline{pick}\ t\right)\ p \\ \wedge\ H\ \overline{pick}\ t\ p = H\ \overline{pick'}\ t'\ p \end{array}}$$

The proof uses the subshape induction theorem from section 3.3.3 (theorem 3.12). For the first fact, the proof applies the definition of the raw recursor to step both invocations of the raw recusor on the two sides of the equality. Using the Umap_Uctor axiom on the right side and further simplifying the goal will result in an equality between two Uctor applications. It is then possible to obtain bijections that are the identity on the free variables and permute the bound variables appropriately. In the recursive position the second part of the induction hypothesis can be used to prove the equality between α-equivalent terms involving the raw recursor. The proof of the second fact is similar except that we also have to deal with the bijections that come from the α-equivalence of t and t'.

[9] **raw_PUmap** $\overline{f}\ t := \lambda g\ p.\ \text{raw\_Umap}\ \overline{f}\ t\ \left(g\ \left(\text{Pmap}\ \overline{f^{-1}}\ p\right)\right)$

The last major step in the construction is to hide the permuting of bound variables on the top-level that would be implied by the definition of the raw recursor. By assuming that the top-level bound variables are disjoint from the parameter structure, we can use the identity function as *pick* function on the top-level and only use *pick0* in the recursive parts. Thanks to the pick-irrelevance proved in the previous theorem, changing the pick functions does not change the result of the recursor. We end up with a theorem that looks more like a normal simplification rule (see theorem 3.27).

Theorem 3.27 (Hiding of permution)**.**

$$\frac{validP\, p \qquad \forall i \in \{1 \dots m_1\}.\ set^{m_1+i}\, x \cap (PFVars_i\, p \cup A_i) = \{\} \qquad noclash\, x}{\begin{array}{l} H\ \overline{pick0}\ (ctor\_raw\_term\, x)\ p = \\ \quad raw\_Uctor\left(map_{term\_pre}\ \overline{id}\ \overline{\left(\lambda t.\ \left(t, H\ \overline{pick0}\ t\right)\right)}\ x\right)\ p \end{array}}$$

To get the final result theorems of the locale, the relevant theorem are lifted to the original constants and the quotient terms. The proofs are straightforward thanks to the equality under α-equivalence of the raw recursor.

Theorem 3.28. *REC_ctor*

$$\frac{validP\, \rho \qquad \forall i \in \{1 \dots m_1\}.\ set^{i+m_1}\, x \cap (PFVars_i\, \rho \cup A_i) = \{\} \qquad noclash\, x}{\begin{array}{l} REC\,(ctor\_term\, x)\ \rho = \\ \quad Uctor\left(map_{term\_pre}\ \overline{id}\ \ (\lambda t.\ (t, REC\, t))\ \ (\lambda t.\ (t, REC\, t))\ \ x\right)\ \rho \end{array}}$$

Theorem 3.28 is the main result theorem of the recursor. It acts as the simplification rule for the final function *REC* by providing an unfolding in terms of the function blueprint *Uctor*. Note that this unfolding is only defined if the choice of bound variables in the term neither clashes with its own free variables nor with the free variables of the parameter structure. In practice, this is not an issue because the strong cases and induction theorems provide exactly this disjointness.

Theorem 3.29. *REC_swap*

$$\frac{\begin{array}{c} validP\, \rho \\ \forall i \in \{1 \dots m_1\}.\ |\boldsymbol{supp}\ f_i| <_o |\alpha_i| \wedge\ \boldsymbol{bij}\ f_i \qquad \forall i \in \{1 \dots m_1\}.\ imsupp\, f \cap A_i = \{\} \end{array}}{REC\,(permute\ \overline{f}\ x)\ \rho = Umap\ \overline{f}\ x\left(REC\, x\ \left(Pmap\ \overline{f^{-1}}\ \rho\right)\right)\ \rho}$$

Theorem 3.30. *REC_FVars*

$$\frac{validP\ \rho}{\forall i \in \{1 \ldots m_1\}.\ UFVars_i\ x\ (REC\ x\ \rho) \subseteq FVars\ x \cup PFVars_i\ \rho \cup A_i}$$

Theorems 3.29 and 3.30 are the direct extensions of the axioms *Umap_Uctor* and *UFVars_subset* to the final function. Together with the function *REC* itself, theorems 3.28 through 3.30 are the "output" of the locale, i.e., the theorems the user has access to.

For the cases where the full generality of the recursor is not needed, we also define two derived locales that partially instantiate the recursor. The first locale removes all of the constants of the parameter structure aside from the static sets $\overline{A}$ (see definition 3.29). This is for the common case of functions that do not have a dynamically changing set of variables to avoid.

Definition 3.29 (Static recursor locale)**.** *For a given type* $\overline{\alpha}$ *term, with* υ *as local type parameter on the locale, the static recursor fixes:*

$$\begin{array}{lcl}
\overline{A} & : & \alpha_i\ set \quad \textbf{\textit{for}}\ i \in \{1 \ldots m_1\} \\
Umap & : & \overline{(\alpha \to \alpha)} \to \overline{\alpha}\ term \to \upsilon \to \upsilon \\
\overline{UFVars} & : & \overline{\alpha}\ term \to \upsilon \to \alpha_i\ set \quad \textbf{\textit{for}}\ i \in \{1 \ldots m_1\} \\
validU & : & \upsilon \to \textbf{\textit{bool}} \\
Uctor & : & \left(\overline{\alpha}, \overline{\alpha}, \overline{\alpha}\ term \times \upsilon, \overline{\alpha}\ term \times \upsilon\right)\ term_pre \to \upsilon
\end{array}$$

The function REC can be obtained by instantiating the general recursor:

$$\begin{array}{lcl}
\rho & := & unit \\
Pmap & := & \lambda \overline{f}\ x.\ () \\
\overline{PFVars} & := & \lambda x.\ \{\} \\
validP & := & \lambda x.\ \textbf{\textit{True}} \\
Uctor & := & \lambda y\ \rho.\ Uctor\ \left(map_term_pre\ \overline{id}\ \overline{(\lambda(t, f).\ (t, f()))}\ y\right)
\end{array}$$

The third locale further instantiates the static locale to fix the codomain type to the term type itself (see definition 3.30). This is the locale used to define functions from terms to terms such as renaming (see section 3.3.5).

Definition 3.30 (Term recursor locale)**.** *For a given type* $\overline{\alpha}$ *term, the term recursor*

fixes:

$$\begin{array}{lcl} \overline{A} & : & \alpha_i \; set \quad \textbf{\textit{for}}\, i \in \{1 \ldots m_1\} \\ Uctor & : & \left(\overline{\alpha}, \overline{\alpha}, \overline{\alpha}\; term \times \overline{\alpha}\; term, \overline{\alpha}\; term \times \overline{\alpha}\; term\right)\; term\_pre \to \overline{\alpha}\; term \end{array}$$

The function REC can be obtained by instantiating the static recursor:

$$\begin{array}{lcl} \upsilon & := & \overline{\alpha}\; term \\ Umap & := & \lambda \overline{f}\; \_\; t.\; permute\; \overline{f}\; t \\ \overline{UFVars} & := & \overline{\lambda \_ .\; FVars} \\ validU & := & \lambda t.\; \textbf{\textit{True}} \end{array}$$

3.3.5 Renaming in datatypes

To prove that the quotient term type itself is again a MRBNF, we need a *map* function. While *permute* looks like a suitable candidate at first glance, it requires bijections for all variable arguments while a MRBNF only requires small endofunctions for free positions.

To fix this, we can define a renaming function using our recursor. More precisely, we will use the third of the recursor locales that only leaves the *Uctor* and the static sets $\overline{A}$ uninstantiated.

First we open a new nested context. In Isabelle, these contexts allow us to fix variables with assumptions. All constants defined in the context will have these variables as arguments outside of the context while theorems will have the assumptions as extra premises. For renaming, we will fix the functions $\overline{f}$ with the assumptions that each of them has small support.

The term recursor can then be instantiated with $A_i := \text{imsupp}\, f_i$ and $\text{Uctor}\, x := \text{ctor\_term}\left(\text{map}_{\text{term\_pre}}\, \overline{f}\, \text{snd}\, \text{snd}\, x\right)$. Smallness of the sets $\overline{A}$ follows directly from the small support assumptions on the context. The axioms *UFVars_subset* and *Umap_Uctor* also only require some simplification. After proving the axioms the nested context can be closed.

The resulting function will have the type $\overline{(f \to f)} \to \overline{\alpha}\, \text{term} \to \overline{\alpha}\, \text{term}$ outside of the context as it has gained the fixed variables as new arguments. To use this renaming function as the *map* function of a MRBNF, several theorems about its interactions with other constants (e.g. *FVars*) are needed (see section 3.3.1).

First, we prove that for bijections, renaming and permuting coincide (see theorem 3.31) using straight forward induction with the fresh induction theorem (see theorem 3.15) to avoid the variables touched by $\overline{f}$. Given that the permute operator respects identity, renaming respects identity as a corollary (see theorem 3.32).

Theorem 3.31 (Renaming and permuting coincide)**.**

$$\frac{\forall i \in \{1 \ldots m_1\}.\ |\boldsymbol{supp}\ f_i| <_o |\alpha_i| \wedge\ \boldsymbol{bij}\ f_i}{\mathit{rename}\ \overline{f} = \mathit{permute}\ \overline{f}}$$

Theorem 3.32 (Renaming preserves identity)**.**

$$\frac{}{\mathit{rename}\ \overline{\mathit{id}} = \mathit{id}}$$

The other MRBNF axioms like preservation of composition and congruence follow similar straightforward inductive proofs.

3.3.6 Binding-aware corecursor

For binding codatatypes, we want to define a binding-aware corecursor. Similar to the recursor it is also implemented as a locale (see definition 3.31). However, given that the corecursor is quite a bit simpler in its definition, we do not need any additional locales that fix part of the constants. The main reason for this simplicity is that for the corecursor the user-chosen type sits in the domain of the function. This means there is no need for a separate parameter structure. Also, instead of dissecting terms – where permuting of bound variables to ensure freshness is needed – the corecursor only needs to construct a term with an arbitrary choice of bound variables. In the following definition, $\mathrm{case}_{\mathrm{sum}}$ is the eliminator for the binary sum type and $\mathrm{map}_{\mathrm{sum}}$ its map function.

Definition 3.31 (Corecursor locale)**.** *For a given type $\overline{\alpha}$ term, with υ as local type*

parameters on the locale, the corecursor fixes:

$$\begin{array}{lcl} \mathit{Umap} & : & \overline{(\alpha \to \alpha)} \to \upsilon \to \upsilon \\ \overline{\mathit{UFVars}} & : & \upsilon \to \alpha_i \; \mathit{set} \quad \textbf{\textit{for}}\, i \in \{1 \ldots m_1\} \\ \mathit{validU} & : & \upsilon \to \textbf{\textit{bool}} \\ \mathit{Udtor} & : & \upsilon \to \left(\overline{\alpha}, \overline{\alpha}, \overline{\alpha}\; \mathit{term} + \upsilon, \overline{\alpha}\; \mathit{term} + \upsilon\right)\; \mathit{term\_pre}\; \mathit{set} \end{array}$$

such that the corecursor axioms hold. Then there exists a function COREC that is characterized by theorems 3.36 through 3.38.

Similar to the recursor, the corecursor fixes a permutation action *Umap* that applies a given bijection to the domain type, as well as operators to extract the free variables from it. To define the function only on a subset of the domain, the validity predicate *validU* can be used.

The corecursor blueprint *Udtor* is non-deterministic, i.e., it returns a set of pre-terms instead of a single pre-term. The axioms will ensure that all members of the set will be equal up to α-equivalence. In practice, this will usually be all the α-equivalent terms that avoid a certain set of variables (see section 3.3.7 for a concrete example).

Axiom 1 (Umap_comp).

$$\frac{\begin{array}{c} \forall i \in \{1 \ldots m_1\}.\, |\mathbf{supp}\; f_i| <_o |\alpha_i| \wedge\; \mathbf{bij}\; f_i \\ \forall i \in \{1 \ldots m_1\}.\, |\mathbf{supp}\; g_i| <_o |\alpha_i| \wedge\; \mathbf{bij}\; g_i \qquad \mathrm{validU}\; u \end{array}}{\mathrm{Umap}\; \overline{f}\; (\mathrm{Umap}\; \overline{g}\; u) = \mathrm{Umap}\; \overline{(f \circ g)}\; u}$$

Axiom 2 (Umap_cong_id).

$$\frac{\begin{array}{c} \forall i \in \{1 \ldots m_1\}.\, |\mathbf{supp}\; f_i| <_o |\alpha_i| \wedge\; \mathbf{bij}\; f_i \\ \forall i \in \{1 \ldots m_1\}.\, \forall a.\, a \in \mathrm{UFVars}_i\; u \longrightarrow f_i\; a = a \qquad \mathrm{validU}\; u \end{array}}{\mathrm{Umap}\; \overline{f}\; u = u}$$

Axioms 1 and 2 require the *Umap* operator to preserve composition and congruence with regard to identity, similar to the recursor. As a permutation action, this operator only accepts small bijections.

Axiom 3 (Udtor_ne).

$$\frac{\mathrm{validU}\; u}{\mathrm{Udtor}\; u \neq \{\}}$$

Axiom 4 (alpha_Udtor).

$$\frac{\text{validU } u \qquad \{X, X'\} \subseteq \text{Udtor } u}{\exists \overline{f}.\ \Bigg(\forall i \in \{1 \ldots m_1\}.\ \mathbf{bij}\ f_i \wedge |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \wedge\ \mathbf{id\_on}\ \Bigg(\Bigg(\bigcup_{z \in \text{set\_brec } X} \text{case}_{\text{sum}}\ \text{FVars}_i\ \text{UFVars}_i\ z\Bigg) \setminus \text{set}^{i+m_1}\ X\Bigg)\ f_i\Bigg) \\ \wedge\ \text{map}_{\text{term\_pre}}\ \overline{\text{id}}\ \overline{f}\ \Big(\text{map}_{\text{sum}}\ \Big(\text{permute}\ \overline{f}\Big)\ \Big(\text{Umap}\ \overline{f}\Big)\Big)\ \text{id}\ \ X = X'}$$

As mentioned previously, all elements in the set returned by the function blueprint *Udtor* have to be α-equivalent (see axiom 4). Furthermore, axiom 4 ensures the set is not empty for valid inputs.

Axiom 5 (UFVars_Udtor).

$$\frac{\text{validU } u \qquad X \in \text{Udtor } u}{\forall i \in \{1 \ldots m_1\}.\ \text{set}^i\ X \cup \Bigg(\bigcup_{z \in \text{set\_rec } X} \text{case}_{\text{sum}}\ \text{FVars}_i\ \text{UFVars}_i\ z\Bigg) \\ \cup \Bigg(\bigcup_{z \in \text{set\_brec } X} \text{case}_{\text{sum}}\ \text{FVars}_i\ \text{UFVars}_i\ z\Bigg) \setminus \text{set}^{i+m_1}\ X \subseteq \text{UFVars}_i\ u}$$

Axiom 6 (Umap_Udtor).

$$\frac{\text{validU } u \qquad \forall i \in \{1 \ldots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \wedge\ \mathbf{bij}\ f_i}{\text{Udtor}\ \Big(\text{Umap}\ \overline{f}\ u\Big) \\ \subseteq \left\{ \text{map}_{\text{term\_pre}}\ \overline{f}\ \overline{f}\ \overline{\Big(\text{map}_{\text{sum}}\ \Big(\text{permute}\ \overline{f}\Big)\ \Big(\text{Umap}\ \overline{f}\Big)\Big)}\ X \;\middle|\; X.\ X \in \text{Udtor } u \right\}}$$

Axioms 5 and 6 are the duals of the *UFVars_Uctor* and *Umap_Uctor* axioms of the recursor. For the former, all free variables after destructuring have to be included in the free variables of the original value. This ensures that the function blueprint does not introduce new free variables. The latter axiom specifies the commuting of *Umap* with the function blueprint.

Axiom 7 (valid_Umap).

$$\frac{\forall i \in \{1 \ldots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \wedge\ \mathbf{bij}\ f_i \qquad \text{validU } u}{\text{validU}\ \Big(\text{Umap}\ \overline{f}\ u\Big)}$$

Axiom 8 (valid_Udtor).

$$\frac{\text{validU}\ u \qquad X \in \text{Udtor}\ u}{\text{pred}_{\text{term\_pre}}\ (\text{pred}_{\text{sum}}\ (\lambda x.\ \textbf{True})\ \text{validU})\ (\text{pred}_{\text{sum}}\ (\lambda x.\ \textbf{True})\ \text{validU})\ X}$$

Finally, axioms 7 and 8 ensure that the respective operators never leave the valid fragment of the domain type.

Within the locale context, the first step is again to lower the constants to work on the raw type. For the corecursor, there is only one constant that directly involves the term type namely the function blueprint *Udtor*:

Definition 3.32 (raw_Udtor)**.**

$$\textbf{\textit{raw\_ Udtor}}\ u := \left\{ map_{term\_pre}\ \overline{id}\ \overline{(map_{sum}\ Rep_{term}\ id)}\ x \;\middle|\; x.\ x \in Udtor\ u \right\}$$

Again, we need a notion of a *suitable* pick function. However for the corecursor these do not pick bijections, but a pre-term from the set returned by the destructor (see definition 3.33).

Definition 3.33. $\textbf{\textit{suitable}}\ pick := \forall u.\ validU\ u \longrightarrow pick\ u \in raw\_ Udtor\ u$

As the corecursor does not need to permute variables, a normal primitive corecursive function is enough. For this we use the corecursor of the raw type and Hilbert's choice to obtain a suitable pick function (see definition 3.34). The final corecursor is just the lifted version of the raw recursor (see definition 3.35). Most of the following theorems will not directly make use of the raw recursor and leave the *pick* function as a parameter. Later in the construction we will prove that the choice of pick function is irrelevant and thus need this parameterization to be able to instantiate the theorems to different functions. The existence of such a pick function is guaranteed by the *Udtor_ne* axiom.

Definition 3.34 (Raw corecursor)**.** $\textbf{\textit{H}} := corec_{raw\_term}\ (\epsilon\ suitable)$

Definition 3.35 (Corecursor)**.** $\textbf{\textit{COREC}}\ u := Abs_{term}\ (H\ u)$

The first lemma of the corecursor estabishes that the corecursor does not add new free variables (see theorem 3.33). It can be proven by induction on the set of free variables and the lowered version of the *UFVars_Udtor* axiom.

Theorem 3.33 (No new free variables)**.**

$$\frac{\mathit{suitable}\ \mathit{pick} \qquad \mathit{validU}\ u}{\mathit{FVars}\ \left(\mathit{corec}_{\mathit{raw\_term}}\ \mathit{pick}\ u\right) \subseteq \mathit{UFVars}\ u}$$

The main theorem proves commuting of the corecursor with permute and pick-irrelevance at the same time:

Theorem 3.34 (Swapping and pick-irrelevance)**.**

$$\frac{\mathit{suitable}\ \mathit{pick} \qquad \mathit{suitable}\ \mathit{pick}' \qquad \mathit{validU}\ u \qquad \forall i \in \{1 \ldots m_1\}.\ |\boldsymbol{supp}\ f_i| <_o |\alpha_i| \wedge\ \boldsymbol{bij}\ f_i}{\mathit{permute}\ \overline{f}\ \left(\mathit{corec}_{\mathit{raw\_term}}\ \mathit{pick}\ u\right) \equiv_\alpha \mathit{corec}_{\mathit{raw\_term}}\ \mathit{pick}'\ \left(\mathit{Umap}\ \overline{f}\ u\right)}$$

The proof uses coinduction on the α-equivalence relation. It is then necessary to find bijections g_i that witness the α-equivalence between the two raw terms. Because both *pick* functions are suitable, both pre-terms returned by it have to come from the *Udtor*. The axiom *alpha_Udtor* guarantees that all pre-terms returned by it are α-equivalent. With this we can obtain a bijection v_i that we then pre and post compose with f_i to get the final bijection $g_i = f_i \circ v_i \circ f_i^{-1}$.

The last major step is to ensure that picking returns the same pre-term by using an identity pick operations on the top-level and *pick0* for all recursive usages (see theorem 3.35). Afterwards pick-irrelevance is used to replace this custom pick operation with *pick0*.

Theorem 3.35 (Picking on the top-level)**.**

$$\frac{X \in \mathit{raw\_Udtor}\ u \qquad \mathit{validU}\ u}{\begin{array}{l}\mathit{corec}_{\mathit{raw\_term}}\ (\lambda u'.\ \boldsymbol{if}\ u = u'\ \boldsymbol{then}\ X\ \boldsymbol{else}\ \mathit{pick0}\ u')\ u \\ \quad \equiv_\alpha \mathit{ctor\_raw\_term}\ \left(\mathit{map}_{\mathit{term\_pre}}\ \overline{\mathit{id}}\ \overline{(\mathit{case}_{\mathit{sum}}\ \mathit{id}\ H')}\ X\right)\end{array}}$$

After lifting the theorems back to the quotient, we obtain the final three result theorems of the locale:

Theorem 3.36. *COREC_ctor*

$$\frac{\mathit{validU}\ u \qquad X \in \mathit{Udtor}\ u}{\mathit{COREC}\ u = \mathit{ctor\_term}\left(\mathit{map}_{\mathit{term\_pre}}\ \overline{\mathit{id}}\ \overline{(\mathit{case}_{\mathit{sum}}\ \mathit{id}\ \mathit{COREC})}\ X\right)}$$

Theorem 3.37. *COREC_swap*

$$\frac{validU\,u \qquad \forall i \in \{1 \ldots m_1\}.\, |\boldsymbol{supp}\, f_i| <_o |\alpha_i| \wedge \boldsymbol{bij}\, f_i}{COREC\left(Umap\,\overline{f}\,u\right) = permute\,\overline{f}\,(COREC\,u)}$$

Theorem 3.38. *COREC_FVars*

$$\frac{validU\,u}{\forall i \in \{1 \ldots m_1\}.\, FVars_i\,(COREC\,u) \subseteq UFVars_i\,u}$$

3.3.7 Renaming in codatatypes

With the corecursor locale, we can now define renaming also for codatatypes. For this we choose the domain type $\upsilon := \overline{\alpha}\,\text{term} \times \overline{\alpha \to \alpha}$. The locale constants are chosen as the following:

Definition 3.36 (Locale instantiation for renaming)**.**

$$\begin{aligned}
Umap &:= \lambda\overline{f}\ (t, \overline{g})\,.\ \left(permute\,\overline{f}\,t, \overline{f \circ g \circ f^{-1}}\right)\\
UFVars^i &:= \lambda\,(t, \overline{g})\,.\,FVars^i\,t\ \cup\ imsupp\,g_i\\
validU &:= \lambda\,(t, \overline{g})\,.\,\forall i \in \{1 \ldots m_1\}.\, |\boldsymbol{supp}\,g_i| <_o |\alpha_i|\\
Udtor &:= \lambda\,(t, \overline{g})\,.\,\Big\{map_{term_pre}\,\overline{g}\,\overline{id}\,\overline{(\lambda t.\ Inr\,(t, \overline{g}))}\,x\\
&\qquad \Big|\ t = ctor_term\,x \wedge \forall i \in \{1 \ldots m_1\}.\, set^{i+m_1}\,x \cap imsupp\,g_i = \{\}\Big\}
\end{aligned}$$

Most of the axioms are pretty straightforward to prove, the two that are more difficult are *alpha_Udtor* and *Umap_Udtor*. The former requires fresh quasi-injectivity (theorem 3.10) to avoid the functions $\overline{g}$. The latter uses fresh cases (theorem 3.8) and some careful instantiation of two nested existentials. Afterwards, the renaming function packages up the functions into the tuple:

Definition 3.37 (Renaming function)**.** $\boldsymbol{rename}\,\overline{g}\,t := COREC\ (t, \overline{g})$

Dual to the renaming operator for datatypes, we can prove most of the MRBNF axioms using straightforward existential coinduction (theorem 3.18). The proof that renaming respects identity again follows from the fact that renaming and permutation coincide for bijections (see theorem 3.31).

The only proof that does not follow this scheme is that renaming commutes with the free variable operators (see theorem 3.39). As this is an equality be-

tween two sets of variables and not two terms, we cannot use coinduction for the proof. Here we have to make use of fresh induction on the free variables (theorem 3.22) and prove both directions of the equality separately. The first direction $a \in \{f_i\ b \mid b.\ b \in \text{FVars}^i\ x\} \longrightarrow a \in \text{FVars}^i\ (\text{rename}\ \overline{f}\ x)$ is pretty straightforward. The other direction additionally requires using fresh cases (theorem 3.8) and fresh quasi-injectivity (theorem 3.10).

Theorem 3.39 (Renaming commutes with free variables)**.** *For all* $j \leq m_1$:

$$\frac{\forall i \in \{1 \ldots m_1\}.\ |\boldsymbol{supp}\ f_i| <_o |\alpha_i|}{FVars^j\ (rename\ \overline{f}\ x) = \{f_j\ a \mid a.\ a \in FVars\ x\}}$$

With these theorems we can prove that the codatatype forms a MRBNF with renaming as its *map* function and the free variable operators as its *set* functions.

3.4 Defining Substitution

As described in section 2.4, our main goal is to automatically maintain and lift substitution operators across types. However, there is more nuance to our axiomatization. For example, recall the syntax of the π-calculus [38], described by the following grammar (where we omit constructors that are not relevant to our discussion):

Definition 3.38. *Syntax of the* π*-calculus*

$$P, Q \quad ::= \quad 0 \mid P + Q \mid P \parallel Q \mid\ !P \mid [x = y]P \mid [x \neq y]P \mid \overline{a}\ x.P \mid a(x).P \mid v(x).P$$

Here, x and y are variables called *channel names*. We assume that x is bound in P within processes of the form $a(x).P$ and $v(x).P$, and processes are equated modulo the induced notion of α-equivalence. For this syntax of processes, (capture-avoiding) renaming can be defined as seen in section 3.3.5. However, there is no injection of free variables into this type, i.e., there is no point where a substitution could replace a variable x with a process P.

If we expand the syntax with process variables and recursive processes similar to Milner's CCS [39] (see definition 3.39), we do introduce such an injection: the PVr constructor.

Definition 3.39. *New constructors in the π-calculus with recursive processes*

$$P \quad ::= \quad \ldots \mid PVr\, X \mid Rec\, X\, P$$

While we could choose different sets for the variables representing channel names, and those representing process variables, nothing should prevent the user from using the same set for both. Indeed, the locations where variables occur in processes would make it unambiguous whether we deal with a channel or a process variable – so we could have two kinds of substitutions for the same kind of variable.

This also reveals another issue: if we define a substitution for this π-calculus now, we can either substitute process variables for processes or we can apply a renaming to all variables – channel names and process variables – via the renaming operator from chapter 3. However, there is no way to only rename channel names.

To remedy this, we should think of newly emerging substitutions as "overwriting" existing renamings either partially or fully. For syntax where the injection of free variables is the only source of free variables, the substitution will fully overwrite the renaming. In the other cases, like here with the π-calculus, we retain a renaming for the free variable stemming from outside the injection.

If we are to have multiple substitutions for a single kind of variable anyways, we can also allow the type to have more then one injection of free variables for the same kind of variable. Again, as long as these injections do not overlap, it is unambiguous from the syntax to which of the injections any given variable belongs.

Lastly, to unify all of the different sources of substitutions (new substitutions from free variable injection, substitutions lifted from nested types and "left-over" renaming), we will look at the monadic structure of substitution [34, 40] as being *relative* to other monads. Lifted substitutions are relative to the type they are lifted from, and renaming can be viewed as being relative to the trivial identity monad.

3.4.1 Multi-Nominal (MN) Monads

A multi-nominal monad is a group of types, with one designated *leader* L. Intuitively, the leader carries the transitive closure of all other types that it has substitutions for. For System F (as seen in section 2.4), the type of types carries no other types as its only substitution is its own. The terms of System F will carry the

types to facilitate a lifted type-in-term substitution.

Every type in the group has a single substitution operator *Sb* that accepts multiple *subst functions* as arguments. In particular, there will be one argument per injection of free variables, be it own injections or injections from other types in the group. The *Sb* operator may also accept renaming functions if the syntax contains extra free variables. Paired with each injection is a fine-grained free variable operator *Vrs* that returns the set of free variables originating from this injection. Additionally, for every renaming function that the *Sb* operator accepts, we also have a free variable operator *RVrs* for the "left-over" free variables outside of any injections. As these types need to be able to produce enough fresh variables, we will also carry one cardinal bound for all of the variable sets of all the types.

Definition 3.40 (Multi-Nominal Monad)**.** *A Multi-Nominal Monad consists of a non-empty set of types called Ops with one designated element L called leader and an infinite, regular cardinal bound bd:*

$Ops = \{L = \overline{\alpha}\, T, \overline{\alpha'}\, T_1, \ldots, \overline{\alpha'^{\cdots\prime}}\, T_n\}$, *where* $\overline{\alpha'} \subseteq \overline{\alpha} \wedge \cdots \wedge \overline{\alpha'^{\cdots\prime}} \subseteq \overline{\alpha}$

For every type $\overline{\alpha}\, T \in Ops$ *there exists:*

$$
\begin{array}{lcl}
\overline{Inj_T} & : & \alpha \to \overline{\alpha'}\, T', \text{where } \alpha \in \overline{\alpha} \wedge \overline{\alpha'}\, T' \in Ops \\
\overline{Vrs_T} & : & \overline{\alpha}\, T \to \alpha \text{ set}, \text{for each } (Inj : \alpha \to \overline{\alpha'}\, T') \in \overline{Inj_T} \\
\overline{RVrs_T} & : & \overline{\alpha}\, T \to \alpha \text{ set}, \text{where } \alpha \in \overline{\alpha} \\
Sb_T & : & \overline{(\alpha_i \to \alpha_i)} \to \overline{(\alpha_j \to \overline{\alpha'}\, T'_j)} \to \overline{\alpha}\, T \to \overline{\alpha}\, T \\
 & & \text{for all } (RVrs_i : \overline{\alpha}\, T \to \alpha_i \text{ set}) \in \overline{RVrs_T} \wedge (Inj_j : \alpha_j \to \overline{\alpha'}\, T'_j) \in \overline{Inj_T}
\end{array}
$$

such that the MN-Monad axioms hold.

A note on notation: for definitions in the rest of this chapter, we make use of a specific forall notation to range over injections and variable operators, for example $\forall(Inj_j : \alpha \to \overline{\alpha'}\, T') \in \overline{\text{Injs}_T}$. Here, we range over the injections of the type T. If the type α is already in use, this means that we range only over the injections with that α as domain. Otherwise the α is bound for later use. The number j represents the index of the bound injection. Given that a multi-nominal monad always has one fine-grained variable operator per injection, as well as one subst function argument on the substitution operator *Sb*, we can use this to select the corresponding operator or argument. Lastly, $\overline{\alpha'}\, T'$ binds the type from the set *Ops* that the injection returns. This type may be the type $\overline{\alpha}\, T$ itself. Similar to α if the

type is already used, we only range over the injections with an appropriate type. Otherwise we bind the type for later use.

Axiom 1 (Sb_Inj).

$$\mathrm{Sb}_T\ \overline{id}\ \overline{\mathrm{Inj}_T} = \mathrm{id}$$

Axiom 2 (Sb_comp_Inj).

$$\frac{\begin{array}{r}\forall(\mathrm{RVrs}_i : \overline{\alpha}\ T \to \alpha_i\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\ T') \in \overline{\mathrm{Inj}_{Tj}}.\ |\{a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \wedge\ \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\rho_j\ a) \mid a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'|\end{array}}{\forall(\mathrm{Inj}_k : \alpha \to \overline{\alpha}\ T) \in \overline{\mathrm{Inj}_T}.\ \mathrm{Sb}_T\ \overline{f}\ \overline{\rho} \circ \mathrm{Inj}_k = \rho_k}$$

Axioms 1 and 2 are the left and right identity of the Kleisli triple [41] where Sb_T is the monadic bind operator and $\overline{Inj_T}$ is the *return* operator (see section 3.1.4). In general, the MN-Monad axioms require small-support substitutions not only with regard to the domain of the subst function, but also including all the free variables in the codomain. Note that in axiom 2, our forall notation only ranges over those injections that return the top-level type T and not potential other injections lifted from other types.

Axiom 3 (Sb_comp).

$$\frac{\begin{array}{r}\forall(\mathrm{RVrs}_i : \overline{\alpha}\ T \to \alpha_i\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\ T') \in \overline{\mathrm{Inj}_{Tj}}.\ |\{a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \wedge\ \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\rho_j\ a) \mid a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'| \\ \forall(\mathrm{RVrs}_i : \overline{\alpha}\ T \to \alpha_i\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ g_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\ T') \in \overline{\mathrm{Inj}_{Tj}}.\ |\{a.\ \sigma_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \wedge\ \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\sigma_j\ a) \mid a.\ \sigma_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'|\end{array}}{\mathrm{Sb}_T\ \overline{f}\ \overline{\rho} \circ \mathrm{Sb}_T\ \overline{g}\ \overline{\sigma} = \mathrm{Sb}_T \overline{(f \circ g)}\ \overline{(\mathrm{Sb}_{T'}\ \overline{f'}\ \overline{\rho'} \circ \sigma)}, \text{where } \overline{f'} \subseteq \overline{f}, \overline{\rho'} \subseteq \overline{\rho}}$$

Axiom 3 is the monad composition law. Again, all the assumptions on this axiom ensure that the subst functions are small. As the substitution operator Sb_T may carry subst functions from other types, on the right hand side of the equality the inner substitution operators $\overline{Sb_{T'}}$ may be from different types. For each of those operators, there exists only one valid choice of $\overline{f'}$ and $\overline{\rho'}$, namely the one that makes the application type correct and where each ρ' corresponds to the respective injections of T'.

Axiom 4 (Vrs_bd).

$$\forall(\mathrm{Vrs} : \overline{\alpha}\, T \to \alpha\ \mathrm{set}) \in (\overline{\mathrm{Vrs}_T} \cup \overline{\mathrm{RVrs}_T}).\ |\mathrm{Vrs}\ x| <_o \mathrm{bd}$$

To ensure we can always obtain fresh variables, the sets of fine-grained free variables have to be bounded by the cardinal bound of the MN-Monad (see axiom 4).

Axiom 5 (Vrs_Inj).

$$\forall(\mathrm{Inj}_i : \alpha \to \overline{\alpha}\, T) \in \overline{\mathrm{Inj}_T}.\forall j \in \{1 \ldots |\overline{\mathrm{Inj}_T}|\}.\mathrm{Vrs}_j\ (\mathrm{Inj}_i\ a) = \begin{cases} \{a\}, & \text{if } i = j \\ \{\}, & \text{otherwise} \end{cases}$$

Axiom 5 specifies the set of free variables of the injections, namely that each *Vrs* operator has to return the empty set of all injections except the one it is paired with.

Axiom 6 (RVrs_Sb).

$$\frac{\begin{array}{r}\forall(\mathrm{RVrs}_i : \overline{\alpha}\, T \to \alpha_i\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\, T') \in \overline{\mathrm{Inj}_{T\,j}}.\ |\{a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \wedge\ \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\rho_j\ a) \mid a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'|\end{array}}{\begin{array}{l}\forall(RVrs_i : \overline{\alpha}\, T \to \alpha\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ \mathrm{RVrs}_i\ (\mathrm{Sb}_T\ \overline{f}\ \overline{\rho}\ x) = \\ \quad \{f_i\ a \mid a.\ a \in RVrs_i\ x\} \cup \bigcup_{(\mathrm{Inj}_j\ :\ \alpha' \to \overline{\alpha}\, T) \in \overline{\mathrm{Inj}_T}} \left(\bigcup_{a \in \mathrm{Vrs}_j\ x} \mathrm{RVrs}_i\ (\rho_j\ a) \right)\end{array}}$$

Axiom 7 (Vrs_Sb).

$$\frac{\begin{array}{r}\forall(\mathrm{RVrs}_i : \overline{\alpha}\, T \to \alpha_i\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\, T') \in \overline{\mathrm{Inj}_{T\,j}}.\ |\{a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \wedge\ \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\rho_j\ a) \mid a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'|\end{array}}{\begin{array}{l}\forall(Vrs_i : \overline{\alpha}\, T \to \alpha\ \mathrm{set}) \in \overline{\mathrm{Vrs}_T}.\ \mathrm{Vrs}_i\ (\mathrm{Sb}_T\ \overline{f}\ \overline{\rho}\ x) = \\ \quad \bigcup_{(\mathrm{Inj}_j\ :\ \alpha' \to \overline{\alpha'}\, T') \in \overline{\mathrm{Inj}_T}} \left(\bigcup_{(\mathrm{Inj}_{T',k}\ :\ \alpha \to \overline{\alpha'}\, T') \in \overline{\mathrm{Inj}_{T'}} \wedge \mathrm{Inj}_{T,i} = \mathrm{Inj}_{T',k}} \left(\bigcup_{a \in \mathrm{Vrs}_{T,j}\ x} \mathrm{Vrs}_{T',k}\ (\rho_j\ a) \right) \right)\end{array}}$$

Axioms 6 and 7 define the set of free variables after applying the substitution operator. For the functorial variables, this is the union of the variables before substitution and the free variables after applying each of the subst functions to the variables returned by their paired *Vrs* operators. For the subst functions (or rather its paired injection) we need to find all the types in *Ops* that also have the

same injection. For each of those we union the variables returned by the paired *Vrs* operator after applying the subst function.

Axiom 8 (Sb_cong).

$$\frac{\begin{array}{c}\forall(\mathrm{RVrs}_i : \overline{\alpha}\ T \to \alpha_i\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\ T') \in \overline{\mathrm{Inj}_{Tj}}.\ |\{a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \qquad \wedge \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\rho_j\ a) \mid a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'| \\ \forall(\mathrm{RVrs}_i : \overline{\alpha}\ T \to \alpha_i\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ g_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\ T') \in \overline{\mathrm{Inj}_{Tj}}.\ |\{a.\ \sigma_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \qquad \wedge \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\sigma_j\ a) \mid a.\ \sigma_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'| \\ \forall(\mathrm{RVrs}_i : \overline{\alpha}\ T \to \alpha\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ \forall a \in \mathrm{RVrs}_i\ x.\ f_i\ a = g_i\ a \\ \forall(\mathrm{Vrs}_j : \overline{\alpha}\ T \to \alpha\ \mathrm{set}) \in \overline{\mathrm{Vrs}_T}.\ \forall a \in \mathrm{Vrs}_j\ x.\ \rho_j\ a = \sigma_j\ a\end{array}}{\mathrm{Sb}\ \overline{f}\ \overline{\rho}\ x = \mathrm{Sb}\ \overline{g}\ \overline{\sigma}\ x}$$

Lastly, axiom 8 ensures that substitution only depends on the free variables similar to the *map_cong* axiom of MRBNFs.

Throughout this chapter, we have been calling the free variable injections "injections", but none of the axioms state this directly. However we can derive this fact:

Theorem 3.40. *Free variable injections are injections*

$\forall Inj_j \in \overline{Inj_T}.\ Inj_j\ a = Inj_j\ b \longrightarrow a = b$

Proof. *If $Inj_j\ a = Inj_j\ b$ then also $Vrs_j\ (Inj_j\ a) = Vrs_j\ (Inj_j\ b)$. Using the* ***Vrs_Inj*** *axiom, we get $\{a\} = \{b\}$.* □

To make the MN-Monad construction more concrete, we can show that both the types and terms of System F form one (see examples 3.1 and 3.2). This also showcases the behavior when a type contains an already existing MN-Monad. All the types in the set *Ops* of the nested type are added to the *Ops* of the outer type, while leaving their constants unchanged.

Example 3.1. *The types of System F form a MN-Monad*

Assuming the type that represents the types of System F is α FType, with the

free variable constructor called "TyVar", we choose:

$$\begin{array}{lcl}
Ops & := & \{L = \alpha\ FType\} \\
bd & := & \aleph_0 \\
\overline{Inj_{FType}} & := & [TyVar : \alpha \to \alpha\ FType] \\
\overline{Vrs_{FType}} & := & [FVars_{FType} : \alpha\ FType \to \alpha\ set] \\
\overline{RVrs_{FType}} & := & [\,] \\
Sb_{FType} & := & (\lambda\rho\ x.\ x[\rho]) : (\alpha \to \alpha\ FType) \to \alpha\ FType \to \alpha\ FType
\end{array}$$

Example 3.2. *The terms of System F form a MN-Monad*

Assuming the type that represents the terms of System F is $(\alpha, \beta)\ FTerm$*, with the free variable constructor called "Var", we choose:*

$$\begin{array}{lcl}
Ops & := & \{L = (\alpha, \beta)\ FTerm, \alpha\ FType\} \\
bd & := & \aleph_0 \\
\\
\overline{Inj_{FTerm}} & := & [Var : \beta \to (\alpha, \beta)\ FTerm, TyVar : \alpha \to \alpha\ FType] \\
\overline{Vrs_{FTerm}} & := & [FVars_{FTerm} : (\alpha, \beta)\ FTerm \to \beta\ set, \\
 & & \ FTVars_{FTerm} : (\alpha, \beta)\ FTerm \to \alpha\ set] \\
\overline{RVrs_{FTerm}} & := & [\,] \\
Sb_{FTerm} & := & (\lambda\rho_1\ \rho_2\ x.\ (x[\rho_2]_\tau)[\rho_1]) : (\beta \to (\alpha, \beta)\ FTerm) \\
 & & \quad \to (\alpha \to \alpha\ FType) \to (\alpha, \beta)\ FTerm \to (\alpha, \beta)\ FTerm \\
\\
\overline{Inj_{FType}} & := & [TyVar : \alpha \to \alpha\ FType] \\
\overline{Vrs_{FType}} & := & [FVars_{FType} : \alpha\ FType \to \alpha\ set] \\
\overline{RVrs_{FType}} & := & [\,] \\
Sb_{FType} & := & (\lambda\rho\ x.\ x[\rho]) : (\alpha \to \alpha\ FType) \to \alpha\ FType \to \alpha\ FType
\end{array}$$

3.4.2 Closure under composition

While we now have a structure that can accommodate the kinds of substitutions we want, we still need some way to automatically *maintain* that structure. The most obvious solution is to mirror the datatype construction and use a compositional approach. However, in order to do this we have to generalize our definition of a MN-Monad to provide "slots" – similar to the live positions on MRBNFs – that can

be filled with nested MN-Monads:

Definition 3.41 (Parameterized Multi-Nominal Monad)**.** *A parameterized Multi-Nominal Monad consists of a non-empty set of types called Ops with one designated element L called leader and an infinite, regular cardinal bound bd:*

$Ops = \{L = (\overline{\alpha}, \overline{\gamma})\,T, (\overline{\alpha'}, \overline{\gamma'})\,T_1, \ldots, (\overline{\alpha'^{\ldots\prime}}, \overline{\gamma'^{\ldots\prime}})\,T_n\}$*, where* $\overline{\alpha'} \subseteq \overline{\alpha} \wedge \cdots \wedge \overline{\alpha'^{\ldots\prime}} \subseteq \overline{\alpha}$ *and* $\overline{\gamma'} \subseteq \overline{\gamma} \wedge \cdots \wedge \overline{\gamma'^{\ldots\prime}} \subseteq \overline{\gamma}$

For every type $(\overline{\alpha}, \overline{\gamma})\,T \in Ops$ *there exists:*

$$
\begin{array}{lcl}
\overline{Inj_T} & : & \alpha \to \overline{\alpha'}\,T', \text{where } \alpha \in \overline{\alpha} \wedge \overline{\alpha'}\,T' \in Ops \\
\overline{Vrs_T} & : & \overline{\alpha}\,T \to \alpha \text{ set}, \text{for all } (Inj : \alpha \to \overline{\alpha'}\,T') \in \overline{Inj_T} \\
\overline{RVrs_T} & : & \overline{\alpha}\,T \to \alpha \text{ set}, \text{where } \alpha \in \overline{\alpha} \\
Sb_T & : & \overline{(\alpha_i \to \alpha_i)} \to \overline{(\alpha_j \to \overline{\alpha'}\,T'_j)} \to \overline{\alpha}\,T \to \overline{\alpha}\,T \\
 & & \text{for all } (RVrs_i : \overline{\alpha}\,T \to \alpha_i \text{ set}) \in \overline{RVrs_T} \wedge (Inj_j : \alpha_j \to \overline{\alpha'}\,T'_j) \in \overline{Inj_T} \\
Map_T & : & \overline{(\gamma \to \gamma')} \to (\overline{\alpha}, \overline{\gamma})\,T \to (\overline{\alpha}, \overline{\gamma'})\,T \\
\overline{Supp_T} & : & (\overline{\alpha}, \overline{\gamma})\,T \to \gamma \text{ set}, \text{for all } \gamma \in \overline{\gamma}
\end{array}
$$

such that all axioms of a MN-Monad hold, as well as:

Axiom 9 (Map_id).

$$\mathrm{Map}_T\ \overline{\mathrm{id}} = \mathrm{id}$$

Axiom 10 (Map_comp).

$$\mathrm{Map}_T\ \overline{f} \circ \mathrm{Map}_T\ \overline{g} = \mathrm{Map}_T\ \overline{(f \circ g)}$$

Axiom 11 (Supp_Map).

$$\forall i \in \{1 \ldots |\overline{\mathrm{Supp}_T}|\}.\ \mathrm{Supp}^i_T\ (\mathrm{Map}_T\ \overline{f}\ x) = \{f_i\ a \mid .\ a \in \mathrm{Supp}^i_T\ x\}$$

Axiom 12 (Supp_bd).

$$\forall i \in \{1 \ldots |\overline{\mathrm{Supp}_T}|\}.\ |\mathrm{Supp}^i_T\ x| <_o \mathrm{bd}$$

Axiom 13 (Map_cong).

$$\frac{\forall i \in \{1 \ldots |\overline{\mathrm{Supp}_T}|\}.\ \forall a \in \mathrm{Supp}^i_T\ x.\ f_i\ a = g_i\ a}{\mathrm{Map}_T\ \overline{f}\ x = \mathrm{Map}_T\ \overline{g}\ x}$$

Axioms 9 through 13 are what we call the *supported functor axioms*. They mirror the axioms of a MRBNF.

Axiom 14 (Vrs_Map).

$$\forall \mathrm{Vrs} \in (\overline{\mathrm{Vrs}_T} \cup \overline{\mathrm{RVrs}_T}).\ Vrs\ (\mathrm{Map}_T\ \overline{f}\ x) = \mathrm{Vrs}\ x$$

Axiom 15 (Map_Inj).

$$\forall \mathrm{Inj} \in \overline{\mathrm{Inj}_T}.\ \mathrm{Map}_T\ \overline{f} \circ \mathrm{Inj} = \mathrm{Inj}$$

On top of the supported functor axioms, axioms 14 and 15 ensure that the map function does not interfere with the fine-grained free variable operators or the injections.

Axiom 16 (Map_Sb).

$$\frac{\begin{array}{r}\forall(\mathrm{RVrs}_i : \overline{\alpha}\ T \to \alpha_i\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\ T') \in \overline{\mathrm{Inj}_{Tj}}.\ |\{a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \wedge\ \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\rho_j\ a) \mid a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'|\end{array}}{\mathrm{Map}_T\ \overline{g} \circ \mathrm{Sb}_T\ \overline{f}\ \overline{\rho} = \mathrm{Sb}_T\ \overline{f}\ (\overline{\mathrm{Map}_{T'}\ \overline{g'} \circ \rho}) \circ \mathrm{Map}_T\ \overline{g}, \text{where } \overline{g'} \subseteq \overline{g}}$$

Axiom 16 requires that mapping is orthogonal to substitution, at least on the free variable arguments. However, the substitutions may introduce new "slots", so we need to map over those as well.

Axiom 17 (Supp_Sb).

$$\frac{\begin{array}{r}\forall(\mathrm{RVrs}_i : \overline{\alpha}\ T \to \alpha_i\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\ T') \in \overline{\mathrm{Inj}_{Tj}}.\ |\{a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \wedge\ \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\rho_j\ a) \mid a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'|\end{array}}{\begin{array}{l}\forall(\mathrm{Supp}^i_T : (\overline{\alpha}, \overline{\gamma})\ T \to \gamma\ \mathrm{set}) \in \overline{\mathrm{Supp}_T}.\ \mathrm{Supp}^i_T\ (\mathrm{Sb}_T\ \overline{f}\ \overline{\rho}\ x) = \mathrm{Supp}^i_T\ x \\ \quad \cup \bigcup_{(\mathrm{Inj}_j : \alpha' \to (\overline{\alpha'}, \overline{\gamma'})\ T') \in \overline{\mathrm{Inj}_T} \wedge \gamma \in \overline{\gamma'}} \left(\bigcup_{a \in \mathrm{Vrs}_{T,j}\ x} \mathrm{Supp}^{\gamma}_{T'}\ (\rho_j\ a) \right)\end{array}}$$

The last axiom specifies how *Supp* interacts with substitution. Similar to the axiom 16, the "slots" are independent of substitution, but new slots may be introduced in the process.

This generalization allows us to create an MN-Monad from any MRBNF:

Theorem 3.41. *Any MRBNF defines a MN-Monad*

Given a MRBNF $\mathcal{F} = \left(\left(\overline{\alpha}_{m_1}, \overline{\beta}_{m_2}, \overline{\gamma}_n, \overline{\delta}\right) F,\ map_F,\ \overline{set_F},\ rel_F,\ bd_F\right)$*, we choose:*

$$
\begin{array}{lcl}
Ops & := & \left\{L = \left(\overline{\alpha}_{m_1}, \overline{\beta}_{m_2}, \overline{\gamma}_n, \overline{\delta}\right) F\right\} \\
bd & := & bd_F \\
\overline{Inj_F} & := & [\,] \\
\overline{Vrs_F} & := & [\,] \\
\overline{RVrs_F} & := & \overline{set_F^i}, \text{ where } 1 \leq i \leq m_1 \\
Sb_{FType} & := & (\lambda \overline{f}\ x.\ map_F\ \overline{f}\ \overline{id}\ \overline{id}\ x) : \overline{(\alpha \to \alpha)} \to \left(\overline{\alpha}, \overline{\beta}, \overline{\gamma}, \overline{\delta}\right) F \to \left(\overline{\alpha}, \overline{\beta}, \overline{\gamma}, \overline{\delta}\right) F \\
Map_F & := & (\lambda \overline{g}\ x.\ map_F\ \overline{id}\ \overline{id}\ \overline{g}\ x) : \overline{(\gamma \to \gamma')} \to \left(\overline{\alpha}, \overline{\beta}, \overline{\gamma}, \overline{\delta}\right) F \to \left(\overline{\alpha}, \overline{\beta}, \overline{\gamma'}, \overline{\delta}\right) F \\
\overline{Supp_F} & := & \overline{set_F^k}, \text{ where } 1 \leq k - (m_1 + m_2) \leq n
\end{array}
$$

All of the MN-Monad axioms follow directly from the MRBNF axioms.

This allows us to treat, among others, the binary sum and product types as MN-Monads. Similar to the datatype construction, we can construct the MN-Monad structure for an arbitrary type with a recursive procedure: if the type is a type variable or a type that is not a MRBNF, return the MN-Monad structure induced by the identity and constant functor respectively. If the type is already a MN-Monad return that structure. If the type is a type constructor (that is at least a MRBNF), recursively convert all of its arguments, convert the type constructor itself and do a single composition step (see definition 3.42). As an optimization, it is possible to delay the construction of the MN-Monad structure for type variables and MRBNFs until any of the involved types is already a MN-Monad (i.e., one with a proper substitution). This means that types that do not nest other syntaxes with existing substitutions (e.g. the types of System F) will never perform a MN-Monad composition, only MRBNF compositions.

Definition 3.42. *Composition of MN-Monads*

Given an outer MN-Monad $\mathcal{M}$ *with its leader* $L = (\overline{\alpha}, \overline{\gamma}_n)\ F$*, as well as* n *inner MN-Monads* $\overline{\mathcal{T}}$*, each with a leader* $L_i = (\overline{\alpha'}, \overline{\gamma'})\ T_i$*, we can construct a composed MN-Monad:*

$$
\begin{array}{lcl}
L & := & (\overline{\alpha}, \overline{\overline{\alpha'}\ T})\ F \\
\overline{Inj_L} & := & \overline{Inj_F} \cup \overline{Inj_{T_1}} \cup \cdots \cup \overline{Inj_{T_n}} \\
\overline{Vrs_L} & := & (Vrs_j)_{j \in \{1 \ldots | \overline{Inj_L} |\}} \\
Vrs_{L,j} & := & \lambda x.\ Vrs_X\ x \cup \bigcup_{i=1}^{n} \left(\bigcup_{Inj_{T_i,k} \in \overline{Inj_{T_i}} \wedge Inj_{T_i,k} = Inj_{L,j}} \left(\bigcup_{t \in Supp_{F,i}\ x} (Vrs_{T_i,k}\ t) \right) \right) \\
 & & \text{where } Vrs_X\ x = \begin{cases} Vrs_{F,i}\ x, & \text{if } \exists i.\ Inj_{F,i} = Inj_{L,j} \\ \{\}, & \text{otherwise} \end{cases} \\
\overline{RVrs_L} & := & (RVrs_\alpha)_{\alpha \in (\overline{\alpha} \cup \overline{\alpha'})} \\
RVrs_{L,\alpha} & := & \lambda x.\ RVrs_{T,X}\ x \cup \bigcup_{i=1}^{n} \left(\bigcup_{t \in Supp_{F,i}\ x} (RVrs_{T_i,X}\ t) \right) \\
 & & \text{where } RVrs_{T',X}\ x\ = \begin{cases} RVrs_{T',k}\ x, & \text{if } \exists k.\ codom(RVrs_{T',k}) = \alpha\ set \\ \{\}, & \text{otherwise} \end{cases} \\
Sb_L & := & \lambda \overline{f}\ \overline{\rho}.\ Sb_F\ \overline{f'}\ \overline{\rho'} \circ Map_F\ \overline{(Sb_{T_i}\ \overline{f'^{\cdots\prime}}\ \overline{\rho'^{\cdots\prime}})}, \text{where } \overline{f'}, \overline{f'^{\cdots\prime}} \subseteq \overline{f} \wedge \overline{\rho'}, \overline{\rho'^{\cdots\prime}} \subseteq \overline{\rho} \\
Map_L & := & \lambda \overline{g}.\ Map_F\ \overline{(Map_{T_i}\ \overline{g'})}, \text{where } \overline{g'} \subseteq \overline{g} \\
\overline{Supp_L} & := & (Supp_\gamma)_{\gamma \in \overline{\gamma'}}, \text{where } Supp_{L,\gamma} = \lambda x.\ \bigcup_{i=1}^{n} \left(\bigcup_{t \in Supp_F^i\ x} Supp_{T_i}^{\gamma}\ t \right)
\end{array}
$$

$$
\begin{array}{lcl}
Ops & := & \{L\} \cup \{codom(Inj) \mid Inj.\ Inj \in \overline{Inj_L}\} \\
bd & := & max\,\{bd_F, bd_{T_1}, \ldots, bd_{T_n}\}
\end{array}
$$

There are several noteworthy things about this definition. First and foremost, we define the set *Ops* only at the very end, after all the definitions for the new leader L. The reason for this is cleanup. We do not want to keep growing the number of types we have to carry if they do not contribute in an interesting, i.e., substitutive way. If we were to just merge all the *Ops* sets, potentially removing the leader of the outer MN-Monad that will be replaced by the new leader L, we would also carry types like the binary sum or products. These types only act as containers for other types, and are not interesting by themselves. Another possible solution would be to exclude all the leaders (so including those of the inners), however that would remove too many types. Consider the composition $\alpha\,\mathrm{FType} + \alpha$, where we have the binary sum as outer type, and the type of System F types as well as a type variable as inner types. As seen in example 3.1, the type of System F types is its own leader, but we need to carry it in the *Ops* of the composed type because it has a relevant substitution. By including the types that are pointed at by free variable injections,

we ensure that this still happens.

The free variable operators merge the sets of free variables from all the types if they are connected to the same injection. This ensures that if a type appears twice, there will only be one subst function that applies to both. The substitution function itself has to apply the inner substitutions first because the other substitution may introduce new instances of the inner types. If they were to come first we would substitute again in the newly substituted types.

With this recursive procedure, we can preserve the monadic structure of an arbitrary type, similarly to how the composition algorithm in section 3.3.2 preserves the functorial structure. Concretely, if we want to construct a binding datatype to represent the terms of System F using the binding fixpoint from section 3.3.3, we start out with a non-recursive pre-datatype (see definition 3.43). The MN-Monad composition gives us free variable operators and a substitution operator on that pre-datatype (see example 3.3).

Definition 3.43. *Pre-datatype of the terms of System F*

$$
\begin{array}{lcll}
(\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2)\ FTerm_{pre} & := & \alpha_2 & \textit{(free) term variables} \\
& + & \gamma_1 \times \gamma_1 & \textit{term application} \\
& + & \beta_2 \times \alpha_1\ FType \times \gamma_2 & \lambda\textit{-abstraction} \\
& + & \gamma_1 \times \alpha_1\ FType & \textit{type application} \\
& + & \beta_1 \times \gamma_2 & \textit{type abstraction}
\end{array}
$$

Example 3.3. *MN-Monad structure of the pre-datatype of System F terms*

$$
\begin{array}{lcl}
Ops & := & \{L = (\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2)\ FTerm_{pre}, \alpha_1\ FType\} \\
bd & := & \aleph_0
\end{array}
$$

$$
\begin{array}{lcl}
\overline{Inj_L} & := & [TyVar : \alpha_1 \to \alpha_1\ FType] \\
\overline{Vrs_L} & := & [Vrs_{L,1} : (\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2)\ FTerm_{pre} \to \alpha_1\ set] \\
\overline{RVrs_L} & : & [RVrs_{L,1} : (\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2)\ FTerm_{pre} \to \alpha_2\ set] \\
Sb_L & : & (\alpha_2 \to \alpha_2) \to (\alpha_1 \to \alpha_1\ FType) \\
& & \quad \to (\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2)\ FTerm_{pre} \to (\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2)\ FTerm_{pre} \\
Map_L & : & (\gamma_1 \to \gamma_1') \to (\gamma_2 \to \gamma_2') \\
& & \quad \to (\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2)\ FTerm_{pre} \to (\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1', \gamma_2')\ FTerm_{pre}
\end{array}
$$

$$\begin{aligned}\overline{Supp_L} \;:=\; [&Supp_{L,1} : (\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2)\ FTerm_{pre} \to \gamma_1\ set,\\ &Supp_{L,2} : (\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2)\ FTerm_{pre} \to \gamma_2\ set]\end{aligned}$$

3.4.3 Map-Restricted Substitutive BNFs (MRSBNFs)

So far, we have achieved the automatic maintenance of substitutions and how to carry them though composite types. However, all the examples so far assumed that the relevant types already have a substitution operator defined. But new substitutions can only appear as part of the fixpoint construction of datatypes.

Given that the binder fixpoint operation is defined on the functorial structure, we need to ensure the compatibility of the functorial and monadic structures. We call this combined structure a Map-Restricted Substitutive Bounded Natural Functor (MRSBNF):.

Definition 3.44. *Map-Restricted Substitutive Bounded Natural Functor (MRSBNF)*

A MN-Monad $\mathcal{M}$ where all the types $(\overline{\alpha}_{m_1}, \overline{\beta}_{m_2}, \overline{\gamma}_n, \overline{\delta})\ T \in Ops_{\mathcal{M}}$ are MRBNFs, is called a MRSBNF if for each of the types the following axioms hold:

Axiom 1 (map_is_Sb).

$$\frac{\forall i \in \{1 \dots m_1\}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i|}{\mathrm{map}_T \overline{f}\ \overline{\mathrm{id}}\ \overline{g} = \mathrm{Sb}_T\ \overline{f'}\ \overline{(\mathrm{Inj}_T \circ f'')} \circ \mathrm{Map}_T\ \overline{g} \quad \text{where } \overline{f'}, \overline{f''} \subseteq \overline{f}}$$

Axiom 1 is the central axiom of a MRSBNF. It ensures that the map operator of the functor and the substitution operator of the MN-Monad coincide.

Axiom 2 (set_Vrs).

$$\forall \alpha \in \overline{\alpha}.\ \mathrm{set}^{\alpha}_T\ x = \mathrm{RVrs}^{\alpha}_T\ x \cup \bigcup_{\mathrm{Vrs} : \overline{\tau}\ T \to \alpha\ \mathrm{set}} \mathrm{Vrs}\ x$$

As the functor only has one set per kind of variable, axiom 2 ensures that all the fine-grained free variable operators combined form the same set as this functorial set.

Axiom 3 (map_Sb).

$$\frac{\begin{array}{c}\forall i \in \{1 \ldots m_2\}.|\mathbf{supp}\ g_i| <_o |\beta_i| \wedge \mathbf{bij}\ g_i \\ \forall(\mathrm{RVrs}_i : \overline{\alpha}\ T \to \alpha_i\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\ T') \in \overline{\mathrm{Inj}_{Tj}}.\ |\{a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \wedge\ \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\rho_j\ a) \mid a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'|\end{array}}{\begin{array}{c}\mathrm{map}_T\ \overline{\mathrm{id}}\ \overline{g}\ \overline{h} \circ \mathrm{Sb}_T\ \overline{f}\ \overline{\rho} = \mathrm{Sb}_T\ \overline{f}\ \overline{(\mathrm{map}_{T'}\ \overline{id}\ \overline{g'}\ \overline{h'} \circ \rho)} \circ \mathrm{map}_T\ \overline{\mathrm{id}}\ \overline{g}\ \overline{h} \\ \text{where } \overline{g'} \subseteq \overline{g} \wedge \overline{h'} \subseteq \overline{h}\end{array}}$$

If a use of the map function does not touch the free variables, axiom 3 requires that we can commute the map function with the substitution operator Sb_T. Again, as substitution may introduce new "slots", we have to compose the mapping with each subst function.

Axiom 4 (set_Sb).

$$\frac{\begin{array}{c}\forall(\mathrm{RVrs}_i : \overline{\alpha}\ T \to \alpha_i\ \mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\ T') \in \overline{\mathrm{Inj}_{Tj}}.\ |\{a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \wedge\ \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\rho_j\ a) \mid a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'|\end{array}}{\begin{array}{c}\forall \tau \in (\overline{\beta} \cup \overline{\gamma}).\ \mathrm{set}^\tau_T\ (\mathrm{Sb}_T\ \overline{f}\ \overline{\rho}\ x) = \mathrm{set}^\tau_T\ x \\ \cup \bigcup_{(\mathrm{Inj}_j : \alpha' \to \overline{\tau'}\ T') \in \overline{\mathrm{Inj}_T} \wedge \tau \in \overline{\tau'}} \left(\bigcup_{a \in \mathrm{Vrs}_{T,j}\ x} \mathrm{set}^\tau_{T'}\ (\rho_j\ a) \right)\end{array}}$$

Axiom 5 (map_Inj).

$$\frac{\forall i \in \{1 \ldots m_2\}.|\mathbf{supp}\ g_i| <_o |\beta_i| \wedge \mathbf{bij}\ g_i}{\forall \mathrm{Inj} \in \overline{\mathrm{Inj}_T}.\ \mathrm{map}_T\ \overline{\mathrm{id}}\ \overline{g}\ \overline{h} \circ \mathrm{Inj} = \mathrm{Inj}}$$

Axioms 4 and 5 are structurally the same as axioms 6 and 15 of a MN-Monad, they just also apply the axioms to the bound and live positions of the functor.

If a MN-Monad is constructed from a MRBNF as shown in theorem 3.41, then all of the MRSBNF axioms are trivially true. We can also carry this MRSBNF structure through composition by composing the individual MRBNFs and the MN-Monad with their respective compositions. As MRSBNFs do not introduce new definitions, we only have to prove the axioms which follow directly from the axioms of the MRSBNFs that participate in the composition.

3.4.4 Closure under least fixpoint

To create a new substitution, the pre-datatype has to be a MRSBNF. Additionally, it has to have at least one injection-like constructor. For the terms of System F, this would be the left injection of the sum type (see definition 3.43). We call such a "pre-injection" η. To be eligible as the basis for new substitutions, all $\overline{\eta}$ have to fulfill certain requirements:

Definition 3.45. *Requirements for pre-injections*

For a pre-datatype $(\overline{\alpha}, \overline{\gamma})T$, with pre-injection operators $\eta_1 : \alpha' \to (\overline{\alpha}, \overline{\gamma})T, \ldots, \eta_n : \alpha'^{\ldots\prime} \to (\overline{\alpha}, \overline{\gamma})\, T$, constructing a recursive MN-Monad requires:

Axiom 1 (eta_natural).

$$\frac{\begin{array}{r}\forall(\mathrm{RVrs}_i : \overline{\alpha}\, T \to \alpha_i \,\mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\, T') \in \overline{\mathrm{Inj}_{Tj}}.\ |\{a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \wedge\ \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\rho_j\ a) \mid a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'|\end{array}}{\mathrm{Sb}_T\ \overline{f}\ \overline{\rho}\ (\mathrm{Map}_T\ \overline{g}\ x) = \eta_i\ (f'\ a), \text{where } f' \in \overline{f}, \text{for all } i \in \{1 \ldots n\}}$$

The parametricity stated in axiom 1 ensures that the pre-injection operator only depends on the variable kind it is responsive for.

Axiom 2 (eta_inversion).

$$\frac{\begin{array}{r}\forall(\mathrm{RVrs}_i : \overline{\alpha}\, T \to \alpha_i \,\mathrm{set}) \in \overline{\mathrm{RVrs}_T}.\ |\mathbf{supp}\ f_i| <_o |\alpha_i| \\ \forall(\mathrm{Inj}_j : \alpha \to \overline{\alpha'}\, T') \in \overline{\mathrm{Inj}_{Tj}}.\ |\{a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha| \\ \wedge\ \forall \alpha' \in \overline{\alpha'}.\ |\{\mathrm{FVars}^{\alpha'}_{T'}\ (\rho_j\ a) \mid a.\ \rho_j\ a \neq \mathrm{Inj}_j\ a\}| <_o |\alpha'| \\ \mathrm{Sb}_T\ \overline{f}\ \overline{\rho}\ (\mathrm{Map}_T\ \overline{g}\ x) = \eta_i\ a\end{array}}{\exists y.\ x = \eta_i\ y, \text{for all } i \in \{1 \ldots n\}}$$

Axiom 2 requires the pre-injection to be *structural*, i.e., no matter now you apply maps and substitutions, if you end up with a pre-injection it means that you already started with a pre-injection.

Axiom 3 (eta_mem).

$$a \in \mathrm{RVrs}^{\alpha}_{T}\ (\eta_i\ a), \text{for all } i \in \{1 \ldots n\}$$

Axiom 3 ensures that the variable of the pre-injection has to be part of the functorial free variables of the type.

Axiom 4 (eta_inj).

$$\frac{\eta_i\ a = \eta_i\ b}{a = b, \text{for all } i \in \{1 \ldots n\}}$$

Axiom 5 (eta_distinct).

$$\forall i, j \in \{1 \ldots n\}.\ i \neq j \longrightarrow \eta_i\ a \neq \eta_j\ b$$

The last two axioms state that pre-injections are injective and that they may not overlap.

Then we can define the new fine-grained free variable operators for the type using inductive predicates that return only the free variables associated with their injection (see definition 3.46) and those that are left over (see definition 3.47). These predicates will later be used for the *Vrs* and *RVrs* operators of the recursive MN-Monad (see definition 3.48).

Definition 3.46. *Fine-grained free variable predicate for pre-injections*

$$\frac{}{Vfree_i\ a\ (ctor_{term}\ (\eta_i\ x))} \qquad \frac{z \in set_{rec}\ x \qquad Vfree_i\ a\ z}{Vfree_i\ a\ (ctor_{term}\ x)}$$

$$\frac{z \in set_{brec}\ x \qquad Vfree_i\ a\ z \qquad a \notin set_{bound}\ x}{Vfree_i\ a\ (ctor_{term}\ x)}$$

Definition 3.47. *Fine-grained free variable predicate for left-over variables*

$$\frac{a \in RVrs^{\alpha}_{T}\ x \qquad \forall i \in \{1 \ldots n\}.\ \forall b.\ x \neq \eta_i\ b}{Rfree_{\alpha}\ a\ (ctor_{term}\ x)} \qquad \frac{z \in set_{rec}\ x \qquad Rfree_{\alpha}\ a\ z}{Rfree_{\alpha}\ a\ (ctor_{term}\ x)}$$

$$\frac{z \in set_{brec}\ x \qquad Rfree_{\alpha}\ a\ z \qquad a \notin set_{bound}\ x}{Rfree_{\alpha}\ a\ (ctor_{term}\ x)}$$

Definition 3.48. *Fine-grained free variable operators*

$$\begin{aligned} \boldsymbol{Vrs}_i &:= \lambda t.\ \{a.\ Vfree\ a\ t\} \\ \boldsymbol{RVrs}_{\alpha} &:= \lambda t.\ \{a.\ Rfree_{\alpha}\ a\ t\} \end{aligned}$$

The last missing operator for a MN-Monad is the substitution operator itself. Using the recursor from section 3.3.4 or the corecursor from section 3.3.6, it is possible to define a suitable function. We only describe the instantiation of the recursor here, but it directly mirrors the instantiation of the corecursor:

Definition 3.49 (Recursor locale instantiation for substitution)**.** *Assuming functions $\overline{f}$ and $\overline{\rho}$ of small support are fixed in the surrounding context, we can define substitution by instantiating the term recursor locale with:*

$$\begin{array}{lcl} A_\alpha & := & imsupp\ f_\alpha \cup \overline{IImsupp\ \rho_\alpha} \\ Udtor & := & \lambda x.\ \textbf{\textit{case}}\ map_{term\_pre}\ \overline{id}\ \overline{fst}\ x\ \textbf{\textit{of}} \\ & & \quad \mid ctor_{term}\ (\eta_i\ a) \ \Rightarrow \rho_i\ a \\ & & \quad \mid \_ \qquad\qquad\ \Rightarrow ctor_{term}\ (map_{term\_pre}\ \overline{id}\ \overline{snd}\ x) \end{array}$$

With these operators we can define the MN-Monad structure on the term type (see definition 3.50). Following that, we can also prove the MRSBNF axioms to connect the existing functorial structure with the new monadic structure.

Definition 3.50. *MN-Monad structure for recursive types*

$$\begin{array}{lcl} Ops & := & \{L = \overline{\alpha}\ T\} \cup \left(Ops_{T\_pre} \setminus L_{T\_pre}\right) \\ bd & := & bd_{T\_pre} \\ \overline{Inj_L} & := & \overline{ctor_{term} \circ \eta_i} \\ \overline{Vrs_L} & := & \overline{Vrs_i} \\ \overline{RVrs_L} & := & \overline{RVrs_\alpha} \\ Sb_L & := & REC_{subst} \end{array}$$

3.5 High-level syntax sugar

At this point in the datatype construction pipeline, the type and all the important operators are defined. However, all the theorems, even those that the user is supposed to interact with like induction, refer to the internal single constructor of the type. Thus the last step of the construction is to lift these internal theorems to a constructor-based variant. The constructors themselves are defined in terms of the single internal constructor, here for example for the untyped λ-calculus:

Definition 3.51. *Constructors for the untyped λ-calculus*

$$\begin{array}{lcll} \boldsymbol{Var} & := & \lambda x. & ctor_{term}\,(Inl\,x) \\ \boldsymbol{App} & := & \lambda t_1\,t_2. & ctor_{term}\,(Inl\,(Inr\,(t_1,t_2)) \\ \boldsymbol{Lam} & := & \lambda x\,t. & ctor_{term}\,(Inl\,(Inr\,(Inr\,(x,t))) \end{array}$$

Then, using case distinction on the pre-datatype, we can provide for example simplification rules for substitution (see theorem 3.42) or a constructor-based strong induction theorem (see theorem 3.43).

Theorem 3.42. *Simplification rules of substitution in the untyped λ-calculus For a function $\rho : \alpha \to \alpha$ term of small support, the follow holds:*

$$\begin{array}{rcll} & & (Var\,x)[\rho] & = & \rho\,x \\ & & (App\,t_1\,t_2)[\rho] & = & App\,(t_1[\rho])\,(t_2[\rho]) \\ x \notin IImsupp\,\rho & \longrightarrow & (Lam\,x\,t)[\rho] & = & Lam\,x\,(t[\rho]) \end{array}$$

Theorem 3.43. *Strong induction for the untyped λ-calculus*

$$\frac{\begin{array}{c} \forall \rho \in \mathcal{P}.\, |K\ \rho| <_o |\alpha| \\ \forall x\ \rho.\, P\,(Var\,x)\,\rho \qquad \forall t_1\ t_2\ \rho.\, (\forall \rho.\, P\ t_1\ \rho) \longrightarrow (\forall \rho.\, P\ t_2\ \rho) \longrightarrow P\,(App\,t_1\,t_2)\,\rho \\ \forall x\ t\ \rho.\, x \notin K\ \rho \longrightarrow (\forall \rho.\, P\ t\ \rho) \longrightarrow P\,(Lam\,x\,t)\,\rho \end{array}}{\forall\, \rho \in \mathcal{P}.\, P\ t\ \rho}$$

Chapter 4

Strong rule induction for inductive predicates

Contents

4.1 Preliminaries 75

4.1.1 Nominal sets 75

4.1.2 The Knaster-Tarski fixpoint theorem 77

4.2 Strengthening induction theorems for inductive definitions 77

For a formalization, providing the syntax of a calculus is only the start of it. We also need to be able to define relations on the terms of our language. Isabelle already supports inductively defined relations and predicates that we can use here. For example, going back to the example of the untyped λ-calculus, we can define β-reduction as an inductive relation between two terms:

Definition 4.1. *β-reduction of the untyped λ-calculus*

$$\frac{}{(\lambda x.\, t_1)\, t_2 \to_\beta t_1[t_2/x]}\ \textsc{Beta} \qquad \frac{t_1 \to_\beta t_1'}{t_1\, t_2 \to_\beta t_1'\, t_2}\ \textsc{AppL}$$

$$\frac{t_2 \to_\beta t_2'}{t_1\, t_2 \to_\beta t_1\, t_2'}\ \textsc{AppR} \qquad \frac{t \to_\beta t'}{\lambda x.\, t \to_\beta \lambda x.\, t'}\ \textsc{Xi}$$

If we want to prove a theorem about this relation, we can either use induction on one of the two arguments to the relation, or we can proceed by induction on the relation itself. While using induction on the term type would allow us to use fresh induction to make the proof easier, it is also requires rule inversions in each of the inductive cases. For example in the application case on the left side of the reduction, we could be in any of the reduction steps aside from Xi, requiring a case distinction.

In general, it is easier to use induction on the relation itself. From the definition of the relation, we obtain the following induction theorem:

Theorem 4.1. *Induction rule for β-reduction*

$$\frac{\begin{array}{c} t_1 \to_\beta t_2 \qquad \forall x\ t_1\ t_2.\ P\ ((\lambda x.\ t_1)\ t_2)\ (t_1[t_2/x]) \\ \forall t_1\ t_2\ t_1'.\ t_1 \to_\beta t_1' \longrightarrow P\ t_1\ t_1' \longrightarrow P\ (t_1\ t_2)\ (t_1'\ t_2) \\ \forall t_1\ t_2\ t_2'.\ t_2 \to_\beta t_2' \longrightarrow P\ t_2\ t_2' \longrightarrow P\ (t_1\ t_2)\ (t_1\ t_2') \\ \forall x\ t\ t'.\ t \to_\beta t' \longrightarrow P\ t\ t' \longrightarrow P\ (\lambda x.\ t)\ (\lambda x.\ t') \end{array}}{P\ t_1\ t_2}$$

Similar to the strong induction on terms (see theorem 3.14), we would like to incorporate extra freshness assumptions in the inductive cases that have "naked" binders. Here this applies to the Beta and Xi cases. But when is such a strengthening sound and how can such a strengthened induction theorem be constructed irrespective of the exact shape of the inductive definition?

4.1 Preliminaries

4.1.1 Nominal sets

Nominal sets were first introduced by Gabbay and Pitts [21, 22]. Given a fixed set of *variables* or *atoms* $\mathbb{A}$, let $Perm(\mathbb{A}) = (\mathbb{A} \to \mathbb{A}, \circ, 1_{\mathbb{A}})$ be the group of all permutations of $\mathbb{A}$. This means that any element of the group is an endobijection on $\mathbb{A}$. We write $(a \leftrightarrow b)$ to denote the *swapping* permutation that sends a to b, b to a, and all other variables to themselves[1].

A *pre-nominal set* $\mathcal{P} = (T, \_[\_]^{\mathcal{P}} : T \to (\mathbb{A} \to \mathbb{A}) \to T)$ is a set equipped with a permutation action that applies a permutation π to an element of the set T. Given a fixed permutation σ, the function $\_[\sigma]^{\mathcal{P}} : T \to T$ is an endobijection on T.

Given an element $a \in T$ of a pre-nominal set $\mathcal{T} = (T, \_[\_]^{\mathcal{T}})$, a set $X \subseteq \mathbb{A}$ *supports* a if $a[x \leftrightarrow y]^{\mathcal{P}} = a$ holds for all $x, y \in \mathbb{A} \setminus X$. Such an element is called *finitely supported* if there exits a finite set X that supports it.

A *nominal set* is a pre-nominal set where every element is finitely supported. For every element a of such a set, it can be shown that the smallest set that supports

[1] In nominal logic this permutation is usually denoted as $(a\ b)$. We chose to diverge here to avoid confusion with function application.

a exists and is called the *support* of a (denoted as $Supp^{\mathcal{T}}\ a$).

The set of functions $F = (A \to B)$ between two pre-nominal sets $\mathcal{A} = (A, \_[\_]^{\mathcal{A}})$ and $\mathcal{B} = (B, \_[\_]^{\mathcal{B}})$ also forms a pre-nominal $\mathcal{F} = (F, \_[\_]^{\mathcal{F}})$ by defining $f[\sigma]^{\mathcal{F}}$ to be the function that sends every $a \in A$ to $f(a[\sigma^{-1}]^{\mathcal{A}})[\sigma]^{\mathcal{B}}$. If we restrict the set F to the set of function with finite support, $\mathcal{F}$ also forms a nominal set.

In addition to the above function-space construction, nominal set structures can also be naturally defined on the products, sums, container-type extensions (such as lists or trees) and quotients of the carrier sets, overall enjoying good category-theoretic properties, in particular forming a topos equivalent to the Schanuel topos [52].

If we relax the finite-support assumption of a nominal set and replace it with a *small-support* assumption, we get what we call a *loosely-supported nominal set*. In this setting, smallness is relative to a given infinite cardinal. For $\aleph_0$ we obtain a normal nominal set.

We also relax the requirement that the support has to be minimal. In infinite structures there also may not be such a smallest supporting set. Take for example the type of streams of atoms, quotiented by almost-everywhere-equality, i.e., $\mathbb{A}\ stream\ //\ \{(xs, ys).\ finite\ \{i.\ xs_i \neq ys_i\}\}$. This means that for any two streams in the same equivariance class, there exists an index i after which the two streams are equal. If we then take a stream xs from this quotient where all elements are pairwise disjoint, we can prove that the set of variables in xs starting from index n supports xs, for any n. The reason is that if we apply a permutation π which is the identity on the variables starting at index n to a stream xs, we get a stream xs' that differs from xs in at most $n - 1$ positions. Thus xs' is in the same equivalence class as xs. Clearly, the supporting set starting at index $n + 1$ is a subset of the set starting at index n, however this chain of subsets is infinite and thus there exists no smallest supporting set.

Concretely, all of the binder data- and codatatypes from chapter 3 form (loosely-supported) nominal sets by using the *permute* function as the permutation action $\_[\_]$. We can also show that the set of free variables of a term t (which is defined syntactically) coincides with its support (which is defined via the permuation operator). In fact, it is known that this coincidence holds for all syntaxes with statically scoped bindings [51].

4.1.2 The Knaster-Tarski fixpoint theorem

The well known Knaster-Tarski fixpoint theorem [59] offers a simple yet powerful foundation for induction. Let $(L, \leq)$ be a complete lattice and $G : L \to L$ a monotonic operator. Then there exists a (unique) least fixpoint I_G for G, such that $G\, I_G = I_G$ and $\forall k \in L.\, G\, k = k \longrightarrow I_G \leq k$. I_G is also the least pre-fixpoint, meaning that $\forall k \in L. G k \leq k \longrightarrow I_G \leq k$. Finally, a useful variation of the theorem also holds, where $\wedge$ is the binary infimum in L: $\forall k \in L.\, G\, (I_G \wedge k) \leq k \longrightarrow I_G \leq k$.

It is this last part that enables inductive reasoning. To prove that $I_G \leq k$ it is sufficient to prove that $G\, (I_G \wedge k) \leq k$. In our work we will only make use of this theorem on the lattice of predicates (equivalently, the lattice of subsets) as initially formulated by Knaster [30]. Given a set A and two predicates $\phi, \psi : A \to \text{Bool}$, $\phi \leq \psi$ is defined as component-wise implication, i.e. $\forall a \in A.\, \phi\, a \longrightarrow \psi\, a$. This also generalizes to n-ary predicates if we take A to be the product $A_1 \times \cdots \times A_n$. Often, the operator G is given by a set of rules (see e.g. definition 4.1). For this reason the induction principle for I_G is referred to as *rule induction*.

4.2 Strengthening induction theorems for inductive definitions

Theorem 4.2 shows the induction rule we would like to have for β-reduction. The main differences to the normal induction rule are highlighted in grey.

Theorem 4.2. *Strong induction rule for β-reduction*

$$\frac{\begin{array}{c} t_1 \to_\beta t_2 \qquad \forall \rho.\, \rho \in \mathcal{P} \longrightarrow |K\, \rho| <_o |\alpha| \\ \forall x\, t_1\, t_2\, \rho.\, x \notin K\, \rho \longrightarrow \rho \in \mathcal{P} \longrightarrow P\ ((\lambda x.\, t_1)\, t_2)\, (t_1[t_2/x])\ \rho \\ \forall t_1\, t_2\, t_1'\, \rho.\, t_1 \to_\beta t_1' \longrightarrow (\forall \rho \in \mathcal{P}.\, P\, t_1\, t_1'\, \rho) \longrightarrow \rho \in \mathcal{P} \longrightarrow P\, (t_1\, t_2)\, (t_1'\, t_2)\, \rho \\ \forall t_1\, t_2\, t_2'\, \rho.\, t_2 \to_\beta t_2' \longrightarrow (\forall \rho \in \mathcal{P}.\, P\, t_2\, t_2'\, \rho) \longrightarrow \rho \in \mathcal{P} \longrightarrow P\, (t_1\, t_2)\, (t_1\, t_2')\, \rho \\ \forall x\, t\, t'\, \rho.\, x \notin K\, \rho \longrightarrow t \to_\beta t' \longrightarrow (\forall \rho \in \mathcal{P}.\, P\, t\, t'\, \rho) \\ \longrightarrow \rho \in \mathcal{P} \longrightarrow P\, (\lambda x.\, t)\, (\lambda x.\, t')\, \rho \end{array}}{\forall \rho \in \mathcal{P}.\, P\, t_1\, t_2\, \rho}$$

While for β-reduction such a strengthening is sound, this is not true in general. Urban et al. [63] note the reason for this: In inductive definitions variables can appear both in a binding and non-binding position at once. Note that this is also

the case for β-reduction: In the *Beta* case, the variable x appears as the binder of a λ-function, as well as a free variable in the substitution on the right. The reason this is sound here is due to substitution being "well behaved".

The main question then is what exactly constitutes being "well behaved". For this, we first need to look at how inductive relations and predicates are represented in Isabelle. As hinted at in section 4.1.2, the rules that define the inductive relation get translated to a monotonic operator G:

Definition 4.2. *Operator of β-reduction*

$$\begin{aligned} G_{(\to_\beta)}\, R\, x_1\, x_2 \;:=\; & (\exists x\, t_1\, t_2.\, x_1 = ((\lambda x.\, t_1)\, t_2) \wedge x_2 = (t_1[t_2/x])) \\ \vee\; & (\exists t_1\, t_1'\, t_2.\, x_1 = (t_1\, t_2) \wedge x_2 = (t_1'\, t_2) \wedge R\, t_1\, t_1') \\ \vee\; & (\exists t_1\, t_2\, t_2'.\, x_1 = (t_1\, t_2) \wedge x_2 = (t_1\, t_2') \wedge R\, t_2\, t_2') \\ \vee\; & (\exists x\, t\, t'.\, x_1 = (\lambda x.\, t) \wedge x_2 = (\lambda x.\, t') \wedge R\, t\, t') \end{aligned}$$

Definition 4.3. *Internal definition of β-reduction* $(\to_\beta) := \mathit{lfp}\, G_{(\to_\beta)}$

Then β-reduction exists as the least fixpoint of the operation $G_{(\to_\beta)}$ (see definition 4.3). To strengthen the associated induction principle, we need to make the aforementioned "naked" bound variables in the conclusion of the rules explicit. We do this by adding an extra argument to the operator (see definition 4.4). Again, differences to the normal operator are highlighted in grey.

Definition 4.4. *Modified operator of β-reduction*

$$\begin{aligned} G'_{(\to_\beta)}\, R\, B\, x_1\, x_2 \;:=\; & \Big(\exists x\, t_1\, t_2.\, B = \{x\} \wedge x_1 = ((\lambda x.\, t_1)\, t_2) \wedge x_2 = (t_1[t_2/x])\Big) \\ \vee\; & \Big(\exists t_1\, t_1'\, t_2.\, B = \{\} \wedge\; x_1 = (t_1\, t_2) \wedge x_2 = (t_1'\, t_2) \wedge R\, t_1\, t_1'\Big) \\ \vee\; & \Big(\exists t_1\, t_2\, t_2'.\, B = \{\} \wedge x_1 = (t_1\, t_2) \wedge x_2 = (t_1\, t_2') \wedge R\, t_2\, t_2'\Big) \\ \vee\; & \Big(\exists x\, t\, t'.\, B = \{x\} \wedge x_1 = (\lambda x.\, t) \wedge x_2 = (\lambda x.\, t') \wedge R\, t\, t'\Big) \end{aligned}$$

For this transformation to make sense, we assume that all arguments that contain variables form a (loosely-supported) nominal set (see chapter 4.1.1). The key notion to ensure soundness rests on the bound variables being *refreshable* in the rules, i.e. we can always rename them to become fresh for a rule's entire conclusion (and not just the location where they are bound) without invalidating its hypotheses. To make this notion precise we need to introduce *equivariance*, a key property of functions in nominal logic:

Definition 4.5. *Given two pre-nominal sets* $\mathcal{A} = (A, \_[\_]^{\mathcal{A}})$ *and* $\mathcal{B} = (B, \_[\_]^{\mathcal{B}})$*, a function* $F : A \to B$ *is called* ***equivariant*** *if it commutes with the permutation actions:* $\forall a \in A.\, F(a[\sigma]^{\mathcal{A}}) = (F\ a)[\sigma]^{\mathcal{B}}$.

Since the two-element set of Booleans (like any set) can be trivially equipped with the identity permutation action to become a (pre-)nominal set, we can speak of the equivariance of predicates $\phi : A \to \text{Bool}$ where $\mathcal{A} = (A, \_[\_]^{\mathcal{A}})$ is a pre-nominal set. Here equivariance can be equivalently expressed using implication: $\forall a \in A.\, \phi\ a \longrightarrow \phi(a[\sigma]^{\mathcal{A}})$.

Refreshability and the stronger property *freshness* can be expressed in terms of the modified operator G':

Definition 4.6. *Given a nominal set* $\mathcal{T} = (T, \_[\_]^{\mathcal{T}})$*, an operator* $G : (T \to \mathit{Bool}) \to (\mathbb{A} \to T \to \mathit{Bool})$ *is said to be:*

- $\mathcal{T}$*-refreshable when, for all* $\phi : T \to \mathit{Bool}$*,* $B \subset_{\mathit{fin}} \mathbb{A}$ *and* $t \in T$*, if* ϕ *is equivariant then* $G\ \phi\ B\ t \longrightarrow \exists B'.\, B' \cap \mathit{Supp}^{\mathcal{T}}\ t = \{\} \wedge G\ \phi\ B'\ t$
- $\mathcal{T}$*-fresh when, for all* $\phi : T \to \mathit{Bool}$*,* $B \subset_{\mathit{fin}} \mathbb{A}$ *and* $t \in T$*,* $G\ \phi\ B\ t \longrightarrow B \cap \mathit{Supp}^{\mathcal{T}}\ t = \{\}$

Note that $\mathcal{T}$-freshness implies $\mathcal{T}$-refreshability by taking $B' = B$. And indeed, $\mathcal{T}$-refreshability together with equivariance of $G_{(\to_\beta)}$ (which essentially ensures robustness of the rules in the refreshing process) is sufficient to derive the strong rule induction theorem for any given inductive predicate.

First, we define the relation $II_{(\to_\beta)}$ that uses the modified operator $G'_{(\to_\beta)}$ and prove that it is the same relation as the original β-reduction. The proof follows by straight forward induction.

Definition 4.7. $(\boldsymbol{II}_{(\to_\beta)}) := \mathit{lfp}\ (\lambda R\ x_1\ x_2.\, \exists B.\, G'_{(\to_\beta)}\ R\ B\ x_1\ x_2)$

Theorem 4.3. $(II_{(\to_\beta)}) = (\to_\beta)$

From the equivariance of $G_{(\to_\beta)}$ we can derive that $II_{(\to_\beta)}$ is also equivariant (see theorem 4.4). For the rest of this chapter we write the permutation action of a nominal set as $\_[\sigma]$ where σ is a permutation, and *Supp* for the support.

Theorem 4.4. $II_{(\to_\beta)}\ x_1\ x_2 \longrightarrow II_{(\to_\beta)}\ (x_1[\sigma])\ (x_2[\sigma])$

Next, we define the relation $II'_{(\to_\beta)}$ that builds in half of the required freshness: It assumes that the bound variables are disjoint from the support, i.e. the free variables, of the arguments:

Definition 4.8.

$$(\boldsymbol{II'_{(\to_\beta)}}) := \mathit{lfp}\,(\lambda R\, x_1\, x_2.\, \exists B.\, B \cap (\mathit{Supp}\, x_1 \cup \mathit{Supp}\, x_2) = \{\} \wedge G'_{(\to_\beta)}\, R\, B\, x_1\, x_2)$$

This new relation obviously implies $II_{(\to_\beta)}$. To prove that the other direction also holds, we need to prove that $II'_{(\to_\beta)}$ is equivariant similar to theorem 4.4. Then we can use refreshability of $G_{(\to_\beta)}$ to replace the existential B with a B' that is disjoint from the support (see theorem 4.5).

Theorem 4.5. $(II'_{(\to_\beta)}) = (II_{(\to_\beta)})$

Finally we can strengthen the induction theorem for $II'_{(\to_\beta)}$ to also factor in freshness with regard to an arbitrary parameter structure $\mathcal{P}$ (see theorem 4.6). Given that the bound variables are already disjoint from the support, we can replace them with new (fresh) variables that are disjoint from the parameter and the support without affecting the free variables. We also need to prove this theorem for all possible permutations in order for the proof to work.

Theorem 4.6.

$$\frac{\begin{array}{r}\forall \rho.\, \rho \in \mathcal{P} \longrightarrow |K\, \rho| <_o |\alpha| \qquad II'_{(\to_\beta)}\, x_1\, x_2 \\ \forall B\, x_1\, x_2\, \rho.\, B \cap (\mathit{Supp}\, x_1 \cup \mathit{Supp}\, x_2) = \{\} \longrightarrow B \cap K\, \rho = \{\} \longrightarrow \rho \in \mathcal{P} \\ \longrightarrow G'_{(\to_\beta)}\, (\lambda y_1\, y_2. II'_{(\to_\beta)}\, y_1\, y_2 \wedge \forall \rho \in \mathcal{P}.\, R\, y_1\, y_2\, \rho)\, B\, x_1\, x_2 \longrightarrow R\, x_1\, x_2\, \rho\end{array}}{\forall \sigma.\, \forall \rho \in \mathcal{P}.\, R\, (x_1[\sigma])(x_2[\sigma])\, \rho}$$

Using the fact that all three definitions actually define the same relation, we can simplify the theorem:

Theorem 4.7.

$$\frac{\begin{array}{r}\forall \rho.\, \rho \in \mathcal{P} \longrightarrow |K\, \rho| <_o |\alpha| \qquad x_1 \to_\beta x_2 \\ \forall B\, x_1\, x_2\, \rho.\, B \cap (\mathit{Supp}\, x_1 \cup \mathit{Supp}\, x_2) = \{\} \longrightarrow B \cap K\, \rho = \{\} \longrightarrow \rho \in \mathcal{P} \\ \longrightarrow G'_{(\to_\beta)}\, (\lambda y_1\, y_2.\, y_1 \to_\beta y_2 \wedge \forall \rho \in \mathcal{P}.\, R\, y_1\, y_2\, \rho)\, B\, x_1\, x_2 \longrightarrow R\, x_1\, x_2\, \rho\end{array}}{\forall \rho \in \mathcal{P}.\, R\, x_1\, x_2\, \rho}$$

The last step is to lift this internal theorem back to the rule-based format that the user expects using the definition of $G'_{(\rightarrow_\beta)}$ (see theorem 4.2).

Chapter 5

Implementation

Contents

5.1 Manually constructing example proofs 83

5.2 Generalizing the example 84

5.3 Automating the construction with Isabelle/ML 85

5.4 Major refactorings during the development 88

5.4.1 Variable type class registry 88

5.4.2 Using locales for the recursor 89

5.5 Statistics 90

At the time of writing, the full datatype construction as well as most of the codatatype construction from chapter 3 are implemented in Isabelle/ML. For codatatypes, defining the renaming function and proving the MRBNF axioms using it is still missing. The strengthening of inductive predicates from chapter 4 is also implemented, except automation for proving refreshability.

In general, the implementation process followed three distinct steps: first, manually construct the proofs and definitions necessary for the given part of the automation, for example the fixpoint construction. Contrary to the usual Isabelle practice, we do not use any high-level Isabelle proof methods for this, but only use simple single step tactics (see section 5.1). The second step is to generalize the example to cover all of the desired features mentioned in chapter 2 (see section 5.2). As the implementation has grown over a timeframe of over four years, several important refactorings took place that we show in section 5.4. Finally, we generate the proof goals and definitions with ML and port over the proofs from our example (see section 5.3). In this step we also make sure that the ML code can handle an arbitrary syntax, i.e. the code has to work independent of the number of mutual types, different kinds of variables, etc.

5.1 Manually constructing example proofs

Before we can think about automating anything, we need to know what to automate in the first place. So the first step is to manually construct the relevant proofs and definitions for a given example syntax. At this point we are still figuring out how the construction should work, so we use the simplest syntax available, i.e. the untyped λ-calculus.

For some parts of the construction, for example the binder fixpoint and the recursor, we do not have to start from scratch as we can use the proof artifact of Blanchette et al. [9] as a starting point. However, given that the artifact was not meant as a template for automation and does not run in modern versions of Isabelle, we will only use its proofs as a rough roadmap while redoing the proofs themselves.

No matter if we know the rough structure of the proofs or not, we will need to write the proofs without using any of Isabelle's high-level proof tactics like simp, auto, blast or metis. While those proof methods can drastically shorten proofs and take a lot of work away from the user, they are too unpredictable to use in our automation. Given that the automation will need to deal with arbitrarily complex syntax, the chance of the tactics suddenly failing in a way that is opaque to the user is too high. Beyond just failing, these high-level tactics may not terminate depending on the goal and the supplied theorems. All of this means that we restrict ourselves to the following simple, single-step proof methods:

- rule, erule, drule, frule
- unfold
- subst
- insert
- rotate_tac
- prefer, defer

Furthermore, we do not use Isabelle's Isar proof language that helps with writing proofs that are meant to be read without having to look at the proof state itself. To make it easier to port the proof to ML, we only use apply scripts, i.e. long chains of backwards reasoning via the simple single-step tactics.

The last unusual restriction we impose on ourselves is that we try to avoid explicit instantiation of theorems as much as possible. In Isabelle/ML, when constructing terms it is necessary to manually supply the correct types to all subterms. While some small helper function can alleviate some of that, constructing well-typed terms is still an annoying process. To avoid this as much as possible, we rely on Isabelle's higher-order unification. In particular, using *erule* helps tremendously here as we often use theorems that have a function application in their conclusion where both the function and its arguments are variables. Naive unification would choose identity as the function and the rest as argument. However, we usually have an assumption about the function in our current proof state. By unifying not only the goal, but also the assumption, the function is unified correctly. To give a concrete example, let's assume we have the following proof state (on the left) where we want to use the elimination rule of *id_on* (on the right) to proceed:

$$\frac{\textbf{id\_on}\ A\ f_1 \qquad \textbf{id\_on}\ B\ f_2 \qquad x \in B}{f_2\ x = x} \qquad\qquad \frac{\textbf{id\_on}\ X\ g \qquad y \in X}{g\ y = y}$$

Using *rule* to unify the goal with the conclusion of the rule requires higher-order unification on both the function and the variable (as both g and y are meta variables in the elimination rule). Isabelle will choose the identity function for g and $f_2\ x$ for y, which is not what we want. If we were to use *drule* to do forwards reasoning, the first assumption of the elimination rule would unify with the first assumption in our goal, resulting in the goal $x \in A$ which is not provable. By using *erule* which does both unifications at the same time, we side-step this issue and get the desired result.

5.2 Generalizing the example

Once we have a suitable example construction, we need to generalize it to cover all of the features the framework supports (see chapter 2). This means going from the syntax of the untyped λ-calculus to a custom syntax with two mutually recursive types and two different kinds of variables that are additionally bound in another type. Concretely we use the following syntax for the generalized example:

```
binder_datatype ('var, 'tyvar, 'a, 'b) T1 =
  A1 'var
```

```
| B1
| C1 'tyvar
| D1 "('var, 'tyvar, 'a, 'b) T1" "('var, 'tyvar, 'a, 'b) T2"
| E1 x::'var "(xs::'var * unit) list" binds x in xs
| F1 x::'var t::"('var, 'tyvar, 'a, 'b) T1" binds x in t
| G1 x::'tyvar t::"('var, 'tyvar, 'a, 'b) T1" binds x in t
| H1 'a
and ('var, 'tyvar, 'a, 'b) T2 =
  A2 'var
| B2 'tyvar
| C2 "('var, 'tyvar, 'a, 'b) T1" "('var, 'tyvar, 'a, 'b) T2"
| D2 x::'var "(xs::'var) list" binds x in xs
| E2 x::'tyvar t::"('var, 'tyvar, 'a, 'b) T2" binds x in t
| F2 'b "('var, 'tyvar, 'a, 'b) T2"
```

We do this as a second step and not straight away because the generalization is difficult enough even when knowing how the proof works. Developing the proof directly for the complex syntax would most likely be too hard.

The generalization step introduces a lot of proof duplication in the apply scripts as the same sequence of tactics is used for example for both variable kinds. We use comments in the example proof to make note of such duplication so we can later use a loop at this point. Similarly, we also annotate magic numbers, for example when rotating premises of a theorem. Usually these numbers depend on the number of different kinds of variables and/or the number of mutually recursive types.

5.3 Automating the construction with Isabelle/ML

The last step is to write the actual automation. Isabelle's logic is implemented in Standard ML and allows to load user-written ML files that can interface with Isabelle's core APIs. Theorems are represented by the abstract *thm* type which can only be constructed by the Isabelle kernel. This way, Isabelle does not need proof objects to witness proofs as the only way to get a theorem is by going through the kernel. All other tactics and high-level proof methods are built on top of the API the kernel provides.

Terms in Isabelle/ML come in two varieties: the *term* type, which captures the inner syntax of Isabelle and *cterm*, short for certified term, which represents terms that are well-typed and well-scoped. While we can directly manipulate the former

via its constructors, the latter is abstract, similar to the *thm* type. Only the kernel may certify a term, ensuring soundness.

The overall structure of an Isabelle proof script is a sequence of *commands*. Library authors can register new commands that are implemented in ML. The implementation will receive a *local theory* which represents the context of the command. For example, all available theorems and definitions are stored in the local theory. The command may read facts and definitions from the context, introduce new definitions and theorems and store them in a new local theory that the command will return at the end. This way, the local theory is never directly mutated which allows Isabelle to jump back in a proof script without suddenly having access to theorems defined later in the script.

The `binder_datatype` command is such a command. However, given its complexity, the implementation is split into smaller subcomponents roughly corresponding to the sections of chapter 3. Each subcomponent then follows a similar structure to Isabelle's locales: it is a function that expects some constants and tactics that prove some axioms about the constants and it returns new definitions and theorems about them.

Within those functions, the main effort left is to construct the terms for the proof goals of the relevant theorems. In Isabelle's internal term representation, every constant carries its own type. For example, compare a simple lemma used in the corecursor construction both in apply script style and in Isabelle/ML (see figure 5.1):

```
lemma Umap_id:
  "valid_U1 d1 ⟹ Umap1 id id d1 = d1"
  "valid_U2 d2 ⟹ Umap2 id id d2 = d2"
   apply -
  apply (rule Umap_cong0)
  apply assumption
        apply (rule bij_id supp_id_bound id_apply)+
  (* repeated *)
  apply (rule Umap_cong0)
  apply assumption
        apply (rule bij_id supp_id_bound id_apply)+
  done

val Umap_ids = @{map 4} (fn valid_prem => fn d => fn Umap => fn Umap_cong0 =>
  let val goal = mk_Trueprop_eq (Term.list_comb (Umap, map HOLogic.id_const vars) $ d, d)
  in Goal.prove_sorry lthy (names [d]) [valid_prem] goal (fn {context=ctxt, prems} => EVERY1 [
    rtac ctxt Umap_cong0,
    resolve_tac ctxt prems,
    REPEAT_DETERM o resolve_tac ctxt @{thms bij_id supp_id_bound id_apply}
  ]) end
) valid_prems ds Umaps Umap_cong_ids;
```

Figure 5.1: A simple lemma in apply script and Isabelle/ML

There several noteworthy aspects about the ML code: Like most Isabelle/ML implementations we make use of *antiquotations* (denoted by `@{<content>}`) that are provided by Isabelle. These antiquotations are a kind of meta-programming as they will be expanded to ML code at compile time. The first antiquotation generates a map function that maps over four lists simultaneously. If the lists do not have the same length, the function will throw an error. Given that the theorem is the same between both mutually recursive types with only the constants being different, we pervasively use such multi-argument maps over lists of constants and variables. The other use of an antiquotation is in the tactic itself to obtain references to theorems defined in user-land Isabelle.

The actual goal generation happens inside the map. While normally Isabelle automatically inserts coercions between the HOL type *bool* and the underlying type *prop*, in ML those coercions are explicit. The function `mk_Trueprop_eq` thus is a small helper that is the composition of the coercions and HOL's equality. Here we can also see the need to explicitly provide types since instead of replicating the identity constant several times, we have to explicitly map it over a list of types.

The tactic itself however closely resembles the apply script proof. This is the main reason why the example proofs are done in apply script style in the first place. There is a direct one-to-one mapping of apply script tactics to ML tactics.

5.4 Major refactorings during the development

As with any implementation work, we did not always arrive at the ideal implementation from the get-go. To also shed some light on earlier design iterations, we want to present aspects of previous iterations of the implementation and why they were replaced.

5.4.1 Variable type class registry

To ensure that the types used for variables are big enough, we attach a type class to the bound and free positions of every MRBNF that assumes the type is at least as big as the cardinal bound of the MRBNF itself. So for a finite datatype, the cardinal bound would be $\aleph_0$ and the type class would require the variable type to be at least countable.

At the beginning of our work, we reused parts of the proof artifact from Blanchette et al.'s work on MRBNFs [9]. The core axiomatization from the artifact generated a new type class for every new MRBNF. Not only that, it also defined a type class for the successor cardinal, in case that the MRBNF is later used in the fixpoint construction to build a codatatype.

Given that during the definition of a syntax, the composition algorithm from section 3.3.2 defines multiple MRBNFs, this started to become a performance issue. It also just polluted the global namespace with a lot of different – but isomorphic – type classes.

Aside from that, in a context where we did not have direct access to the MRBNF structure and thus the name of its type class, it was difficult to access theorems needed to prove boundedness of variable sets (see theorems 5.1 and 5.2). For example during the strengthening of inductive predicates described in chapter 4, this was an issue.

Theorem 5.1.

$$\frac{|A| <_o |\alpha| \qquad |B| <_o |\alpha|}{|A \cup B| <_o |\alpha|}$$

Theorem 5.2.

$$\frac{|A| <_o |\alpha| \qquad \forall x \in A.\, |B\ x| <_o |\alpha|}{\left|\bigcup_{x \in A} (B\ x)\right| <_o |\alpha|}$$

To combat this, we have added a registry for these type classes which is a mapping from cardinal to type class name. We manually define the type class for

countable types and prove that it is a subclass of Isabelle's infinite class. We also add theorem 5.1 to this type class. Theorem 5.2 needs the regularity of the cardinal, so we add this theorem to the manually defined class for countable types.

Now, whenever we generate a new MRBNF, we check if for the given bound there exists a type class already. As most syntaxes are finite, the registry will usually return the class for countable types. If a class does not exist, we generate it anew. Crucially, we also prove that the new class is a subclass of the countable type class (and thus a subclass of the *infinite* class). This way, the class directly inherits the two boundedness theorems.

Aside from fixing the performance issue due to the same class getting generated over and over, we can now also always use the theorems from the *infinite* class and the class for countable types as all variable type classes will always be a subclass of them. This means we do not need to know the exact variable class for a given type, fixing our other issue.

5.4.2 Using locales for the recursor

Most of the structures the automation defines take the form of an *axiomatization*. This means it is a function which takes a struct of constants and tactics, then uses those tactics to prove some axioms involving the constants and finally produces some constants and theorems as output.

This approach is very flexible, in our case for example in the number of positions on the MRBNF. Additionally, axiomatizations can make use of polymorphic constants, usually to exploit parametricity. For example, the creation of substitutions from section 3.4.3 relies on the parametricity of the pre-injections.

However, it also means that Standard ML is needed to construct these structures. It is possible to implement new Isabelle commands that wrap these internal functions – the `mrbnf` command is a good example of this – but this requires writing a parser, input validation and more.

Originally, we implemented the recursor in a similar way. For every function that the user wants to define, they would have to call a Standard ML function with the constants and tactics and they would get the recursive function and its theorems as result. Aside from forcing the user to learn Isabelle/ML, this means that for every function, all the theorems needed for its construction are proved again. Even

with just renaming and substitution defined by default, this was having a noticeable performance impact.

As already seen in section 3.3.4, at some point we realized that we do not actually need the additional flexibility that a full axiomatization would bring and that Isabelle's locales would fix the issues with our recursor implementation.

Locales are a common feature in Isabelle, so the user will already know how to use them. Additionally, we only have to do the proofs once abstractly during the definition of the locale. Instantiating the locale does not run the proofs again, fixing the performance issues of the axiomatized version.

5.5 Statistics

At the time of writing, the implementation consists of 30,500 lines of Standard ML across 23 ML modules. Additionally it uses 1000 lines of supporting lemmas written in normal Isabelle. The example proofs from chapter 5.2 amount to another 40,000 lines of Isabelle. Finally, the case studies from chapter 6 combined are around 15,000 lines of Isabelle.

During the development of the package, every syntax we encountered that was throwing an error was recorded to form a library of regression tests. On top of that, a few tests were added that focus on edge cases in the system.

The implementation is publicly available on GitHub[1]. The complexity of the project required adding some guard rails in the form of continuous integration. Before any change is added to the master branch of the repository, Isabelle's batch mode is used to run every test and case study to ensure the change does not break existing code.

[1] https://github.com/jvanbruegge/binder_datatypes/tree/v0.1

Chapter 6

Case Studies

Contents

6.1 Type safety of the simply typed λ-calculus 91

6.2 The POPLmark challenge 93

6.2.1 Transitivity of subtyping 93

6.2.2 Type safety of System $F_{<:}$ 96

6.3 Mazza's infinite affine λ-calculus 100

During the development of the framework, we validated our work in several case studies. Three of them constitute full formalizations, i.e., after defining the syntax and relevant relations like reduction, our framework is used to prove theorems about the system. Several other case studies are "definition-only" and were used to inform design decisions or generalizations. In the following chapters we will only talk about the full formalizations. We will also omit most proofs and refer instead to the Isabelle formalizations[1].

6.1 Type safety of the simply typed λ-calculus

The simply typed λ-calculus was developed by Church [16] as a restricted form of the untyped λ-calculus. In our case study, we want to prove type safety of this calculus via the usual progress and preservation theorems popularized by Wright and Felleisen [66].

As the types of the simply typed λ-calculus do not bind any variables, we can use a normal datatype to represent them:

```
datatype τ = Base | Arrow τ τ (infixr "→" 40)
```

The terms mirror those of the untyped λ-calculus, except that the variable binder in a λ-abstraction has to be annotated with a type:

[1] https://github.com/jvanbruegge/binder_datatypes/tree/v0.1/case_studies

```
binder_datatype (FVars: 'a) "term" =
  Var 'a
| Abs x::'a τ t::"'a term" binds x in t
| App "'a term" "'a term"
for subst: tvsubst
```

This definition will already provide a function to extract the free variables from a term (here called `FVars`) and a parallel substitution function `tvsubst`. The typing relation of the simply typed λ-calculus can be represented as an inductive relation:

```
abbreviation "dom Γ ≡ fst ‘ fset Γ"
inductive Ty :: "('a::var * τ) fset ⇒ 'a term ⇒ τ ⇒ bool"
  ("_ ⊢ _ : _" [25, 25, 25] 26) where
  Ty_Var: "(x, τ) |∈| Γ ⟹ Γ ⊢ Var x : τ"
| Ty_App: "Γ ⊢ e1 : τ1 → τ2 ⟹ Γ ⊢ e2 : τ1 ⟹ Γ ⊢ App e1 e2 : τ2"
| Ty_Abs: "x ∉ dom Γ ⟹ Γ,x:τ ⊢ e : τ2 ⟹ Γ ⊢ Abs x τ e : τ → τ2"
```

Using the strengthening described in chapter 4 we obtain a strong induction theorem that will be used to prove the main theorems later. For the operational semantics we have chosen a call-by-name evaluation that only ever evalutates the left redex:

```
inductive Step :: "'a::var term ⇒ 'a term ⇒ bool" (infixr "⟶β" 25) where
  ST_Beta: "App (Abs x τ e) e2 ⟶β tvsubst (Var(x:=e2)) e"
| ST_App: "e1 ⟶β e1' ⟹ App e1 e2 ⟶β App e1' e2"
```

With this setup done, we can prove the main helper lemmas for our case study. For both of these lemmas we use our strong induction to make the binders avoid the variables that are already in the context.

```
lemma context_invariance:
assumes "Γ ⊢ e : τ" "∀x∈FVars e. ∀τ. (x, τ) |∈| Γ ⟶ (x, τ) |∈| Δ"
shows "Δ ⊢ e : τ"

lemma substitution:
assumes "Γ,x:τ1 ⊢ e : τ" "x ∉ dom Γ" "{||} ⊢ v : τ1"
shows "Γ ⊢ tvsubst (Var(x:=v)) e : τ"
```

Finally, proving progress and preservation is straightforward, we do not even need strong induction:

```
theorem progress: "{||} ⊢ e : τ ⟹ (∃x τ t. e = Abs x τ t) ∨ (∃e'. e ⟶β e')"

theorem preservation: "{||} ⊢ e : τ ⟹ e ⟶β e' ⟹ {||} ⊢ e' : τ"
```

6.2 The POPLmark challenge

The POPLmark challenge [3] was designed as a benchmark for the usage of theorem provers in programming language theory. It consists of three parts, where the first two are concerned with the meta-theory of System $F_{<:}$, a polymorphic λ-calculus with subtyping. The third part is about extracting executable code from the formalization which is out of scope for our framework.

Part one and two are each split in two subtasks, where part A only deals with pure System $F_{<:}$ and part B extends the system with records. In the following sections we will directly discuss the full syntax, i.e., including part B.

6.2.1 Transitivity of subtyping

The first problem of the POPLmark challenge only deals with the type syntax of System $F_{<:}$ (see definition 6.1). Variables are from a countably infinite set of type variables, written in uppercase. In the forall binder, the variable X is bound in the body (the second type) and not in the supertype (the first type). The labels in a record type have to be pairwise distinct and their ordering is irrelevant.

Definition 6.1. *Types of System* $F_{<:}$

$$\begin{array}{rcll} T & ::= & X & \textit{type variables} \\ & | & \textit{Top} & \textit{maximum type} \\ & | & T \to T & \textit{type of functions} \\ & | & \forall X <: T.\, T & \textit{universal type} \\ & | & \{l_i : T_i{}^{i \in 1 \ldots n}\} & \textit{type of records} \end{array}$$

To represent records we choose a subtype of finite sets of pairs that ensures the first components of the pairs are pairwise distinct. We call this type *labeled fset*:

```
typedef ('k, 'v) lfset = "{ A::('k × 'v) fset
  | ∀x∈A. ∀y∈A. x ≠ y ⟶ fst x ≠ fst y }"
```

We can prove the MRBNF axioms for this type if we choose the *bound* usage for the keys. This will make our type require bijections for the keys which always preserve the uniqueness of labels. The types of System $F_{<:}$ can then be defined as a normal binder datatype:

```
binder_datatype (FVars_typ: 'a) "typ" =
```

```
  TyVar 'a
| Top
| Fun "'a typ" "'a typ"
| Forall X::'a "'a typ" T::"'a typ" binds X in T
| TRec "(string, 'a typ) lfset"
```

Next, the POPLmark challenge defines the subtyping relation (see definition 6.2). It uses a context that is either an empty context or the extension of a context with a variable and type pair. The variable may not be part of the rest of the context while all of the free variables of the type have to be part of the context. We capture these conditions in a well-formedness predicate:

```
type_synonym 'a Γτ = "('a × 'a typ) list"
abbreviation "dom Γ ≡ fst ‘ set Γ"
inductive wf_ty :: "'a::var Γτ ⇒ bool" ("⊢ _ ok" [70] 100) where
  wf_Nil: "⊢ [] ok"
| wf_Cons: "x ∉ dom Γ ⟹ FVars_typ T ⊆ dom Γ ⟹ ⊢ Γ ok ⟹ ⊢ Γ,x<:T ok"
```

Definition 6.2. *Subtyping in System $F_{<:}$*

$$\frac{\vdash \Gamma ok}{\Gamma \vdash T <: Top}\ SA\text{-}TOP \qquad \frac{\vdash \Gamma ok \qquad X \in dom\,\Gamma}{\Gamma \vdash X <: X}\ SA\text{-}REFL\text{-}TVAR$$

$$\frac{X <: U \in \Gamma \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}\ SA\text{-}TRANS\text{-}TVAR$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2}\ SA\text{-}ARROW$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: S_1.\, S_2 <: \forall X <: T_1.\, T_2}\ SA\text{-}ALL$$

This inductive relation can directly be expressed as an Isabelle inductive. Using the strong rule induction automation described in chapter 4 we also obtain a strong induction theorem for the subtyping relation:

```
inductive ty :: "'a::var Γτ ⇒ 'a typ ⇒ 'a typ ⇒ bool"
  ("_ ⊢ _ <: _" [55,55,55] 60) where
  SA_Top: "⊢ Γ ok ; FVars_typ S ⊆ dom Γ ⟹ Γ ⊢ S <: Top"
```

```
| SA_Refl_TVar: "⊢ Γ ok ; x ∈ dom Γ ⟹ Γ ⊢ TyVar x <: TyVar x"
| SA_Trans_TVar: "x<:U ∈ Γ ⟹ Γ ⊢ U <: T ⟹ Γ ⊢ TyVar x <: T"
| SA_Arrow: "Γ ⊢ T₁ <: S₁ ⟹ Γ ⊢ S₂ <: T₂ ⟹ Γ ⊢ Fun S₁ S₂ <: Fun T₁ T₂"
| SA_All: "Γ ⊢ T₁ <: S₁ ⟹ Γ, x<:T₁ ⊢ S₂ <: T₂
    ⟹ Γ ⊢ Forall x S₁ S₂ <: Forall x T₁ T₂"
| SA_TRec: "⊢ Γ ok ⟹ labels Y ⊆ labels X
    ⟹ (∀x T. (x, T) ∈ X ⟶ FVars_typ T ⊆ dom Γ)
    ⟹ (∀x T. (x, T) ∈ Y ⟶ ∃S. (x, S) ∈ X ∧ Γ ⊢ S <: T)
    ⟹ Γ ⊢ TRec X <: TRec Y"
```

The goal of part one of the POPLmark challenge is to prove transitivity of subtyping. To do this we can directly follow the proof sketch given by the challenge, as the framework already takes care of observing the variable convention. For example the first lemma is reflexivity of subtyping, which the definition only directly has for type variables:

```
lemma ty_refl: "⊢ Γ ok ⟹ FVars_typ T ⊆ dom Γ ⟹ Γ ⊢ T <: T"
proof (binder_induction T arbitrary: Γ avoiding: "dom Γ" rule: typ.strong_induct)
  case (TyVar x Γ)
  then show ?case by blast
next
  case (TRec x Γ)
  then show ?case using SA_TRec by (force intro: lfin_values)
qed (auto simp: Diff_single_insert SA_All wf_Cons)
```

Here we use our custom binder_induction tactic that allows the user to specify which variables the binders should avoid. Here, this is necessary because in the forall case, we have to go under the binder and add the variable to the context. This is only allowed if the variable is not already in the context. By ensuring that the binder is fresh with regard to the context we can side step this issue. Thus the proof is easily completed by Isabelle's automation.

After additionally proving context permutation and context weakening, we can prove transitivity and narrowing. The POPLmark challenge suggest proving both at the same time via mutual induction on one of the terms. While this works and we formalized this version also in Isabelle, a simpler proof can be obtained by first proving narrowing under the assumption of transitivity and then proving transitivity while instantiating this assumption with the induction hypothesis:

```
lemma ty_narrowing_aux:
  assumes ty_trans: "∀Γ S T. Γ ⊢ S <: Q ⟶ Γ ⊢ Q <: T ⟶ Γ ⊢ S <: T"
```

```
  and "(Γ, X <: Q), Δ ⊢ M <: N" "Γ ⊢ R <: Q"
shows "(Γ, X <: R), Δ ⊢ M <: N"

theorem ty_transitivity: "Γ ⊢ S <: Q ⟹ Γ ⊢ Q <: T ⟹ Γ ⊢ S <: T"
```

This has the advantage that for the narrowing proof, we can use induction on the typing derivation and not one of the terms which makes the proof a lot simpler. If narrowing is wanted as standalone lemma, it can be derived directly from the auxiliary lemma and the transitivity theorem.

6.2.2 Type safety of System $F_{<:}$

Part two of the POPLmark challenge adds the term syntax of System $F_{<:}$ (see definition 6.3). The goal is to prove type safety via preservation (evaluating a well-typed term is still well-typed) and progress (if a well-typed term is closed, it is either a value or it can be evaluated further).

Definition 6.3. *Term syntax of System* $F_{<:}$

$$
\begin{array}{rcll}
t & ::= & x & \textit{variable} \\
 & | & \lambda x : T.\, t & \textit{abstraction} \\
 & | & t\, t & \textit{application} \\
 & | & \lambda X <: T.\, t & \textit{type abstraction} \\
 & | & t[T] & \textit{type application} \\
 & | & \{l_i : t_i^{\,i \in 1 \ldots n}\} & \textit{record} \\
 & | & t.l & \textit{projection} \\
 & | & \textit{let}\, p = t\, \textit{in}\, t & \textit{pattern binding} \\
 & & & \\
p & ::= & x : T & \textit{variable pattern} \\
 & | & \{l_i = p_i^{\,i \in 1 \ldots n}\} & \textit{record pattern}
\end{array}
$$

While we can directly represent the term syntax itself, some extra care is needed for the patterns. As hinted at in chapter 2.3, we want to ensure that all of the pattern variables in a pattern are unique. For our formalization we have manually defined a suitable subtype and proved the MRSBNF axioms for it. Since then, Krayer has developed a command that can automate this linearization step [31]. The term syntax is very straightforward then:

```
binder_datatype (FTVars: 'tv, FVars: 'v) trm =
    Var 'v
  | Abs x::'v "'tv typ" t::"('tv, 'v) trm" binds x in t
  | App "('tv, 'v) trm" "('tv, 'v) trm"
  | TAbs X::'tv "'tv typ" t::"('tv, 'v) trm" binds X in t
  | TApp "('tv, 'v) trm" "'tv typ"
  | Rec "(label, ('tv, 'v) trm) lfset"
  | Proj "('tv, 'v) trm" label
  | Let "('tv, p::'v) pat" "('tv, 'v) trm" t::"('tv, 'v) trm" binds p in t
```

Importantly, this definition automatically provides a (parallel) substitution function that can not only substitute terms for term variables, but also types for type variables in terms. In fact, this case study and other variations of System F sparked our development of MN-Monads (see chapter 3.4). Here, we use them to automatically lift the substitution on types to one on terms while also adding a new substitution for term variables in the process.

Similar to the context for the subtyping relation, the typing context has some well-formedness requirements that we capture in an inductive predicate. Again, the variable may not already exist in the context while all the free variables of the associated type have to be part of the context already. Instead of creating a bespoke type for the elements of the context, we use a sum type for the first component of the pair:

```
type_synonym ('tv, 'v) Γt = "(('tv + 'v) × 'tv typ) list"

inductive wf_ctxt :: "('tv::var, 'v::var) Γt ⇒ bool" ("⊢ _ OK" [70] 100) where
  wf_ctxt_Nil: "⊢ [] OK"
| wf_ctxt_Cons: "x ∉ dom Γ ⟹ FVars_typ T ⊆ { a | Inl a ∈ dom Γ }
    ⟹ ⊢ Γ OK ⟹ ⊢ Γ,x<:T OK"
```

The typing judgement of System $F_{<:}$ (see definition 6.5) requires the typing judgement of record patterns (see definition 6.4). This pattern judgements builds a context that contains all the pattern variables with their types to be used in the let rule of the typing judgement.

Definition 6.4. *Pattern typing in System $F_{<:}$*

$$\frac{}{\vdash (x : T) : T \Rightarrow x : T}\ \textit{P-Var}$$

$$\frac{\forall i \in \{1 \ldots n\}.\ \vdash p_i : T_1 \Rightarrow \Delta_i}{\vdash \{l_i = p_i^{i \in 1 \ldots n}\} : \{l_i : T_i^{i \in 1 \ldots n}\} \Rightarrow \Delta_n, \ldots, \Delta_1}\ \textit{P-Rcd}$$

Definition 6.5. *Typing in System $F_{<:}$*

$$\frac{x : T \in \Gamma \qquad \vdash \Gamma\ \mathrm{OK}}{\Gamma \vdash x : T}\ \textit{T-Var} \qquad \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.\, t_2 : T_1 \to T_2}\ \textit{T-Abs}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\, t_2 : T_2}\ \textit{T-App} \qquad \frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1.\, t_2 : \forall X <: T_1.\, T_2}\ \textit{T-TAbs}$$

$$\frac{\Gamma \vdash t_1 : \forall X <: T_1.\, T_2 \qquad \Gamma \vdash T <: T_1}{\Gamma \vdash t\,[T] : T_2[X := T]}\ \textit{T-TApp} \qquad \frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash t : T}\ \textit{T-Sub}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \vdash p : T_1 \Rightarrow \Delta \qquad \Gamma, \Delta \vdash t_2 : T_2}{\Gamma \vdash \textbf{\textit{let}}\, p = t_1\, \textbf{\textit{in}}\, t_2 : T_2}\ \textit{T-Let}$$

$$\frac{\forall i \in \{1 \ldots n\}.\, \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i^{i \in 1 \ldots n}\} : \{l_i : T_i^{i \in 1 \ldots n}\}}\ \textit{T-Rcd} \qquad \frac{\Gamma \vdash t_1 : \{l_i : T_i^{i \in 1 \ldots n}\}}{\Gamma \vdash t_1.l_j : T_j}\ \textit{T-Proj}$$

Similar to the subtyping relation, both of these judgements are represented using an inductive relation:

```
inductive pat_typ :: "('tv::var, 't::var) pat ⇒ 'tv typ ⇒ ('tv, 't) Γ_t ⇒ bool"
  ("⊢ _ : _ → _" [30,29,30] 30) where
  PVar: "⊢ PVar x T : T → [], Inr x <: T"
| PRec: "nonrep_PRec PP ⟹ labels PP = labels TT
    ⟹ (∀l P T. (l, P) ∈ PP ⟹ (l, T) ∈ TT ⟹ ⊢ P : T → Δ l)
    ⟹ xs = labelist TT ⟹ ⊢ PRec PP : TRec TT → List.concat (map Δ xs)"

inductive typing :: "('tv::var, 't::var) Γ_t ⇒ ('tv, 't) trm ⇒ 'tv typ ⇒ bool"
  ("_⊢ _: _" [30,29,30] 30) where
  TVar: "⊢ Γ OK ⟹ (Inr x, T) ∈ set Γ ⟹ Γ ⊢ Var x : T"
| TAbs: "Γ, Inr x <: T1 ⊢ t: T2 ⟹ Γ ⊢ Abs x T1 t : T1 → T2"
```

```
| TApp: "Γ ⊢ t1: T11 → T12 ⟹ Γ ⊢ t2: T11 ⟹ Γ ⊢ App t1 t2 : T12"
| TTAbs: "Γ, Inl X <: T1 ⊢ t: T2 ⟹ Γ ⊢ TAbs X T1 t: ∀X <: T1. T2"
| TTApp: "Γ ⊢ t1: ∀X <: T11. T12 ⟹ proj_ctxt Γ ⊢ T2 <: T11
    ⟹ Γ ⊢ TApp t1 T2 : tvsubst_typ (TyVar(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) X Y ⟹ Γ ⊢ Rec X : TRec Y"
| 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"
```

A few of these rules deserve some explanation. Rules `TVar` and `TRec` assume that the context is well-scoped $\vdash \Gamma$ OK; other rules preserve this invariant inductively. Rule `TRec` uses the relator `rel_lfset` to relate the values in two labeled finite sets pairwise grouped by label. Rule `TTApp` uses the parallel substitution function on System $F_{<:}$ types, which is automatically defined by the framework. Rule `TSub` uses `proj_ctxt` to extract all the type variable bindings from the context to obtain a context suitable for the subtying relation of part one. Rule `PRec` assumes that the destructed record pattern is non-repetitive (`nonrep_PRec`); it also constructs the resulting context by sorting the finite set of labels lexicographically (`labelist`).

We next define matching and the evaluation function for the terms. Similar to Berghofer's solution [8], we use a matching predicate rather than a (partial) function that computes the matching substitution:

```
inductive match for σ where
  MPVar: "σ X = v ⟹ match σ (PVar 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 v x = (if x ∈ A then σ x else v x)"

inductive "value" where
  "value (Abs x T t)"
| "value (TAbs X T t)"
| "(∀v ∈ values XX. value v) ⟹ value (Rec XX)"

inductive step where
  AppAbs: "value v ⟹ step (App (Abs x T t) v) (tvsubst (Var(x := v)) TyVar t)"
| TAppTAbs: "step (TApp (TAbs X T t) T2) (tvsubst Var (TyVar(X := T2)) t)"
| LetV: "value v ⟹ match σ p v
```

```
    ⟹ step (Let p v u) (tvsubst (restrict σ (PVars p) Var) TyVar u)"
| ProjRec: "(∀v ∈ values V. value v) ⟹ (l, v) ∈ V ⟹ step (Proj (Rec VV) l) v"
| AppCong1: "step t t' ⟹ step (App t u) (App t' u)"
| AppCong2: "value v ⟹ step t t' ⟹ step (App v t) (App v t')"
| TAppCong: "step t t' ⟹ step (TApp t T) (TApp 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)"
```

The rules `AppAbs`, `TAppTAbs`, `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 `AppAbs`, `TAppTAbs`, and `LetV` use the parallel substitution `tvsubst`, which we again obtain automatically from the definition of the term syntax. 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 [3]. We pervasively use binding-aware induction on our datatypes but also on the shown inductive predicates, with the strengthening from chapter 4. 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
```

6.3 Mazza's infinite affine λ-calculus

In his work on connecting the meta-theory of the untyped λ-calculus with the notion of metric completion, Mazza [36] establishes an isomorphic translation between the standard untyped λ-calculus (using the syntax from definition 2.1) and

a (uniform affine) infinitary λ-calculus (see definition 2.12). Both of these syntaxes can be defined directly using our framework, using infinite streams or distinct infinite streams for the latter syntax:

```
binder_datatype (FVars: 'a) term =
  Var 'a
| App "'a term" "'a term"
| Abs x::'a t::"'a term" binds x in t

binder_datatype (iFVars: 'a) iterm =
  iVar 'a
| iApp "'a iterm" "'a iterm stream"
| iAbs "(xs::'a) dstream" t::"'a iterm" binds xs in t
```

Recall that the type used as variables in the untyped λ-calculus must be infinite (since the cardinal bound of the type representing its syntax is $\aleph_0$), and the one for the infinitary λ-calculus must be uncountably infinite. For the rest of this chapter, we will fix the types used as variables, namely a countable type *var* (a copy of *nat*) and an uncountable type *ivar*. We will refer to the elements of *ivar* as *ivariables* and we will write *term* instead of *var term* and *iterm* instead of *ivar iterm*.

Following Mazza, we choose a countable set Spr : (var dstream) set of distinct streams of variables called supervariables, having the property that any two are mutually disjoint: $\forall xs, ys \in Spr.\ \text{sset}\ xs \cap \text{sset}\ ys = \emptyset$. The intention is restricting the λ-iterms to only use these as bindings. Moreover, we choose a function spr : var $\to$ (var dstream) set which is a bijection between variables and supervariables. We refer to the elements of *nat list* as positions, and choose a bijection $natOf$: nat list $\to$ nat.

For p : nat list and n : 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 $[\![\_]\!]\_$: term $\to$ nat list $\to$ iterm given by:

(1) $[\![\text{Var}\ x]\!]_p = \text{iVar}\ ((\text{spr}\ x)_{\text{natOf}\ p})$

(2) $[\![\text{Abs}\ x\ t]\!]_p = \text{iAbs}\ (\text{spr}\ x)\ [\![t]\!]_p$

(3) $[\![\text{App}\ t_1\ t_2]\!]_p = \text{iApp}\ [\![t_1]\!]_{p\cdot 0}\ ([\![t_2]\!]_{p\cdot 1},\ [\![t_2]\!]_{p\cdot 2},\ [\![t_2]\!]_{p\cdot 3}, \ldots)$

The intuition is that every variable x in the original term is duplicated in the translation into countably many ivariable "copies" sourced from its corresponding

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

Definition 6.6. *Renaming equivalence*

$$\frac{xs \in Spr \qquad \{x, x'\} \subseteq sset\, xs}{iVar\, x \approx iVar\, x'}\ \textit{iVar} \qquad \frac{xs \in Spr \qquad t \approx t'}{iLam\, xs\, t \approx iLam\, xs\, t'}\ \textit{iLam}$$

$$\frac{t \approx t' \qquad \forall t_1\, t_2.\ \{t_1, t_2\} \subseteq sset\, ts \cup sset\, ts' \longrightarrow t_1 \approx t_2}{iApp\, t\, ts \approx iApp\, t'\, ts'}\ \textit{iApp}$$

Moreover, to describe the image of this translation, Mazza defines the notion of renaming equivalence expressed as the relation $\approx$: iterm $\to$ iterm $\to$ bool (see definition 6.6). It relates two λ-iterms t and t' just in case they:

(i) have the same (iVar, iLam, iApp)-structure (as trees)

(ii) only use supervariables in binders

(iii) at the leaves have variables appearing in the same supervariable

(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: $uniform\, t = (t \approx t)$.

We aim to define a function satisfying clauses (1)–(3) above, with (3) formally written as iApp $[\![t_1]\!]_{p\cdot 0}$ (smap $(\lambda i.\, [\![t_2]\!]_{p\cdot i})$ (natsFrom 1)) where, for any n, natsFrom n denotes the stream of naturals starting from n. To turn these clauses into a formal definition, we use the recursor described in chapter 3.3.4 on terms. The recursor also requires indicating the desired interaction between the to-be-defined function with

permutation and free variables. More concretely, we need to specify a permutation-like operator (called *Umap*) on the future codomain that commutes with permutation on terms as well as a free-variable-like operator (called *UFVars*) that may not introduce new free variables.

As *Umap* has to work on ivariables while mirroring the permutation action on terms, we have to define a mapping between variables and ivariables that we can use with the permutation action on iterms: $\mathit{Umap}\,\sigma\,t := t[\mathrm{v2iv}\,\sigma]$ where we define $\mathrm{v2iv}\,\sigma\,y := \textbf{let}\ (i, xs) = \epsilon_{(i,xs)}\ (y = xs_i)\ \textbf{in}\ (\mathrm{spr}\ (\sigma\ (\mathrm{spr}^{-1}\ xs)))_i$. Because the ivariables that form supervariables are all distinct, any ivariable is part of exactly one supervariable. So we use Hilbert's choice operator to obtain this one supervariable and the index at which the ivariable y can be found in it. We then use the *spr* bijection between variables and supervariable to map the permutation σ to a permutation on supervariables to obtain the final permuation.

For the free-variable operator *UFVars* we choose the set of variables corresponding to the set of "touched" supervariables: $\mathit{UFVars}\,t = \{\mathrm{spr}^{-1}\ xs \mid xs \in \mathrm{touched}\,t\}$ where $\mathit{touched}\,t$ is the set of supervariables that contain the free ivariables of the term: $\{xs \in \mathrm{Spr} \mid \mathrm{sset}\ xs \cap \mathrm{iFVars}\,t \neq \emptyset\}$.

Recall that the recursor requires the permutation operator to commute with permutation on terms: (4) $[\![t[\sigma]]\!]_p = [\![t]\!]_p[\mathrm{v2iv}\ \sigma]$. Intuitively, this axiom holds as the *v2iv* translation sends variables to supervariables, which means that bijections σ between variables naturally correspond to bijections between supervariables (via the bijection *spr*). Hence (thanks to the supervariables being mutually disjoint) they correspond to supervariable-structure preserving bijections between ivariables. Therefore applying a bijection on variables and then translating should be the same as first translating and then applying the corresponding bijection.

For the free-variable operator, the requirement is that the returned set is included in the free variables of the original term:

(5) $\{\mathrm{spr}^{-1}\ xs \mid xs \in \mathrm{touched}\ [\![t]\!]_p\} \subseteq \mathrm{FVars}\ t$.

The intuition behind this axiom stems again from the direct variable-to-supervariable correspondence witnessed by *spr*.

Clauses (1)–(5) give us a structure on the intended codomain of $[\![\_]\!]\_$, nat list $\to$ *iterm*. However, for these to give us a model of the codomain as demanded by the recursor, we must restrict the codomain to only include those functions f :

nat list → item whose image consists of renaming-equivalent items only – otherwise the model properties do not hold. This is not surprising, since the goal of Mazza's translation is to produce uniform iterms. Our recursor allows such narrowing by specifying a predicate on the codomain: $validU\ f := \forall p\ q.\ f\ p \approx f\ q$. With this we obtain our final translation function:

Theorem 6.1. *There exists a unique function* $[\![\_]\!]\_ : term \to nat\ list \to iterm$ *such that clauses (1)-(5) hold, and in addition* $\forall p\ q.\ [\![t]\!]_p \approx [\![t]\!]_q$*; in particular* $\forall p.\ uniform\ [\![t]\!]_p$.

For the opposite translation $(\!|\_|\!)$ (from infinitary back to finitary terms), Mazza writes equations that in our notation look as follows, restricting the domain to uniform iterms:

(1) $(\!|\mathrm{iVar}\ xs_i|\!) = \mathrm{Var}\ (\mathrm{spr}^{-1}\ xs)$

(2) $(\!|\mathrm{iAbs}\ xs\ t|\!) = \mathrm{Abs}\ (\mathrm{spr}^{-1}\ xs)\ (\!|t|\!)$

(3) $(\!|\mathrm{iApp}\ t\ ts|\!) = \mathrm{App}\ (\!|t|\!)\ (\!|ts_0|\!)$

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

Theorem 6.2. *There exists a unique function* $(\!|\_|\!) : iterm \to term$ *satisfying the clauses (1)-(3) above when restricted to uniform iterms.*

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

Theorem 6.3. *The following hold:*

(1) $t \approx t' \longrightarrow (\!|t|\!) = (\!|t'|\!)$ *(Lemma 16 from [36])*

(2) $(\!|[\![s]\!]_p|\!) = s$ *(Thm. 19(1) from [36])*

(3) $uniform\ t \longrightarrow [\![(\!|t|\!)]\!]_p \approx t$ *(Thm. 19(2) from [36])*

The theorem states that, for any position p, $[\![\_]\!]_p$ and $(\!|\_|\!)$ give mutually inverse bijections between terms and equivalence classes of uniform iterms with regard to renaming equivalence. An additional lemma (omitted here) shows that the $\approx$-representative produced by $[\![\_]\!]_p$ 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 $[\![\_]\!]_p$ and $(\!|\_|\!)$ preserve β-reduction in both calculi in a manner that matches the number of reduction steps (Thm. 19(3,4) in [36]). We omit this result here, but details can be found in our formalization.

Chapter 7

Related Work

Contents

7.1 Nameless representations 106

7.2 Higher-order abstract syntax 107

7.3 Nameful representations 107

7.4 Other comparisons between paradigms in the literature 108

7.5 Comparison to Nominal2 108

There are thee main paradigms in reasoning about syntax with binders in a formal setting: the nameless paradigm, the "nameful" paradigm, and higher-order abstract syntax (HOAS). Of course, there are also hybrids between these paradigms like "locally nameless" [37] where bound variables are nameless and free variables have names.

Given the syntax of the untyped λ-calculus, all of these paradigms differ in the type of the constructor of their λ-abstraction. For the rest of this chapter we will assume that the type representing said syntax is called *term* with *var* being the type uses as variables where appropriate.

7.1 Nameless representations

The representation of variables as de Bruijn indices [18] is widely popular in proof assistants [8, 65, 55, 26, 17, 53, 60, 43] because it is readily available via standard datatypes. Bound variables point to their 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, i.e. its constructor has type $\mathit{term} \to \mathit{term}$. Free variables are numbers, too, namely those larger than the number of binders above them.

For example, Lam (Lam (App (App 1 0) 2)) is the de Bruijn version of the 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 for example provided by the Autosubst tool in Coq [54, 57] – 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.

7.2 Higher-order abstract syntax

Higher-Order Abstract Syntax (HOAS) [48] uses binding primitives available in the meta-logic to represent binders of the object of study. This can be seen in the type of their λ-constructor that accepts a function in the meta-logic as argument. Under weak HOAS the constructor has type $(\mathit{var} \to \mathit{term}) \to \mathit{term}$, while it has type $(\mathit{term} \to \mathit{term}) \to \mathit{term}$ under HOAS. It thus reuses the λ-abstraction of the meta-logic when writing specific terms, e.g. $\mathsf{Lam}\ (\lambda x.\ \mathsf{Lam}\ (\lambda y.\ \mathsf{App}\ (\mathsf{App}\ y\ x)\ (\mathsf{Var}\ z)))$.

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 and was pioneered by Twelf [49] and refined and extended in Beluga [50] and Abella [4].

7.3 Nameful representations

Nominal Logic [22] provides a nameful alternative to the two above approaches: binders carry explicit bound variable names, but the syntax is quotiented by a notion of α-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 the variable convention popularized by Barendregt [5]. Nominal Isabelle is the Isabelle/HOL implementation of nominal logic [28, 64], which has been used in several substantial formalizations efforts [7, 46, 14, 11]. Our framework follows the nominal approach, while generalizing the support for nested recursion and allowing infinitely many variables in terms.

7.4 Other comparisons between paradigms in the literature

Blanchette et al. [9, Section 9] provide a broad overview of the different approaches of syntax with bindings in programming languages and proof assistants. Berghofer and Urban [8] provide a detailed comparison between the de Bruijn and nominal approaches; Momigliano et al. [42] perform a similar exercise for de Bruijn and (weak) HOAS. Ambal et al. [1] compare all above approaches in the context of a higher-order π-calculus. Norrish and Vestergaard [44] establish a formal connection between de Bruijn and nominal terms. Dowek and Gabbay [19] as well as Gabbay and Nanevski [23] relate nominal logic to HOAS, in particular the exotic terms mentioned in section 7.2.

Solutions using different representations [58] target the POPLmark challenge [3]. 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 [65, 56] and one using HOAS in Twelf [2]. We provide the first complete solution following the nominal approach in chapter 6.2.

7.5 Comparison to Nominal2

In the context of Isabelle/HOL, the Nominal2 tool [62] is the closest to our framework in terms of features. It is also built on nominal logic, but is restricted to finite syntax. A major limitation of Nominal2 is the missing support for nested recursion. For example, the λ-calculus with n-ary application from chapter 2.2 would have to be expressed as a mutually recursive datatype compared to recursing through the normal HOL list type:

```
atom_decl name
nominal_datatype "term" =
  Var name
| App "term" term_list
| Abs x::name t::"term" binds x in t
and term_list =
  TNil
| TCons "term" "term_list"
```

```
binder_datatype 'a "term" =
  Var 'a
| App "'a term" "'a term list"
| Abs x::'a t::"'a term" binds x in t
```

This of course means that the user cannot directly use the rich library of theorems

about lists that already exists in Isabelle, and instead has to either manually prove them again for this isomorphic type or use for example the transfer tool [27] to automate this process. Additionally it makes proofs more complicated as every statement now becomes a mutually inductive proof if the user does not manually construct a more convenient induction theorem.

Another major limitation is the support for non-free datatypes, e.g. in binders. While Nominal2 supports binding (finite) sets of variables, it is not possible to use a type like dlist (the type of lists of distinct elements). This makes it hard to enforce uniqueness of variables in nested binders (as required for part two of the POPLmark challenge, see chapter 6.2.2).

Aside from a free variable operator, Nominal2 does not provide other common operators like renaming or substitution. Given that especially the latter is required for most formal developments, every user has to define those operators themselves. This obviously includes all the theorems about substitution like composition and identity mentioned in chapter 2.1.

One advantage of Nominal2 is the support for high-level function definitions on nominal datatypes. Our framework currently only provides the low-level recursor in the form of an Isabelle locale that the user can instantiate to obtain their function. A good first step in the direction of Nominal2 could be to define a high-level locale that works on the user-facing constructors of the type, not the internal single constructor.

Chapter 8

Conclusion and Future Work

After over four years of work, our framework has crossed the line from "prototype" to "usable" while providing a combination of features that no other system can match. That said, there are still a lot of areas that can and should be improved. We are not yet at a point where formal reasoning is easier or even just as easy as informal pen and paper proofs.

From an engineering perspective, there are some issues with how binder datatypes and normal datatypes interact. For example, if the user defines a binder datatype and uses it in a datatype which in turn they want to use in a binder datatype, they will not be able to access the variables of the first binder datatype. The solution here is to port our generalization of BNFs to Isabelle itself. However, we know that our generalization that adds bound and free positions is not the only interesting one. Other reasearchers for example are exploring co- and contravariant positions for Isabelle's datatypes [33], so any future implementation work should take great care to ensure these independent generalizations work together.

On the automation side, we currently have an ad hoc tactic that can automatically prove the equivariance needed by the strengthened rule induction (see chapter 4). While this automation works for all our case studies, there will most likely be cases where the automation fails. In his paper "Equivariant ZFA and the foundations of nominal techniques" Gabbay [20] argues that to make large formalizations based on nominal techniques feasible, equivariance has to have a constant cost, i.e., it may not scale with the complexity of the formalization. He proposes to embrace nominal atoms together with their equivariance infrastructure as part of the base logic to achieve this. While our personal preference is to always stay within standard HOL, possibly requiring us to help the automation with manual proofs, nothing prevents the user from axiomatizing the required equivariance if that is their preference. Ideally, we could find a more principled approach to solving equivariance proofs that would allow us to replace the ad hoc automation with a tactic will not fail on more complicated definitions. That said, the current automation is

already useful for a broad range of definitions (see chapter 6).

There are also several improvements that could be made to the usability of the framework. Currently, we automatically provide parallel renaming and substitution. However, many calculi only deal with single variables and thus only need singular substitution or renaming. As the parallel variants come with extra smallness assumptions, the singular versions would be easier to use. For (co)recursive functions, we currently only provide the low-level locale to the user. To make it easier to define functions on binder datatypes, there should be a `binder_primrec` command similar to the one for datatypes. Finally, our framework does not provide strengthened inversion rules for inductive predicates. For example, given an inductive rule like the typing judgement of the λ-abstraction in the simply typed λ-calulus (see definition 8.1), the associated inversion rule only provides an α-equivalent term, not the same term (see theorem 8.1).

Definition 8.1.

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}\ \textit{TyAbs}$$

Theorem 8.1.

$$\frac{\Gamma \vdash \lambda x : \tau_1.\ e : \tau \qquad \forall y\ e'\ \tau_2.\ \lambda x : \tau_1.\ e = \lambda y : \tau_1.\ e' \longrightarrow \tau = \tau_1 \to \tau_2 \longrightarrow \Gamma, y : \tau_1 \vdash e' : \tau_2 \longrightarrow P}{P}$$

However, if we know that the variable x is fresh outside of the binder – so in Γ in this case – we can derive a better rule that uses the same binder in the inversion (see theorem 8.2). The required freshness is not an issue in practice as we can easily get it from the strong induction theorem (see theorem 3.14).

Theorem 8.2.

$$\frac{\Gamma \vdash \lambda x : \tau_1.\ e : \tau \qquad x \notin \mathit{dom}\,\Gamma \qquad \forall \tau_2.\ \tau = \tau_1 \to \tau_2 \longrightarrow \Gamma, x : \tau_1 \vdash e : \tau_2 \longrightarrow P}{P}$$

Despite the areas that can still be improved, we would describe our framework as ready for experimentation. In fact, we already have a few early users that tried their hand at using the framework for their formalizations. The feedback from

these users is critical to further improve the framework in the long run, be it for improvements in usability or simply bug reports for tactics that fail in specific edge cases. Once the framework has gone through this stabilization process, we plan to integrate it with the Isabelle distribution. For the users that want to help us with this stabilization, we encourage trying out the framework:

`https://github.com/jvanbruegge/binder_datatypes/tree/v0.1`

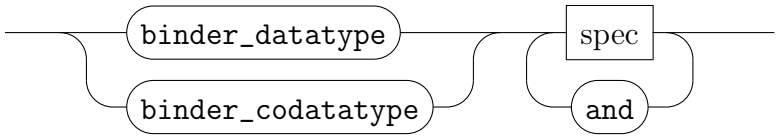
Appendix A

Command syntax

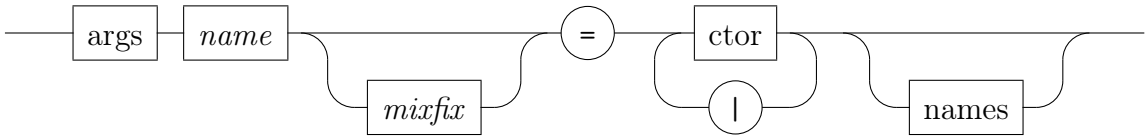
The full syntax of the commands introduced by the binders package are presented using syntax or railroad diagrams. Terminals are written in a `typewriter` font and use rounded corners. Nonterminals are written in *italics* if they refer to a builtin syntax category in Isabelle.

A.1 Binder (Co)Datatypes

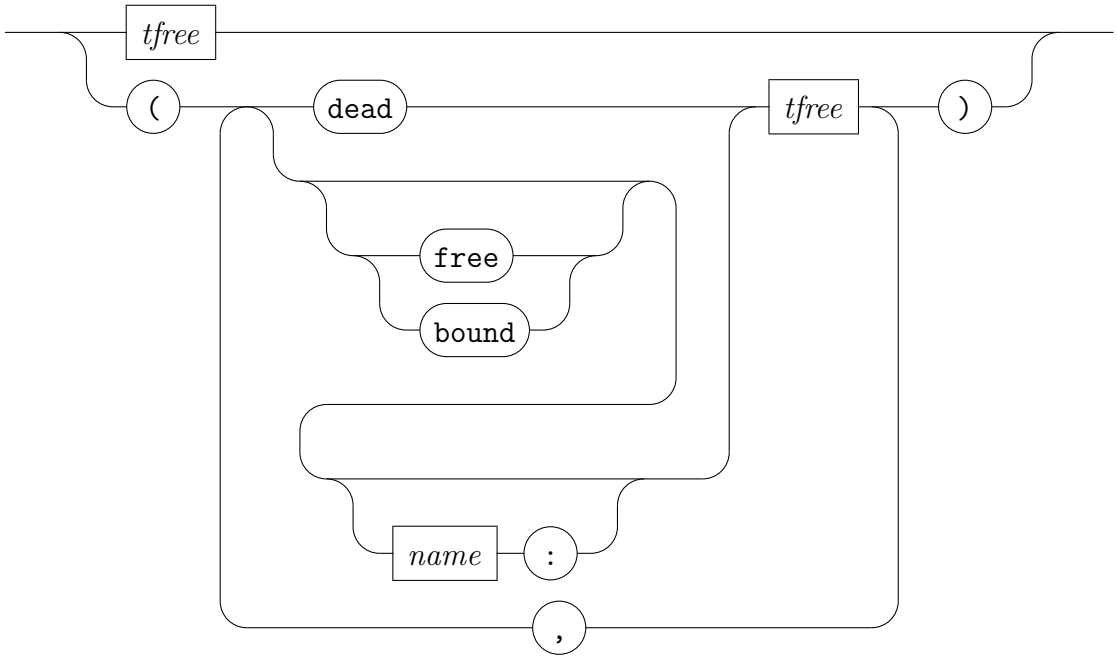
decl



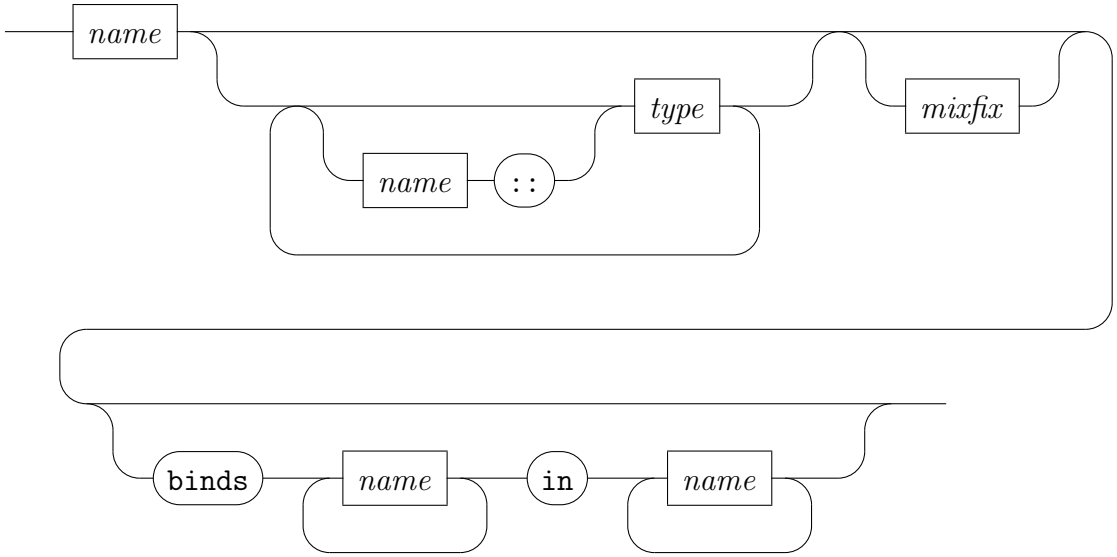
spec



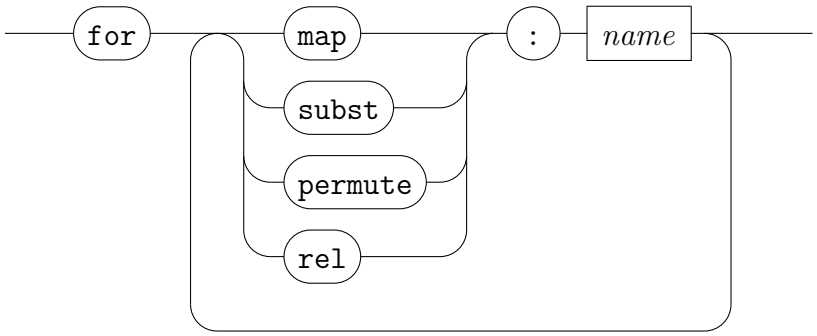
args



ctor

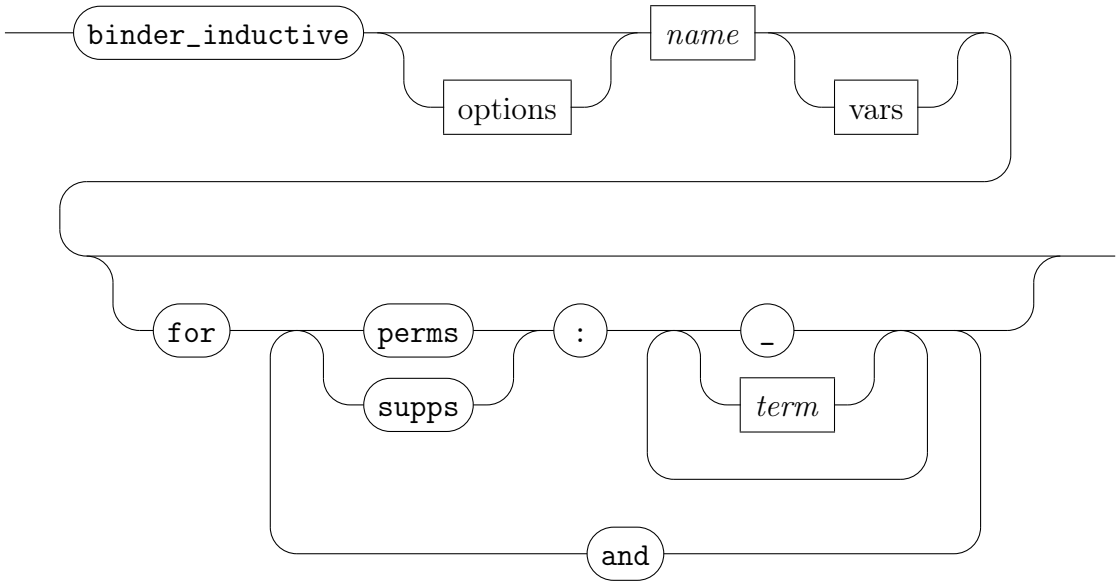


names

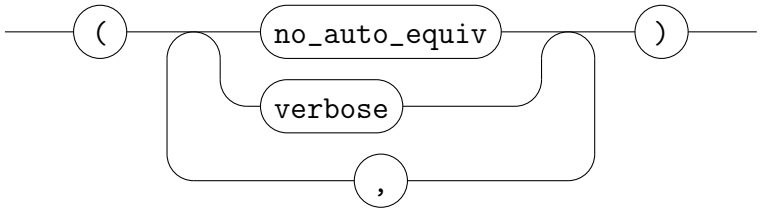


A.2 Binder Inductives

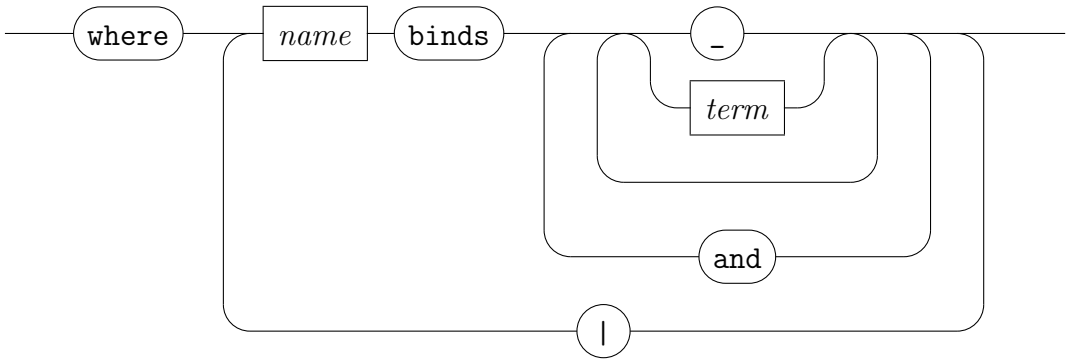
idecl



options



vars



Bibliography

[1] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt. "HOπ in Coq". In: *J. Autom. Reason.* 65.1 (2021), pp. 75–124. DOI: 10.1007/S10817-020-09553-0.

[2] 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.

[3] Brian E. Aydemir et al. "Mechanized Metatheory for the Masses: The POPLMark Challenge". In: *TPHOLs 2005*. Ed. by Joe Hurd and Thomas F. Melham. Vol. 3603. LNCS. Springer, 2005, pp. 50–65. DOI: 10.1007/11541868_4.

[4] David Baelde et al. "Abella: A System for Reasoning about Relational Specifications". In: *J. Formaliz. Reason.* 7.2 (2014), pp. 1–89. DOI: 10.6092/ISSN.1972-5787/4650.

[5] Hendrik Pieter Barendregt. *The lambda calculus – its syntax and semantics*. Vol. 103. Studies in logic and the foundations of mathematics. North-Holland, 1985. ISBN: 978-0-444-86748-3.

[6] Henk P. Barendregt. "The Lambda Calculus: Its Syntax and Semantics". In: *Studies in logic and the foundations of mathematics*. Vol. 40. 1984.

[7] Jesper Bengtson and Joachim Parrow. "Formalising the pi-calculus using nominal logic". In: *Log. Methods Comput. Sci.* 5.2 (2009).

[8] Stefan Berghofer. "A Solution to the POPLMark Challenge Using de Bruijn Indices in Isabelle/HOL". In: *J. Autom. Reason.* 49.3 (2012), pp. 303–326. DOI: 10.1007/S10817-011-9231-4.

[9] Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. "Bindings as bounded natural functors". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 22:1–22:34. DOI: 10.1145/3290335.

[10] Jasmin Christian Blanchette et al. "Truly Modular (Co)datatypes for Isabelle/HOL". In: *ITP 2014*. Ed. by Gerwin Klein and Ruben Gamboa. Vol. 8558. LNCS. Springer, 2014, pp. 93–110. DOI: 10.1007/978-3-319-08970-6_7.

[11] Joachim Breitner. "Formally proving a compiler transformation safe". In: *Haskell Symposium 2015*. Ed. by Ben Lippmeier. ACM, 2015, pp. 35–46. DOI: 10.1145/2804302.2804312.

[12] Jan van Brügge, James McKinna, Andrei Popescu, and Dmitriy Traytel. "Barendregt Convenes with Knaster and Tarski: Strong Rule Induction for Syntax with Bindings". In: *Proc. ACM Program. Lang.* 9.POPL (Jan. 2025). DOI: 10.1145/3704893.

[13] Jan van Brügge, Andrei Popescu, and Dmitriy Traytel. "Animating MRB-NFs: Truly Modular Binding-Aware Datatypes in Isabelle/HOL". In: *16th International Conference on Interactive Theorem Proving (ITP 2025)*. Ed. by Yannick Forster and Chantal Keller. Vol. 352. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 11:1–11:20. ISBN: 978-3-95977-396-6. DOI: 10.4230/LIPIcs.ITP.2025.11.

[14] Matthias Brun and Dmitriy Traytel. "Generic Authenticated Data Structures, Formally". In: *ITP 2019*. Ed. by John Harrison, John O'Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 10:1–10:18. DOI: 10.4230/LIPICS.ITP.2019.10.

[15] Alonzo Church. "A formulation of the simple theory of types". In: *Journal of Symbolic Logic* 5.2 (1940), pp. 56–68. DOI: 10.2307/2266170.

[16] Alonzo Church. "A formulation of the simple theory of types". In: *Journal of Symbolic Logic* 5.2 (1940), pp. 56–68. DOI: 10.2307/2266170.

[17] Zaynah Dargaye and Xavier Leroy. "Mechanized Verification of CPS Transformations". In: *LPAR 2007*. Ed. by Nachum Dershowitz and Andrei Voronkov. Vol. 4790. LNCS. Springer, 2007, pp. 211–225. DOI: 10.1007/978-3-540-75560-9_17.

[18] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0.

[19] Gilles Dowek and Murdoch J. Gabbay. "PNL to HOL: From the logic of nominal sets to the logic of higher-order functions". In: *Theoretical Computer Science* 451 (2012), pp. 38–69. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2012.06.007.

[20] Murdoch Gabbay. "Equivariant ZFA and the foundations of nominal techniques". In: *Journal of Logic and Computation* 30.2 (Mar. 2020), pp. 525–548. ISSN: 0955-792X. DOI: 10.1093/logcom/exz015. eprint: https://academic.oup.com/logcom/article-pdf/30/2/525/33039963/exz015.pdf.

[21] Murdoch Gabbay and Andrew M. Pitts. "A New Approach to Abstract Syntax Involving Binders". In: *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 1999, pp. 214–224. DOI: 10.1109/LICS.1999.782617.

[22] Murdoch Gabbay and Andrew M. Pitts. "A New Approach to Abstract Syntax with Variable Binding". In: *Formal Aspects Comput.* 13.3-5 (2002), pp. 341–363. DOI: 10.1007/s001650200016.

[23] Murdoch J. Gabbay and Aleksandar Nanevski. "Denotation of contextual modal type theory (CMTT): Syntax and meta-programming". In: *Journal of Applied Logic* 11.1 (2013), pp. 1–29. ISSN: 1570-8683. DOI: https://doi.org/10.1016/j.jal.2012.07.002.

[24] Jean-Yves Girard. "Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure". PhD thesis. Université Paris VII, 1972.

[25] Robert Harper, Furio Honsell, and Gordon D. Plotkin. "A Framework for Defining Logics". In: *LICS 1987*. IEEE Computer Society, 1987, pp. 194–204.

[26] Gérard P. Huet. "Residual Theory in lambda-Calculus: A Formal Development". In: *J. Funct. Program.* 4.3 (1994), pp. 371–394. DOI: 10.1017/S0956796800001106.

[27] Brian Huffman and Ondřej Kunčar. "Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL". In: *Certified Programs and Proofs: Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*. Melbourne, VIC, Australia: Springer-Verlag, 2013, pp. 131–146. ISBN: 978-3-319-03544-4. DOI: 10.1007/978-3-319-03545-1_9.

[28] Brian Huffman and Christian Urban. "A New Foundation for Nominal Isabelle". In: *ITP 2010*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. LNCS. Springer, 2010, pp. 35–50. DOI: 10.1007/978-3-642-14052-5_5.

[29] Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. "Infinitary Lambda Calculus". In: *Theor. Comput. Sci.* 175.1 (1997), pp. 93–125.

[30] B. Knaster. "Un théorème sur les fonctions d'ensembles." French. In: *Ann. Soc. Polon. Math.* 6 (1928), pp. 133–134.

[31] Felix Krayer. "Constructing Linear Types in Isabelle/HOL". Master's thesis. 2025.

[32] Paul Blain Levy. "Call-by-Push-Value: A Subsuming Paradigm". In: *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*. TLCA. Springer-Verlag, 1999, pp. 228–242. ISBN: 3540657630.

[33] Andreas Lochbihler and Joshua Schneider. "Relational Parametricity and Quotient Preservation for Modular (Co)datatypes". In: *Interactive Theorem Proving*. Ed. by Jeremy Avigad and Assia Mahboubi. Cham: Springer International Publishing, 2018, pp. 411–431. ISBN: 978-3-319-94821-8.

[34] Ernest G. Manes. *Algebraic Theories*. Springer, 1976.

[35] Simon Marlow, ed. *Haskell 2010 Language Report*. July 2010.

[36] Damiano Mazza. "An Infinitary Affine Lambda-Calculus Isomorphic to the Full Lambda-Calculus". In: *2012 27th Annual IEEE Symposium on Logic in Computer Science*. 2012, pp. 471–480. DOI: 10.1109/LICS.2012.57.

[37] Conor McBride and James McKinna. "Functional pearl: I am not a number—I am a free variable". In: *Haskell Workshop 2004*. Ed. by Henrik Nilsson. ACM, 2004, pp. 1–9. DOI: 10.1145/1017472.1017477.

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

[39] Robin Milner. *Communication and concurrency*. Prentice Hall, 1989.

[40] Eugenio Moggi. "Notions of Computation and Monads". In: *Inf. Comput.* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4.

[41] Eugenio Moggi. "Notions of computation and monads". In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 55–92. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(91)90052-4.

[42] 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*. 2001, pp. 267–282.

[43] Tobias Nipkow. "More Church-Rosser Proofs". In: *J. Autom. Reason.* 26.1 (2001), pp. 51–66. DOI: 10.1023/A:1006496715975.

[44] Michael Norrish and René Vestergaard. "Proof Pearl: De Bruijn Terms Really Do Work". In: *TPHOLs 2007*. Ed. by Klaus Schneider and Jens Brandt. Vol. 4732. LNCS. Springer, 2007, pp. 207–222. DOI: 10.1007/978-3-540-74591-4_16.

[45] Berk Özkütük. "Towards Binding-Aware Corecursion for Isabelle/HOL". Master's thesis. 2025.

[46] Lawrence C. Paulson. "A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle". In: *J. Autom. Reason.* 55.1 (2015), pp. 1–37. DOI: 10.1007/s10817-015-9322-8.

[47] Lawrence C. Paulson. "The Foundation of a Generic Theorem Prover". In: *J. Autom. Reason.* 5.3 (1989), pp. 363–397. DOI: 10.1007/BF00248324.

[48] Frank Pfenning and Conal Elliott. "Higher-Order Abstract Syntax". In: *PLDI 1988*. Ed. by Richard L. Wexelblat. ACM, 1988, pp. 199–208. DOI: 10.1145/53990.54010.

[49] Frank Pfenning and Carsten Schürmann. "System Description: Twelf – A Meta-Logical Framework for Deductive Systems". In: *CADE 1999*. Ed. by Harald Ganzinger. Vol. 1632. LNCS. Springer, 1999, pp. 202–206. DOI: 10.1007/3-540-48660-7_14.

[50] Brigitte Pientka and Jana Dunfield. "Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)". In: *IJCAR 2010*. Ed. by Jürgen Giesl and Reiner Hähnle. Vol. 6173. LNCS. Springer, 2010, pp. 15–21. DOI: 10.1007/978-3-642-14203-1_2.

[51] Andrew M. Pitts. "Alpha-Structural Recursion and Induction". In: *J. ACM* 53.3 (2006), pp. 459–506.

[52] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2013. DOI: 10.1017/CBO9781139084673.

[53] Tom Ridge and James Margetson. "A Mechanically Verified, Sound and Complete Theorem Prover for First Order Logic". In: *TPHOLs 2005*. Ed. by Joe Hurd and Thomas F. Melham. Vol. 3603. LNCS. Springer, 2005, pp. 294–309. DOI: 10.1007/11541868_19.

[54] Steven Schäfer, Tobias Tebbi, and Gert Smolka. "Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions". In: *ITP 2015*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. LNCS. Springer, 2015, pp. 359–374. DOI: 10.1007/978-3-319-22102-1_24.

[55] Natarajan Shankar. "A mechanical proof of the Church-Rosser theorem". In: *J. ACM* 35.3 (1988), pp. 475–522. DOI: 10.1145/44483.44484.

[56] Kathrin Stark. "Mechanising syntax with binders in Coq". PhD thesis. Saarland University, Saarbrücken, Germany, 2020.

[57] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. "Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions". In: *CPP 2019*. Ed. by Assia Mahboubi and Magnus O. Myreen. ACM, 2019, pp. 166–180. DOI: 10.1145/3293880.3294101.

[58] *Submitted Solutions to The POPLmark Challenge*. https://www.seas.upenn.edu/~plclub/poplmark/, accessed March 22, 2025. 2005.

[59] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications". In: *Pacific Journal of Mathematics* 5 (1955), pp. 285–309.

[60] Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. "A Sound and Complete Projection for Global Types". In: *ITP 2023*. Ed. by Adam Naumowicz and René Thiemann. Vol. 268. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 28:1–28:19. DOI: 10.4230/LIPICS.ITP.2023.28.

[61] 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*. IEEE Computer Society, 2012, pp. 596–605. DOI: 10.1109/LICS.2012.75.

[62] Christian Urban, Stefan Berghofer, and Cezary Kaliszyk. "Nominal 2". In: *Archive of Formal Proofs* (Feb. 2013). https://isa-afp.org/entries/Nominal2.html, Formal proof development. ISSN: 2150-914x.

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

[64] Christian Urban and Cezary Kaliszyk. "General Bindings and Alpha-Equivalence in Nominal Isabelle". In: *Log. Methods Comput. Sci.* 8.2 (2012). DOI: 10.2168/LMCS-8(2:14)2012.

[65] Jérôme Vouillon. "A Solution to the POPLMark Challenge Based on de Bruijn Indices". In: *J. Autom. Reason.* 49.3 (2012), pp. 327–362. DOI: 10.1007/S10817-011-9230-5.

[66] A.K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness". In: *Information and Computation* 115.1 (1994), pp. 38–94. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1093.