MIDDLESEX UNIVERSITY LONDON

DOCTORAL THESIS

A General Theory of Syntax with Bindings

Author:
Lorenzo GHERI

Supervisors:
Dr. Andrei POPESCU
Prof. Raja NAGARAJAN
Prof. Franco RAIMONDI

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Foundations of Computing Group
Department of Computer Science

April 23, 2019

Declaration of Authorship

I certify that this thesis, and the research to which it refers, are the product of my own work, and that any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

"*Death or glory becomes just another story"*

The Clash

MIDDLESEX UNIVERSITY LONDON

Abstract

Faculty of Science and Technology
Department of Computer Science

Doctor of Philosophy

A General Theory of Syntax with Bindings

by Lorenzo GHERI

In this thesis we give a general theory of syntax with bindings. We address the problem from a mathematical point of view and at the same time we give a formalization, in the Isabelle/HOL proof assistant.

Our theory uses explicit names for variables, and then deals with alpha-equivalence classes, remaining intuitive and close to informal mathematics, although being fully formalized and sound in classical high-order logic. In this sense it can be regarded as a generalization of nominal logic. Our end product can be used to construct complex binding patterns and binding-aware datatypes, including non-well-founded and infinitely branching types, in a modular fashion. We provide definitions of the fundamental operators on terms (free variables, alpha-equivalence, and capture-avoiding substitution) and reasoning and definition principles, obeying Barendregt's convention.

We present our work as a thinking process that starts from some desiderata, and then evolves in different formalization stages for the general theory. We start by taking a "universal algebra" approach, modelling syntaxes via algebraic-style binding signatures, which we employ in a substantial case study on formal reasoning: Church-Rosser and standardization theorems for λ-calculus. This solution proves itself too restrictive, so we refine it into a more flexible one, which constitutes the *main original contribution of this thesis*: We construct a universe of functors on sets that handle bindings on a general, flexible and modular level. Our functors do not commit to any *a priori* syntactic format, cater for codatatypes in addition to datatypes, and are supported by powerful definition and reasoning principles. They generalize the bounded natural functors (BNFs), which have been previously implemented in Isabelle/HOL to support (co)datatypes.

Acknowledgements

I thank my supervisor Andrei Popescu. He has been serious about my project and passionate about science. He has helped me with research and all the weirdness of the academic world. He has supported me, he has had an unbelievable amount of patience and overall he has been an excellent and pleasant person.

A special thank to the other members of my supervising team: Raja Nagarajan and Franco Raimondi. They helped me a lot with the university issues. In particular, I have to say that, during these PhD-in-London years, Franco has helped me to such an extent that I do not think I could ever settle the score. Was this not enough, he has also proved to be a very good friend.

I thank Marco Maggesi, who has always been interested in my work and happy to discuss science. He has never stopped welcoming me in his office in Florence and taking me to the bar for a *spuma*.

I thank Pietro Gheri, for taking the time to discuss group theory with me and for all the examples and clarifications he has provided.

I thank Christian Urban for the pleasant and motivating discussion and for the deep scientific insights that he provided me during my dissertation.

I thank Nikos Gorogiannis for the challenging and encouraging role he took during my dissertation and my whole period at Middlesex. He has been maybe the most stubborn person in believing that I could change from a mathematician to a computer scientist.

Thank you to all the people who had lunch with me at the university: their friendship (which goes from wine to cigars, from barbecues to rock concerts) helped me through my PhD more than they know.

Thanks a lot to all my friends in London: these guys shared with me the city, a flat, a house, some meal or, most likely, a couple of beers and a lot of thoughts.

Thanks a lot to all my friends in Florence, all the above holds for them as well (especially the beers part), but in this case the city is my city. They have taken good care of it. Here my honourable mention goes to the tenants of Casa Vandelli, maybe the most patient people of this whole page.

Thanks a lot to all my friends that are now spread across Tuscany, Europe and wherever. That these friendships are still alive means a lot.

The grand finale: my sincere thanks to Laura, Paolo, Pietro and Friedel. In no particular

order.

Contents

List of Figures

to Friedel and Pietro

Chapter 1

Introduction

This thesis focuses on developing a general theory of syntax with bindings, supported by a mechanization in the Isabelle/HOL proof assistant.

1.1 Problem Statement

Syntax with bindings is an essential ingredient in the formal specification and implementation of logics and programming languages. Reasoning about bindings becomes necessary in different contexts, such as when verifying properties of programming languages or when formalizing mathematics. In the first area a notable recent example is the CakeML project [43, 44]: in their huge verification effort, the team deals with different languages (hence with different syntaxes involving bindings) and translations from one to another. Another example, this time from mathematical logic, is the Isabelle/HOL formalization of Gödel's incompleteness theorems, by Lawrence Paulson [55], where the relevance of bindings is highlighted already in the abstract of the paper: "The Isabelle formalisation uses two separate treatments of variable binding: the nominal package is shown to scale to a development of this complexity, while de Bruijn indices turn out to be ideal for coding syntax."

Correctly and formally specifying, assigning semantics to, and reasoning about bindings is notoriously difficult and error-prone. This fact is widely recognized in the formal verification community, witnessed by examples such as the above, and reflected in the presence of manifestos and benchmarks such as the influential POPLmark challenge [7]. The origin of this difficulty is that every time a syntax contains bindings, its terms are equated via *alpha-equivalence*; namely the same syntactic objects may be written in several different ways. For example, let us consider the following paradigmatic example of syntax with bindings, the *untyped lambda-calculus*.

Example 1 (Syntax for the untyped λ-calculus)**.** *Every λ-term is either a variable x, the application of the term t_1 to the term t_2 or a λ-abstraction:*

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

Every λ-term is obtained by a finite number of applications of the three constructs above. Here the only binding construct is $\lambda x.t$; it binds the variable x in the term t.

In this case alpha-equivalence is the equivalence relation that equates, e.g., $\lambda x.xy$ and $\lambda z.zy$ (where x, y and z are variables); intuitively, we can think of the *bound* variable x in the first term as something that can be renamed, because it falls under the *scope* of the λ; in other

words, "x" is just a name used as a placeholder inside the binding construct. Of course the term $\lambda y.\, yy$ is not equivalent to the previous terms: in the former case y appears *free*, in the latter case y appears bound; if we use y for the renaming of x in $\lambda x.\, xy$, it gets *captured* by the binder, which is wrong.

There are some fundamental operations that can be performed at the syntactic level; the most notable one is the *substitution* of a term t' for a variable y in a term t, written $t[t'/y]$. Let us consider the previous term $\lambda x.\, xy$. Since bound variables can be renamed, it only makes sense to substitute a term t' for y, but not for x. At a first glance the resulting term will be $(\lambda x.\, xy)[t'/y] = \lambda x.\, xt'$. Of course this will hold if the term t' does not contain x, but what if it does? Let us take t' to be xy, then we want to express $(\lambda x.\, xy)[xy/y]$. The right way to do this, in order to avoid capture, is to exploit another representation of $\lambda x.\, xy$ alpha-equivalent to it, e.g. $\lambda z.\, zy$ with $z \notin \{x, y\}$, and then substitute, thus obtaining $\lambda z.\, z(xy)$.

One typically wants more structure on the considered syntax, and in particular to be able to define other operations and relations. Hence a general theory needs to be provided with useful *definition principles* (recursion) and *reasoning principles* (induction), that acknowledge the binding mechanism.

All the objects mentioned above are not peculiar to the λ-calculus, but are *common to many syntaxes*. For example, let us consider the following syntax, as presented in the POPLmark challenge [7] (in order to focus only on those elements of the syntax that are relevant to this discussion, some objects are omitted and some other are replaced by "..."):

Example 2 (Simplified Syntax for System $F_{<:}$)**.**

T	$::=$	$\ldots \mid$	*types*
t	$::=$	$\ldots \mid$	*terms*
		$\{l_i = t_i\ ^{i=1,\ldots,n}\} \mid$	*records*
		$t.l$	*projections*
		pattern-let $p = s$ in t	*pattern binding*
p	$::=$		*patterns*
		$x : T \mid$	*variable pattern*
		$\{l_i = p_i\ ^{i=1,\ldots,n}\}$	*record pattern*

where the pattern binding construct, pattern-let $p = s$ in t*, binds the pattern p in the term t. The* labels $l_i^{i\in\{1,\ldots,n\}}$ *appearing in a certain record are assumed to be pairwise distinct.*

System $F_{<:}$ extends the syntax of λ-calculus with record projections and pattern binding, and two other syntactic categories (types and patterns); these three distinct syntactic categories are defined mutually recursively. Moreover the binders involved have now increased complexity: the pattern p is an arbitrarily nested, recursively specified object, so the pattern-let construct binds in a term a structure containing an *arbitrary number* of variables. The POPLmark challenge sets some goals for the formalization of, and reasoning about this syntax. Here variable binding becomes a fundamental issue; in particular substitution needs to be well-defined in a capture-avoiding fashion, in order to obtain correctly

a set of inductively-defined rules regarding typing and evaluation, and to prove meaningful theorems from these.

We believe that such problems should be addressed in one shot with a general theory of syntax with bindings, that identifies the features common to such structures and provides a rich (and *sound*) reasoning and definitional infrastructure. Such a theory should be expressive enough to capture the wide variety of syntaxes belonging to the literature of mathematical logic and theoretical computer science, of which the just introduced λ-calculus and System $F_{<:}$ are fundamental instances. This is what we are set to achieve in this thesis. Binding and alpha-equivalence will be treated uniformly for the generic syntax, independently of the particular structures employed and of their complexity. In order to handle this complexity it is often useful to be able to nest previously-defined syntaxes in new ones, in a modular fashion. A rich theory of the main syntactic operators will be developed, in particular, substitution, freshness (or equivalently free variables) and swapping (renaming variables). Reasoning (induction) and definition (recursion) principles will be provided, and necessarily they will be tailored to a smooth treatment of non-free constructors. Assigning semantics to the syntax is an issue that can be covered to some extent by a general theory; this will engage our theory in a "confrontation" with infinitary structures. As an additional outcome of our effort to deal with infinitary structures, the theory will capture infinitely-branching terms and non-well-founded syntaxes.

The natural environment to implement this feature is a proof assistant—for our work we have chosen Isabelle/HOL. Once the formalization of the theory is instantiated, the user is provided with operators and theorems for their syntax, already defined and proved sound at the general level, once and for all.

1.2 State of the Art

The literature offers a variety of general frameworks that address the problem of modelling correctly syntaxes with bindings in their specification and behaviour.

A first popular example is higher order abstract syntax (HOAS) [34, 56]; here a fixed syntax is chosen as the metalanguage, then any other syntax is encoded using this privileged language. In particular the variables of the object syntax are represented as metavariables (variables in the metalogic) and the object binding structures are modelled exploiting the binders of the metalanguage. The adequacy of the encoding of the object syntax in the metalogic usually relies on pen and paper proofs. This approach has been shaped in different frameworks, e.g., weak HOAS [26] and parametric HOAS [23] (formalized in Coq).

Another approach is to treat bound variables as nameless, pointer-like objects. In this context many different frameworks have been developed, such as presheaf-based abstract syntax [29, 38, 3], binding signatures based on modules over monads [1] (building on [36, 37]; results checked in the UniMath library in Coq), bindings embedded in nested datatypes [11], bindings embedded in dependent types [2] (formalized in Agda), the locally nameless representation [6, 21], Autosubst [66] (the last two formalized in Coq).

The arguably most successful approach is the one originating with nominal logic [59,

58]. Nominal logic offers a theoretical foundation and systematization of syntax with binding, as well as prolific formalization work. The main development has been conducted in Isabelle/HOL (Nominal Isabelle [71] and Nominal2 [74, 75]). Prominent aspects of nominal techniques, such as Barendregt's variable convention, have been formalized in Coq [5] and Agda [25].

We will write "nominal approach" to indicate those developments that adopt explicit variable names and the native constructors of the syntax for specifying it. According to this informal convention, anachronistically Barendregt's work on λ-calculus [8] had already adopted the nominal approach; of course Pitts' work [59] was the real initiator. By comparison with HOAS and the nameless (or locally nameless) approaches, the main advantage of the nominal approach is the lack of encoding for constructors involving binding: using native constructors for the specification of the syntax does not require adequacy proofs and helps stating, proving and using reasoning and definition principles. Nominal formalizations have been very prolific and they provide a rich collection of formalized and sound results that ease reasoning about bindings. We can say, in the lax sense specified above, that our development also takes the nominal approach.

A more detailed discussion of related work is postponed to the end of the thesis (Section 7.1, Chapter 7), but, overall, we believe that, when aiming at both theoretical generality and practical usability, all these representations and systematizations have some features that can be significantly improved. Namely, we identify some major limitations:

1 **General Binders.** Most of the aforementioned works are not able to capture complex bindings with a general approach, however some techniques have been indeed deployed to cover some of the most famous examples from literature. Considering the POPLmark challenge as a significant instance, among the 15 solutions reported on the POPLmark website, only three address Parts 1B and 2B of the challenge, which involve recursively defined patterns (Example 2) and in each case, this is done through a low-level, ad hoc technical effort that is not entirely satisfactory. A notable exception is the Nominal2 development, which covers complex recursively specified bindings. Nominal2 does not feature primitive recursion for complex bindings, relying instead on general recursion as supported by Isabelle/HOL's powerful infrastructure, the function package [20, 41, 42].

2 **Modularity Features.** Modularity has not been addressed often in the literature. Moreover, to our knowledge there is no formalized framework for the specification of syntax with bindings, that allows to reuse a previously declared syntax in the specification of a new one with a smooth treatment of bindings.

3 **Infinitely Many Free Variables.** Most of the developed theories and techniques for syntaxes with bindings are limited to finitely supported syntaxes, in the sense that the terms of the syntax contain a finite number of free variables. Although these cover the majority of instances from literature, problems arise when we want to assign denotational semantics to the syntax, since often finite support is an unnecessary restriction for the semantic domains picked for the interpretation. Moreover there are some important examples of non-finitary syntaxes; in particular the same syntax specification, when interpreted in a

coinductive fashion, leads to the non-well-founded version of the syntax. For example, non-well-founded λ-terms (also known in literature as Böhm trees [8]).

Recalling Example 1, we define its following non-well-founded version.

Example 3 (Syntax for the non-well-founded untyped λ-calculus)**.** *Every* non-well-founded *λ-term is either a variable x, the application of the term t_1 to the term t_2 or a λ-abstraction:*

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

The only binding construct is $\lambda x.t$; it binds the variable x in the term t. Interpreting this specification coinductively, every λ-term is obtained by a possibly infinite *number of applications of the three constructs above, namely not only terms as $\lambda x.xy$ are allowed, but also terms as $\lambda x.xyxyxyxyx\ldots$ (here the term is obtained by repeating infinitely the application constructor).*

1.3 Our Improvement on the State of Art

Here we briefly describe the contribution of our work. In synthesis, we develop a theory at the same time general and expressive, for the specification of and reasoning about syntaxes with bindings, together with its Isabelle/HOL formalization. We address all the features described in Section 1.1—providing a rich infrastructure of operators (e.g. capture-avoiding substitution), lemmas, definition and reasoning principles—and in particular:

1 We capture binding structures uniformly, covering different examples from the literature, from fundamental ones (e.g., Example 1) to complex binders as the recursively specified pattern-let from the POPLmark Challenge (Example 2).

2 It will be possible to nest syntaxes specified in our framework, with their binding structures, in the definition of new ones in a modular fashion. This also means that properties and operators obtained about the nested syntax, will be still valid when ported to the wider context of the newly specified one.

3 Thanks to an insight of cardinality theory, our development will move from finite support to terms with infinitely many variables, with two generalizations: it will allow for infinitely branching syntaxes (e.g., calculus of communicating systems [51]) and for non-well-founded ones, as in Example 3. Other approaches that go beyond finitary support can be found in [22] and [30].

In the next section we give more details about the features that we consider fundamental to a theory like ours.

1.4 Technical Goals of Our Theory

Our work aims at systematizing and simplifying the task of constructing and reasoning about variable binding and variable substitution, namely the operations of binding variables into terms and of replacing them with other variables or terms in a well-scoped fashion. These mechanisms play a fundamental role in the metatheory of programming languages and logics.

The development of our theory has brought us to the development of two different frameworks, formalized in Isabelle/HOL. The two formalizations will be presented in this thesis, in a chronological order (Chapter 3 for the first one, and Chapters 5 and 6 for the second). The need for a second framework arose from the limitations we could find in the first (and in literature as well).

Below we give an overview of the desired features for a general theory of syntax with bindings and how we address those.

Complex Bindings and Modularity

We aim at a widely general framework, capturing a multitude of different syntaxes with bindings. We claim that all binding structures share the same fundamental mechanisms, therefore we treat uniformly the different instances of the theory, both the classic fundamental ones (for example λ-calculus, first order logic and π-calculi) and those that contain complex binding structures (just like the aforementioned System $F_{<:}$ from the POPLmark challenge). We address this via modularity: a complex (binding) structure can be defined as a syntax itself and then nested inside the new syntax. Here we can think of the syntax of System $F_{<:}$ and its pattern binding (Example 2): patterns can be defined as a standalone datatype (syntax, with no bindings) in a first moment and secondarily used within the pattern-let binder. Another, simpler, example is the following:

$$\begin{aligned} l &::= [] \mid t\#l \\ t &::= x \mid F\ l \\ \varphi &::= t_1 \equiv t_2 \mid P\ l \mid \varphi \wedge \psi \mid \neg\varphi \mid \forall v.\varphi \end{aligned}$$

The above is a common syntax specification for first order logic, with terms t and formulas φ, where functions symbols F and predicate symbols P take a list l as arguments. Here lists are mutually recursively defined together with terms and formulas, but we would like to simply define:

$$\begin{aligned} t &::= x \mid F\ ts \\ \varphi &::= t_1 \equiv t_2 \mid P\ ts \mid \varphi \wedge \psi \mid \neg\varphi \mid \forall v.\varphi \end{aligned}$$

and relying on the datatype construction built in the proof assistant to obtain *ts* as a simple list of terms of type $term_{\mathrm{FOL}}$ *list*. The advantage of this second definition is clear: we have at our disposal an already existing datatype, with all its properties (lemmas) and functions (e.g., the mapper map). To address this, we will introduce a class of functors to specify the constructors of the syntax. Functors can be composed and so can their map-actions, which will serve as the basic variable-renaming operators (we will expand on this in Section 5.2, Chapter 5). The first formulation, with lists mutually recursively defined with the other syntactic categories, is indeed equivalent to the second. In this case, however, we define a type of lists on terms that is *isomorphic* to $term_{\mathrm{FOL}}$ *list*, and therefore we will need to define a new mapper and prove properties about these new objects absolutely identical to the ones already provided for *list*. In our framework there is no need to do this, since, thanks to functor composition, the datatype of list is one and its properties still hold when nested.

Let us elaborate a bit more on the example above and consider a slightly different syntax for FOL but with one more constructor:

$$t ::= \ldots$$
$$\varphi ::= \ldots \mid \mathsf{conj}\ \varphi s$$

where conj represent the conjunction of a list of formulas $\varphi s : \mathit{formula}_{\mathrm{FOL}}\ \mathit{list}$. In this case a recursive occurrence of the currently defined type of formulas (which *can contain bindings*) appears nested in the list structure (a structure *defined elsewhere*). Our theory—thanks again to the properties of a particular class of functors—captures this kind of modularity, by simply enabling the user to freely nest any syntax with binders in any other during the specification process. One can argue that, in the particular example, there is no need to add such a feature, since FOL with conj is equivalent to FOL without it and with just the binary $\_ \wedge \_$. That is true, but if we substitute lists of formulas with *lazy lists* (possibly infinite lists) of formulas, $\mathit{formula}_{\mathrm{FOL}}\ \mathit{llist}$, we are capturing infinitary logic, where binary conjunction is not enough any more.

Basic Infrastructure and Substitution

We use explicit names for variables, we then work with terms as alpha-equivalence classes. We define operators on terms that are fundamental to reasoning about the syntax, in particular *freshness* (or equivalently *free variables*), *swapping* and *substitution*. Here we present them in the context of λ-terms (Example 1):

- the freshness predicate: $\mathsf{fresh}\ x\ t$ states that x is fresh for (i.e., does not occur free in) t; for example, it holds that $\mathsf{fresh}\ x\ (\lambda x.\, x)$ and $\mathsf{fresh}\ x\ y$ (when $x \neq y$), but not that $\mathsf{fresh}\ x\ x$; note that, in place of the freshness predicate, we can equivalently work with the set of free variables of a term, FVars, since it holds:

$$\mathsf{fresh}\ x\ t \quad \text{iff} \quad x \notin \mathsf{FVars}\ t \qquad \textit{and} \qquad \mathsf{FVars}\ t\ =\ \{x \mid \neg \mathsf{fresh}\ x\ t\}$$

- the swapping operator: $t[x \wedge y]$ indicates the term t where every occurrence (free or bound, indifferently) of the variable x has been replaced by an occurrence of y and vice versa; for example if t is $\lambda x.\, xy$ and $z \notin \{x, y\}$
  - $t[x \wedge y] = t[y \wedge x] = \lambda y.\, yx$
  - $t[x \wedge z] = t[z \wedge x] = \lambda z.\, zy$
  - $t[y \wedge z] = t[z \wedge y] = \lambda x.\, xz$
- the substitution operator: $t_1\,[t_2 / x]$ denotes the (capture-free) substitution of term t_2 for (all free occurrences of) variable x in term t_1; e.g., if t is $\lambda x.\, xy$ and $x \notin \{y, z\}$, then:
  - $t[z/y] = \lambda x.\, xz$
  - $t[z/x] = t$ (since bound occurrences like those of x in t are not affected)
  - $t[x/y] = \lambda z.\, zx$ (since after the substitution x becomes free in the term, the substitution operator needs to make sure that x is not captured by the binder, hence the bound variable must be renamed, e.g., to z which is available)

In particular the definition of substitution, mainly for its *capture-avoiding* behaviour, must be taken particular care of. However it is possible to define it in the theory independently of the syntax.

Beside substitution, freshness and swapping are pervasive in most logical systems and formal semantics of programming languages. The basic properties of these three operators lay at the core of important meta-theoretic results in these fields. Our theory provides a rich collection of results about these operators, so that using them is made easy, as independent of their formal definitions.

Beyond Finite Support

Extending the theory to infinitary syntaxes (namely, syntaxes with terms containing infinitely many variables) can be done in two orthogonal directions:

- **Infinitely Branching Syntaxes** For example, calculus of communicating systems includes infinitary sums [51]: the construct $\sum_{i\in I} P_i$ models nondeterministic choice from a collection $(P_i)_{i\in I}$ of processes indexed by a set I; it is important that I is allowed to be infinite, for modelling different decisions based on different received inputs;
- **Syntaxes Allowing Infinite Depth** This is a generalization of finitary syntaxes conceptually much more demanding than the previous one, since it requires to take a step from well-founded syntaxes, to non-well-founded ones (to this regard see Examples 1 and 3). This process of interpreting the specification of the syntax in a coinductive fashion, thus obtaining non-well-funded objects, can be done for all syntaxes and so it is a perfect candidate to be captured by a general theory.

Going beyond finite support in our development has been made possible by an insight of cardinality theory and the identification of *regularity* as the correct property to go after. We have developed the two theories, for well-founded and non-well-founded syntaxes, in parallel and, when possible, we pursued the duality aspects between the two.

(Co)Induction and (Co)Recursion Principles

Besides specification expressiveness, another criterion for assessing a formal framework is the amount of infrastructure built around the specification language, including reasoning and definitional mechanisms. Nominal logic ([59, 58]) provides a significant example for reasoning in presence of bindings: in this context a strong induction principle is introduced, known as "*fresh induction*". Thanks to fresh induction, when proving a property on terms by induction, bound variables can be assumed fresh for some parameters from the statement of the property. When defining functions on syntax, the situation is similar: particular care must be used in the presence of bindings. It is not possible to apply simple recursion on the constructors of the syntax, as it is commonly done where no bindings are involved. Let us consider the substitution for lambda calculus (Example 1): the two notations $\lambda x.\, xy$ and $\lambda z.\, zy$ indicate *the same* term, since they differ only in the name of the bound variable; hence $(\lambda x.\, xy)[x/y]$ and $(\lambda z.\, zy)[x/y]$ must lead to the same result $(\lambda z.\, zx)$, but it is clear that in the first case a renaming is needed in order to avoid capture. To this aim, recursion principles that behave well with respect to binding must be provided. Moreover, there is the non-well-founded case

that needs to be taken care of: symmetrically, coinduction and corecursion principles are also needed.

We formalize fresh induction and a recursion principles that work smoothly with bindings, thanks to the properties of freshness and swapping. We also develop a different recursor based on freshness and substitution (instead of swapping). When developing non-well-founded syntaxes, we provide a coinduction principle and a freshness-swapping corecursion, both adequate for dealing with bindings.

1.5 Isabelle/HOL Formalization

Informal techniques aimed at easing the reasoning tasks have turned out to be very difficult to represent formally, partly due to their reliance on unstated assumptions without which they would be unsound. For example, textbooks on λ-calculi employ the principle of primitive recursion to define functions on λ-terms, after which they tacitly assume these functions to be invariant under alpha-equivalence; as another example, the so-called Barendregt's variable convention assumes that, in a proof or definition context, the bound variables are fresh for all the parameters located outside the scope of their binders. Both these principles are unsound in general, that is, if employed without checking some sanity conditions on the defining clauses or on the definition and proof context.

Formal reasoning frameworks are designed to recover such informal principles on a sound basis. The approaches in literature range from a clever manipulation of the bound variables as in nominal logic and the locally named representation [59, 71, 47, 61] to the removal of the very notion of bound variable—by either encoding away bound variables as numeric positions in terms as in de Bruijn-style and locally nameless representations [19, 29, 21] or by representing them using meta-variables as in Higher-Order Abstract Syntax (HOAS) [34, 28].

In Section 1.1, we have seen the demand that comes for such frameworks to be efficient and powerful, from both applications (in verification [43, 44] and in the formalization of mathematics [55]) and explicit benchmarks (the POPLmark challenge [7]).

Therefore it was mandatory for our theory to be supported by a formalization. We have chosen the Isabelle/HOL proof assistant for this duty. Our development has evolved over the time span of this PhD and in this thesis it is presented in two frameworks. Although both formalizations take a nominal-style approach, they differ from nominal logic in important aspects, including operators-aware recursion principles, the presence of built-in substitution and features regarding infinitary syntaxes and modularity.

The formalization of the first framework (presented in Chapter 3) is a perfected version, which we have put to use with our case study from Chapter 4. The second formalization (Chapters 5 and 6) is a significant generalization of the first, that came naturally to address the limitations of the previous one. During the development for the second framework, its features—e.g., modularity, infinite structures, reasoning and definition principles—were constantly tested against Nominal Isabelle and Nominal2, in order to improve on these formalizations and on the state of the art in general. Our functors-based framework will form the

	Nominal Isabelle	Nominal2	Our Framework I	Our Framework II
Complex Binders	✗	✓	✗	✓
Modularity	✗	✓[1]	✗	✓
Built-In Substitution	✗	✗	✓	✓
Infinitely Branching Terms	✗	✗	✓	✓
Non-Well-Founded Terms	✗	✗	✗	✓
Reasoning Principles	✓	✓	✓	✓
Definition Principles	✓	✓[1]	✓	✓
Automated User Interface	✓	✓	✗	✗

✓: The feature has been formalized in the framework.
✓[1]: The feature has been partially formalized in the framework.
✗: The feature has not been formalized in the framework.

FIGURE 1.1: Comparison of our formalization with nominal

basis of a proper Isabelle package, for the specification of and reasoning about syntax with bindings, and it will be general enough to cover all the features that are aims of our work (Section 1.4).

In Figure 1.1, we show a synthetic comparison between our frameworks and the two existing Isabelle nominal frameworks; this is just intended as a tool to help the reader, but it is far from being exhaustive: further insights are needed. There are two fields marked with ✓[1]. One of these is modularity, which the nominal formalization covers up to some extent: considering the Modularity paragraph from the previous Section 1.4, Nominal2 can handle the first nesting example (nesting lists of terms in the definition of terms and formulas), but not the second, where lists of formulas (a type with bindings) are nested in the specification of formulas themselves. Concerning instead definition principles, Nominal2 covers some inductive ones, but lacks the implementation of primitive recursion. The comparison with nominal logic will be discussed throughout the whole thesis, given the importance of this theory, also in relation to our development. A wider discussion about related work is postponed to the conclusive Chapter 7.

1.6 Relevant Publications and Drafts

Lorenzo Gheri and Andrei Popescu. "A Formalized General Theory of Syntax with Bindings." In *Interactive Theorem Proving - 8th International Conference, ITP 2017*, Brasília, Brazil, September 26-29, 2017, Proceedings, pages 241–261, 2017. https://doi.org/10.1007/978-3-319-66107-0_16
An extended journal version of the paper has been accepted to *Journal of Automated Reasoning* (2019). https://doi.org/10.1007/s10817-019-09522-2

The paper presents the first formalization of our theory of syntax with bindings. It features many-sortedness of the syntax, infinitely branching terms, a rich theory of substitution, and induction and recursion principles tailored for a well behaviour in the presence of binders. The content is described in detail in Chapter 3.

Lorenzo Gheri and Andrei Popescu. "A Case Study in Reasoning about Syntax with Bindings: The Church-Rosser and Standardization Theorems." Draft, submitted. Available at https://sites.google.com/view/lorgheri/research

In the paper we instantiate the first framework for our general theory to the syntax of λ-calculus and formalize the development leading to two main results: the Church-Rosser and Standardization theorems. Our work covers both the call-by-name and call-by-value version of the calculus, following a classic paper by Gordon Plotkin. This important case study will be the object of Chapter 4 of the present dissertation.

Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. "Bindings As Bounded Natural Functors." In Weirich, S. (ed.) *46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019)*, ACM, 2019, Article 22, pp. 22:1–22:34. http://doi.acm.org/10.1145/3290335

In this paper we present our latest and most general framework for specifying and reasoning about syntax with bindings. Abstract binder types are modelled using a universe of functors on sets, subject to a number of operations that can be used to construct complex binding patterns and binding-aware datatypes, including non-well-founded and infinitely branching types, in a modular fashion. Again we cover the theory of classic operators, such as free-variables and capture-avoiding substitution, and we define binding-aware reasoning and definition principles. Among these features, modularity, complex binding structures and non-well-founded syntaxes are complete novelties with respect to our previous formalization. A significant improvement has been made also for recursion principles: the swapping-based principle developed in this work generalizes the general nominal recursor from the paper [58], Norrish's one [53] and our own from the first publication in this list. The process that brought us to this second development is described in Chapter 5, while the framework itself is presented in Chapter 6.

Formal Proof Developments

Lorenzo Gheri, Andrei Popescu. "A General Theory of Syntax with Bindings"
In *Archive of Formal Proofs*, 2019.

1.7 Structure of the Thesis

In this document we present a general theory theory of syntax with bindings. Central to the thesis will be the description of the two frameworks, with which we have formalized it. Before we move to this, we dedicate **Chapter 2** to the discussion of some technical preliminaries.

In **Chapter 3** we present in detail our first Isabelle/HOL framework, highlighting its main achievements, such as a rich theory of substitution, a substitution-freshness based recursion principle and a swapping based one, fresh induction and the interpretation of the generic syntax in semantics domains. Furthermore, we exploit this chapter to formally introduce all the main objects common to the different syntaxes with bindings, giving their formal definitions and the properties holding for them.

Chapter 4 contains an instance of the framework from the previous chapter. We show the benefits of a general theory of syntax with bindings when reasoning about a chosen syntax, facing actual formalization problems. As our case study, we have picked the call-by-name and call-by-value versions of the λ-calculus syntax, following a classic paper by Gordon Plotkin [60]. We formalize for both syntaxes the development leading to two major results: the Church-Rosser and Standardization theorems for β-reduction.

The time for meditation has come and in **Chapter 5** we reflect upon what we have achieved and the limitations of our formalization. We then start extending our theory to a new framework, based on functors. The last sections of the chapter are dedicated to the process we went through when modelling syntaxes with bindings by employing functors: we identify a special class of functors that we call *map restricted bounded natural functors* (MRBNFs).

Our second framework in its entirety is described in **Chapter 6**. Here we discuss the novelties of this work, among which the uniform treatment of binding structures (independently of their complexity), modularity features, non-well-founded syntaxes, and reasoning and definition principles suitable for smoothly dealing with bindings. In the case of definition principles, we also provide a comparison with recursors from literature: in the section we have dedicated to this, we choose and fix a simple syntax, so that we can focus on the features of the recursion principles themselves and not on the formalization choices.

The last **Chapter 7** is dedicated to literature review and a conclusive discussion about the overall merits of this thesis.

Chapter 2

Preliminaries

Our work relies on HOL (Section 2.1) and on BNFs (Sections 2.2 and 2.3). Higher order logic is simply the logic of our formalization environment, Isabelle/HOL, while bounded natural functors are a category-theoretic approach to defining and reasoning about types in a modular way. BNFs are the "ancestors" of MRBNFs, the class of functors on which our second framework is based (Chapters 5 and 6).

We also recall some of the fundamental concepts of nominal logic [59, 58], since we take an approach very close to the nominal one. We will need to refer to this important related work in different sections of the thesis (Section 2.4).

2.1 Higher-Order Logic

We consider classical higher-order logic with Hilbert choice, the axiom of infinity, and rank-1 polymorphism. HOL is based on simple type theory [24]. It is the logic of the original HOL system [33] and of HOL4, HOL Light, and Isabelle/HOL.

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

A *polymorphic type* is a type T that contains type variables. If T is polymorphic with variables $\overline{\alpha} = (\alpha_1, \ldots, \alpha_n)$, we sometimes write $\overline{\alpha}\, T$ instead of T. An *instance* of a polymorphic type is obtained by replacing some of its type variables with other types. For example, $(\alpha \to bool) \to \alpha$ is a polymorphic type, and $(ind \to bool) \to ind$ is an instance of it. A *polymorphic function* is a function that has a polymorphic type—for example, $\mathsf{Cons} : \alpha \to \alpha\ list \to \alpha\ list$. Semantically, we think of polymorphic functions as families of functions, one for each type—for example, the $\alpha := bool$ instance of Cons has type $bool \to bool\ list \to bool\ list$. *Formulas* are closed terms of type *bool*. Polymorphic formulas are thought of as universally quantified over their type variables. For example, $\forall x : \alpha.\ x = x$ really means $\forall \alpha.\ \forall x : \alpha.\ x = x$.

To keep the notation simple, we use type variables in two different ways: (1) as part of polymorphic types; and (2) as arbitrary but fixed types. For example, we can write (1) "α *list* is a polymorphic type" and (2) "given any type α and any function $f : \alpha \to \alpha$, such and such

holds." These conventions allow us to express the concepts in a more "semantic" style, closer to standard mathematical notation. The reader is free to think of types as nonempty sets.

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

2.2 Bounded Natural Functors

Often it is useful to think not in terms of polymorphic types, but in terms of type constructors. For example, *list* is a type constructor in one variable, whereas sum types ($+$) and product types ($\times$) are binary type constructors. Most type constructors are not only operators on types but have a richer structure, that of *bounded natural functors* [70].

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

Definition 4. Let $F = (F, \mathsf{map}_F, (\mathsf{set}^i_F)_{i\in[n]}, \mathsf{bd}_F)$, where

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

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

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

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

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

(Nat) each set^i_F is a natural transformation between the functor (F, map_F) and the powerset functor (set, image), namely the following diagram commutes (for every $f : \alpha \to \beta$):

$$\begin{array}{ccc} \alpha & \alpha\,F & \xrightarrow{\mathsf{set}_F} \alpha\ set \\ {\scriptstyle f}\downarrow & {\scriptstyle \mathsf{map}_F\ f}\downarrow & \downarrow{\scriptstyle \mathsf{image}} \\ \beta & \beta\,F & \xrightarrow[\mathsf{set}_F]{} \beta\ set \end{array}$$

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

FIGURE 2.1: map_F (left) and $\mathsf{rel}_F\ R$ (right)

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

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

Requiring that (F, rel_F) is a relator is equivalent to requiring that (F, map_F) preserves weak pullbacks [65]. It follows from the BNF axioms that the relator structure is an extension of the map function, in that mapping with a function f has the same effect as taking its graph $\mathsf{Gr}\ f$ and relating through $\mathsf{Gr}\ f$.

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

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

Consider a unary BNF F. For a fixed α, we represent a typical element of $x : \alpha\,F$ as depicted in Fig. 2.1, where we indicate the container as a triangle and its content via a typical atom $a : \alpha$. The left-hand side of the figure shows how mapping $f : \alpha \to \alpha'$ amounts to replacing each a with $f\ a$. The right-hand side shows how the relator applied to $R : \alpha \to \alpha' \to$ *bool* states that each a is R-related to an a' located at the same position in the container.

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

2.3 (Co)datatypes from Bounded Natural Functors

A *strong* BNF is a BNF whose map preserves not only weak pullbacks but also strong pullbacks. Strong BNFs include the basic type constructors of sum, product, and positive function space. Examples of BNFs that are not strong are the permutative (nonfree) type constructors, such as the finite powertype denoted $\alpha\ fset$ and the type of finite multisets (bags). Both the BNFs and the strong BNFs are closed under composition and (least and greatest) fixpoint definitions [70]. This enables us to mix and nest BNFs arbitrarily when defining (co)datatypes.

Datatypes $\overline{\alpha}\,T$, where $\overline{\alpha}$ is a tuple of type variables of length m, written $\mathsf{len}\,(\overline{\alpha}) = m$, can be defined recursively from $(m+1)$-ary BNFs $(\overline{\alpha}, \tau)\,F$, by taking their least fixpoint (initial

algebra): the minimal solution up to isomorphism of the recursive equation $\overline{\alpha}\,T \simeq (\overline{\alpha}, \overline{\alpha}\,T)\,F$ If instead we interpret the equation maximally—which we will indicate with the superscript ∞—it yields the codatatype $\overline{\alpha}\,T$. In either case, the construction yields an $\overline{\alpha}$-polymorphic bijection $\mathsf{ctor} : (\overline{\alpha}, \overline{\alpha}\,T)\,F \to \overline{\alpha}\,T$.

Abstractly, the difference between datatypes and codatatypes lies in their reasoning and definitional principles. For datatypes we have *structural induction*—which allows us to prove a that predicate holds on all its elements—and *recursion*—which allows us to define a function from the datatype to another type. Dually, for codatatypes we have *structural coinduction*—which allows us to prove that a binary relation is included in equality—and *corecursion*—which allows us to define functions from another type to the codatatype. Concretely, the difference can be understood in terms of well-foundedness: A datatype contains only well-founded entities, whereas a codatatype contains possibly non-well-founded ones. For example, if we take $(\alpha, \tau)\,F$ to be $\mathit{unit} + \alpha \times \tau$ (where *unit* is a fixed singleton type), the datatype defined as $\alpha\,T \simeq (\alpha, \alpha\,T)\,F$ is $\alpha\,\mathit{list}$, the type of (finite) lists. If instead we consider the maximal interpretation, $\alpha\,T \simeq^{\infty} (\alpha, \alpha\,T)\,F$, we obtain the codatatype of finite or infinite ("lazy") lists, $\alpha\,\mathit{llist}$. A substantial benefit of BNFs is that fixpoints can be freely nested in a modular fashion. Because $\alpha\,\mathit{llist}$ is itself a BNF, it can be used in further fixpoint definitions: Taking $(\alpha, \tau)\,F$ to be $\alpha + \tau\,\mathit{llist}$, the datatype $\alpha\,T$ defined as $\alpha\,T \simeq (\alpha, \alpha\,T)\,F$ is the type of α-labeled well-founded infinitely branching rose trees.

2.4 Group Actions, Finite Support and Nominal Logic

Our work adopts an approach similar to the one of nominal logic [59, 58]. To help the discussion with respect to this theory in this thesis, we introduce some basic concepts: *nominal sets* are defined starting from the notion of *finite support*, itself based on *actions of permutations*, a classical concept of group theory.

On the syntax of untyped λ-calculus (example 1, from subsection 1.1), the syntactic operation of *swapping* is defined. By swapping in a term t the variables x and y, a term $t\,[x \wedge y]$ is obtained where every occurrence (free or bound, indifferently) of the variable x has been replaced by an occurrence of y and vice versa. For example if t is $\lambda x.\,xy$ and $z \notin \{x, y\}$, then:

- $t\,[x \wedge y] = t\,[y \wedge x] = \lambda y.\,yx$
- $t\,[x \wedge z] = t\,[z \wedge x] = \lambda z.\,zy$
- $t\,[y \wedge z] = t\,[z \wedge y] = \lambda x.\,xz$

(A formal and complete definition of this operator will be given for the generic syntax in Chapter 3). Swapping takes a fundamental role for syntaxes with bindings: it embodies the basic intuition of *renaming* variables in terms. As we have seen in the just above example, in particular, when the variable z is not in the term swapping plugs it in, in place of another chosen one.

Now let us change our point of view and notation, thus getting closer to group theory and nominal logic [58]. For any given set S, we can consider the bijections $S \to S$, or

permutations[1] on S. The set of all permutations of any set S is a group, with respect to function composition, having the identity function as the identity element of the group. This is called the *symmetric group* of S, $Sym(S)$.

Definition 1. *An* action of a group G on a set S *is defined as a group homomorphism:*

$$\Phi \ : \ G \to Sym(S)$$

The defining properties of a group homomorphism (here we write them for Φ) are: (1) $\Phi\ 1_G = \mathsf{id}$, where 1_G is the identity element for the group G and (2) $\Phi(g \cdot h) = (\Phi\ h) \circ (\Phi\ g)$, where $\_ \cdot \_$ is the notation for the group operation of G. Since we deal only with groups of permutations, we can write these properties considering G as such, namely its group operation is the usual function composition and id_G is the identity:

1. $\Phi\ \mathsf{id}_G = \mathsf{id}_{Sym(S)}$;
2. $\Phi\ (h \circ g) = (\Phi\ h) \circ (\Phi\ g)$.

Note that now the order of g and h in $h \circ g$ is the opposite of their order in $g \cdot h$; this is because in the most common notation in group theory, when we write $g \cdot h$, morally g comes first; just like g comes first in $h \circ g$ since $(h \circ g)\ x = h(g\ x)$, namely we first apply g and then h. To avoid all these *contravariance* issues, in group theory $(h \circ g)\ x$ is written $x\ (g \cdot h)$ and it is said that functions are applied "on the left". We are dealing just with groups of bijections, so we will not go deeper into this and from now on we stick to the common practice of "applying functions on the right" and to the usual $\_ \circ \_$ notation for it. We will also drop the subscripts in $\mathsf{id}_G, \mathsf{id}_{Sym(S)}, \ldots$ when it is clear which is the group we are considering.

For our group actions we can lighten a bit the notation and, for every s element of S and g in G, write $g.s$ instead of $\Phi\ g\ s$. This brings us to the following equivalent definition of a group action:

Definition 2. *An* action of a group G on a set S *is defined as a function*

$$G \times S \to S$$
$$(g, s) \mapsto g.s$$

for which the following properties hold, for every $s \in S$ and $g, h \in G$:

1. $\mathsf{id}.s = s$;
2. $(h \circ g).s = h.(g.s)$.

Let us consider now an infinite set A of variables (still ranged over by $x, y, z, \ldots$), which from now on we call *atoms*, as is done in nominal logic. If we consider any two atoms $x, y \in A$, we note that the *transposition of x and y*, $(x \leftrightarrow y)$, defined by

$$(x \leftrightarrow y)\ z = \begin{cases} y & \text{if } z = x \\ x & \text{if } z = y \\ z & \text{otherwise} \end{cases}$$

is obviously a bijection, namely $(x \leftrightarrow y) \in Sym(A)$. Also it is uniquely identified by the atoms x and y, since $(x \leftrightarrow y) = (y \leftrightarrow x)$. Now, if we go back to the swapping operation on

[1] We will later call these "endobijections", when we abandon the group theory terminology.

λ-terms (just from here to the end of this section we call their set L), we can see that for every $(x \leftrightarrow y)$ as above (which is the same as saying "for every x and y") we get an element of $Sym(L)$, since $\_[x \wedge y] = \_[y \wedge x]$ is indeed a bijection.

We focus on well-founded λ-terms (obtained by a finite number of applications of the syntax constructs), so every term contains a finite number of variables (and of variable occurrences). Also swapping is finitary, in the sense that it "moves" just a finite number (two) of variables leaving all the other unchanged. These considerations suggest to restrict ourselves to $Sym_f(A)$—indeed a *subgroup* of $Sym(A)$—, namely *finitely supported bijections* of atoms, where the *support* of a function f is defined as $\mathsf{supp}\ f = \{x \mid f\ x \neq x\}$. This restriction to finitary objects is exactly what nominal logic does (e.g. in [58]). An action

$$Sym_f(A) \times L \to L$$

is defined—where, according to our previous notation, nominal logic picks the group G as $Sym_f(A)$ and the set on which it acts as L—by posing $g.t$ equal to that term where every occurrence (free or bound, indifferently) of the atom x has been replaced by an occurrence of $g(x)$, for all x simultaneously. Proving that the just-defined function is actually an action is a routine check.

If g is the transposition $(x \leftrightarrow y)$, we have that $g.t = (x \leftrightarrow y).t = t\ [x \wedge y]$, namely we obtain the swapping operator. Noticing that a classic result of group theory is that $Sym_f(A)$ is a group generated by transpositions, i.e., every finitely supported bijection is obtainable from the composition of a finite number of transpositions, we understand how swapping plays a central role with respect to this action and to the whole nominal theory.

Nominal logic is indeed a general theory of syntax with bindings: if we substitute the set of λ-terms L with any set (or *syntax*) S, on which is defined an action of the finitely-supported permutations of a set A of atoms (or *variables*), we automatically obtain on S the syntactic operator of swapping. Below we present some fundamental definitions of nominal logic that follow from this first idea. We fix a universe of atoms $\mathbb{A}$.

Definition 3 (Nominal Support)**.** *Let $A \subseteq \mathbb{A}$ be any set of atoms, S any set equipped with an action of $Sym_f(\mathbb{A})$ and $s \in S$.*

- *We say that A* supports *S if*

$$\forall\ a\ a'.\ a, a' \notin A \longrightarrow (a \leftrightarrow a')\ s = s$$

- *We say that s has* finite support *if there exist a set of atoms A such that A supports s and A is finite.*

Now we can define the key concepts of nominal logic.

Definition 4 (Nominal Set)**.** *We say that a set S is a* nominal set *if*

1. *S is equipped with an action of $Sym_f(\mathbb{A})$ and*
2. *every element $s \in S$ is supported by some finite set of atoms.*

Nominal logic deals exclusively with nominal sets and builds an expressive theory of syntaxes with bindings that covers many instances and goes up to induction and recursion

principles. Moreover for nominal sets we can obtain for free a rigorous definition of the freshness operator.

Definition 5 (Nominal Freshness)**.** *We say that an atom $a \in \mathbb{A}$* is fresh for *$s \in S$ if there exists some $A \subseteq \mathbb{A}$ such that A supports s and $a \notin A$.*

Note here that in nominal logic, if swapping is a primitive concept (we have to *assume* an action of atom permutations on our objects), freshness is not, but as a matter of fact its definition is derived from the one of swapping.

Chapter 3

A First Formalization: a Universal Algebra Approach

In this chapter we present a first Isabelle/HOL formalization of a theory of syntax with bindings. Terms are defined for an arbitrary number of constructors of varying numbers of inputs, quotiented to alpha-equivalence and sorted according to a binding signature. The theory includes a rich collection of properties of the standard operators on terms, including substitution, swapping and freshness—namely, there are lemmas showing how each of the operators interacts with all the others and with the syntactic constructors. The theory also features induction and recursion principles and support for semantic interpretation, all tailored for smooth interaction with the bindings and the standard operators.

This framework has evolved through the years. Its initial development (from Popescu's thesis [62]) has been a fundamental background work for our thesis and the first material we worked on ourselves. Starting from Chapter 5, until the end of this thesis, we will see a second evolution of the framework: while retaining the overall approach, a different underlying theory will be used to model complex bindings in a modular fashion.

The formalization in this chapter aims at mechanizing a form of *universal algebra for bindings*. This universal algebra approach, namely statically modelling the generic syntax by means of an instantiable signature, seemed to us the natural way to proceed in the first place. This first development successfully captures most of the fundamental features of a generic syntax with bindings and provides some original insights and principles, that are very useful in themselves and have constituted the basis for future developments, e.g., substitution-aware recursion (Section 3.3)—along with the formalization of Norrish's swapping-aware principle [52]—and the application of the theory of cardinality to infinitary syntaxes (Section 3.2.3).

The chapter is structured as follows. We start with an example-driven overview of our design decisions (Section 3.1). Then we present the general theory: terms as alpha-equivalence classes of "quasiterms," standard operators on terms and their basic properties (Section 3.2), custom induction (Section 3.4) and recursion schemes (Section 3.3), including support for the semantic interpretation of syntax, and the sorting of terms according to a signature (Section 3.5). In the next chapter, Chapter 4, we present an extensive case study for the framework, where we point out the usage of its various features.

3.1 Design Decisions

In this section, we present, via some examples, our design choices for the framework. We also introduce conventions and notations that will be relevant throughout this and the following chapter.

We start with the paradigmatic examples of λ-calculus [8]. We assume an infinite supply of variables, $x \in \mathit{var}$. The λ-terms, $X, Y \in \mathit{term}_\lambda$, are defined by the following standard datatype grammar:

$$X \quad ::= \quad \mathsf{Var}\ x \mid \mathsf{App}\ X\ Y \mid \mathsf{Lm}\ x\ X$$

Thus, a λ-term is either a variable, or an application, or a λ-abstraction. This grammar specification, while sufficient for first-order abstract syntax, is incomplete when it comes to syntax with bindings—we also need to indicate which operators introduce bindings and in which of their arguments. Here, Lm is the only binding operator: When applied to the variable x and the term X, it binds x in X. After knowing the binders, the usual convention is to *identify terms modulo alpha-equivalence*, i.e., to treat as equal terms that only differ in the names of bound variables, such as, e.g., $\mathsf{Lm}\ x\ (\mathsf{App}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ y))$ and $\mathsf{Lm}\ z\ (\mathsf{App}\ (\mathsf{Var}\ z)\ (\mathsf{Var}\ y))$. The end results of our theory will involve terms modulo alpha. We will call the raw terms "quasiterms," reserving the word "term" for alpha-equivalence classes.

3.1.1 Standalone Abstractions

To make the binding structure manifest, we will "quarantine" the bindings and their associated intricacies into the notion of *abstraction*, which is a pairing of a variable and a term, again modulo alpha. For example, for the λ-calculus we will have

$$X \ ::= \ \mathsf{Var}\ x \mid \mathsf{App}\ X\ Y \mid \mathsf{Lam}\ A \qquad\qquad A \ ::= \ \mathsf{Abs}\ x\ X$$

where X are terms and A abstractions. Within $\mathsf{Abs}\ x\ X$, we assume that x is bound in X. The λ-abstractions $\mathsf{Lm}\ x\ X$ of the the original syntax are now written $\mathsf{Lam}\ (\mathsf{Abs}\ x\ X)$.

Resorting to abstractions in order to isolate bindings has been our first means to clearly identify which variable was bound in which term. After we formalized the general framework and we moved to using it, instantiating it to particular instances, we had to hide those to the user. Moreover in our new framework (Chapters 5 and 6) we will definitively drop abstractions and just indicating explicitly the positions inside the constructor of which variables are bound in which terms.

3.1.2 Freshness, Substitution and Swapping

The three most fundamental and most standard operators on λ-terms are:

- the freshness predicate, $\mathsf{fresh} : \mathit{var} \to \mathit{term}_\lambda \to \mathit{bool}$, where $\mathsf{fresh}\ x\ X$ states that x is fresh for (i.e., does not occur free in) X; for example, it holds that $\mathsf{fresh}\ x\ (\mathsf{Lam}\ (\mathsf{Abs}\ x\ (\mathsf{Var}\ x)))$ and $\mathsf{fresh}\ x\ (\mathsf{Var}\ y)$ (when $x \neq y$), but not that $\mathsf{fresh}\ x\ (\mathsf{Var}\ x)$
- the substitution operator, $\_[\_/\_] : \mathit{term}_\lambda \to \mathit{term}_\lambda \to \mathit{var} \to \mathit{term}_\lambda$, where $Y\,[X/x]$ denotes the (capture-free) substitution of term X for (all free occurrences of) variable x in term Y; e.g., if Y is $\mathsf{Lam}\ (\mathsf{Abs}\ x\ (\mathsf{App}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ y)))$ and $x \notin \{y, z\}$, then:

  - $Y\,[(\mathsf{Var}\ z)/y] = \mathsf{Lam}\ (\mathsf{Abs}\ x\ (\mathsf{App}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ z)))$
  - $Y\,[(\mathsf{Var}\ z)/x] = Y$ (since bound occurrences like those of x in Y are not affected)

- the swapping operator $\_[\_ \wedge \_] : \mathit{term}_\lambda \to \mathit{var} \to \mathit{var} \to \mathit{term}_\lambda$, where $Y[x \wedge y]$ indicates the term Y where every occurrence (free or bound, indifferently) of the variable x has been replaced by an occurrence of y and vice versa; for example if Y is $\mathsf{Lam}\ (\mathsf{Abs}\ x\ (\mathsf{App}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ y)))$ and $z \notin \{x, y\}$
  - $Y\,[x \wedge y] = Y\,[y \wedge x] = \mathsf{Lam}\ (\mathsf{Abs}\ y\ (\mathsf{App}\ (\mathsf{Var}\ y)\ (\mathsf{Var}\ x)))$
  - $Y\,[x \wedge z] = Y\,[z \wedge x] = \mathsf{Lam}\ (\mathsf{Abs}\ z\ (\mathsf{App}\ (\mathsf{Var}\ z)\ (\mathsf{Var}\ y)))$
  - $Y\,[y \wedge z] = Y\,[z \wedge y] = \mathsf{Lam}\ (\mathsf{Abs}\ x\ (\mathsf{App}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ z)))$

And there are corresponding operators for abstractions—e.g., $\mathsf{freshAbs}\ x\ (\mathsf{Abs}\ x\ (\mathsf{Var}\ x))$ holds.

3.1.3 Advantages and Obligations from Working with Terms Modulo Alpha

In our theory, we start with defining quasiterms and quasiabstractions and their alpha-equivalence. Then, after proving all the syntactic constructors and standard operators to be compatible with alpha, we quotient to alpha, obtaining what we call terms and abstractions, and define the versions of these operators on quotiented items. For example, let qterm_λ and qabs_λ be the types of quasiterms and quasiabstractions in λ-calculus. Here, the quasiabstraction constructor, $\mathsf{qAbs} : \mathit{var} \to \mathit{qterm}_\lambda \to \mathit{qabs}_\lambda$, is a free constructor, of the kind produced by standard datatype specifications [10, 18]. The types term_λ and abs_λ are qterm_λ and qabs_λ quotiented to alpha. We prove compatibility of qAbs with alpha and then define $\mathsf{Abs} : \mathit{var} \to \mathit{term}_\lambda \to \mathit{abs}_\lambda$ by lifting qAbs to quotients.

The decisive advantages of working with quasiterms and quasiabstractions modulo alpha, i.e., with terms and abstractions, are that (1) substitution behaves well (e.g., is compositional) and (2) Barendregt's variable convention [8] (of assuming, w.l.o.g., the bound variables fresh for the parameters) can be invoked in proofs.

However, this choice brings the obligation to prove that all concepts on terms are compatible with alpha. Without employing suitable abstractions, this can become quite difficult even in the most "banal" contexts. Due to nonfreeness, primitive recursion on terms requires a proof that the definition is well formed, i.e., that the overlapping cases lead to the same result. As for Barendregt's convention, its rigorous usage in proofs needs a principle that goes beyond the usual structural induction for free datatypes.

A framework that deals gracefully with these obligations can make an important difference in applications—enabling the formalizer to quickly leave behind low-level "bootstrapping" issues and move to the interesting core of the results.

3.1.4 Many-Sortedness

While λ-calculus has only one syntactic category of terms (to which we added that of abstractions for convenience), this is often not the case. FOL has two: terms and formulas. The Edinburgh Logical Framework (LF) [34] has three: object families, type families and kinds. More complex calculi can have many syntactic categories.

Our framework will capture these phenomena. We will call the syntactic categories *sorts*. We will distinguish syntactic categories for terms (the sorts) from those for variables (the *varsorts*). Indeed, e.g., in FOL we do not have variables ranging over formulas, in the π-calculus [50] we have channel names but no process variables, etc.

Sortedness is important, but formally quite heavy. Here we present the core of the formalization, but at the same time we postpone dealing with sorts for as long as possible. We introduce an intermediate notion of *good* term, for which we are able to build the bulk of the theory—only as the very last step we introduce many-sorted signatures and transit from "good" to "sorted."

This is indeed how things have been formalized in the first framework: the notion of good terms ensured a good behaviour of the syntax with respect to cardinality issues. But, once we introduced it, all the theory was developed and just later organized according to sorts. In our new framework (Chapters 5 and 6), thanks to functors and ideas from the BNFs' theory (Sections 2.2 and 2.3, Chapter 2), we did not need any more this concept of "goodness" and hence dismissed it.

3.1.5 Possibly Infinite Branching

Nominal Logic's [59, 76] notion of finite support has become central in state-of-the-art techniques for reasoning about bindings. However, important developments step outside finite support and here we first encounter with such syntaxes. We have already discussed this point in Section 1.4 from Chapter 1 and how it is desirable to go in two orthogonal directions: allowing for infinitely-branching terms and capturing non well-founded syntaxes. Our next framework (Chapters 5 and 6) will cover both aspects while this formalization captures only infinitely-branching objects, but the crucial means to go beyond finite support, is indeed the understanding of regular cardinals we had for this first development.

We now consider a motivating example of infinitely-branching syntaxes: (a simplified version of) CCS [51]. This syntax has the following syntactic categories of data expressions $E \in exp$ and processes $P \in proc$:

$$E ::= \mathsf{Var}\ x \mid 0 \mid E + E \qquad\qquad P ::= \mathsf{Inp}\ c\ x\ P \mid \mathsf{Out}\ c\ E\ P \mid \textstyle\sum_{i \in I} P_i$$

Above, $\mathsf{Inp}\ c\ x\ P$, usually written $c(x).P$, is an input prefix $c(x)$ followed by a continuation process P, with c being a channel and x a variable which is bound in P. Dually, $\mathsf{Out}\ c\ E\ P$, usually written $c\overline{E}.P$, is an output-prefixed process with E an expression. The exotic constructor here is the sum $\sum$, which models nondeterministic choice from a collection $(P_i)_{i \in I}$ of alternatives indexed by a set I. It is important that I is allowed to be infinite, for modelling different decisions based on different received inputs. But then process terms may use infinitely many variables, i.e., may not be finitely supported. Similar issues arise in infinitary FOL [40] and Hennessey-Milner logic [35].

3.2 General Terms with Bindings

We start the presentation of our formalized theory, in its journey from quasiterms (3.2.1) to terms via alpha-equivalence (3.2.2). The journey is fuelled by the availability of fresh

variables, ensured by cardinality assumptions on constructor branching and variables (3.2.3). It culminates with a systematic study of the standard term operators (3.2.4).

3.2.1 Quasiterms

The types *qterm* and *qabs*, of quasiterms and quasiabstractions, are defined as mutually recursive datatypes polymorphic in the following type variables: *index* and *bindex*, of indexes for free and bound arguments, *varsort*, of varsorts, i.e., sorts of variables, and *opsym*, of (constructor) operation symbols. For readability, below we omit the occurrences of these type variables as parameters to *qterm* and *qabs*:

$$\begin{array}{lrcl} \mathsf{datatype} & \mathit{qterm} & = & \mathsf{qVar}\ \mathit{varsort}\ \mathit{var} \mid \\ & & & \mathsf{qOp}\ \mathit{opsym}\ ((\mathit{index},\mathit{qterm})\,\mathit{input})\ ((\mathit{bindex},\mathit{qabs})\,\mathit{input}) \\ \mathsf{and} & \mathit{qabs} & = & \mathsf{qAbs}\ \mathit{varsort}\ \mathit{var}\ \mathit{qterm} \end{array}$$

Thus, any quasiabstraction has the form qAbs *xs x X*, putting together the variable x of varsort *xs* with the quasiterm X, indicating the binding of x in X. On the other hand, a quasiterm is either an injection qVar *xs x*, of a variable x of varsort *xs*, or has the form qOp δ *inp binp* , i.e., consists of an operation symbol applied to some inputs that can be either free, *inp*, or bound, *binp*.

We use $(\alpha,\beta)\,\mathit{input}$ as a type synonym for $\alpha \to \beta\ \mathit{option}$, the type of partial functions from α to β; such a function returns either None (representing "undefined") or Some b for $b : \beta$, namely:

$$\mathsf{datatype}\ \beta\ \mathit{option}\ =\ \mathsf{None} \mid \mathsf{Some}\ \beta\ .$$

This type models inputs to the quasiterm constructors of varying number of arguments. An operation symbol $\delta : \mathit{opsym}$ can be applied, via qOp, to: (1) a varying number of free inputs, i.e., families of quasiterms modeled as members of $(\mathit{index},\mathit{qterm})\,\mathit{input}$ and (2) a varying number of bound inputs, i.e., families of quasiabstractions modeled as members of $(\mathit{index},\mathit{qabs})\,\mathit{input}$. For example, taking *index* to be *nat* we capture n-ary operations for any n (passing to qOp δ inputs defined only on $\{0,\ldots,n-1\}$), as well as as countably-infinitary operations (passing to qOp δ inputs defined on the whole *nat*). Therefore in our framework, for Milner's CCS (see Section 3.1.5 above), e.g., we will have:

$$\sum_{i\in\mathbb{N}} P_i \quad \text{represented as} \quad \mathsf{qOp}\ \textstyle\sum_{\mathbb{N}}\ f\ ,$$

where $\sum_{\mathbb{N}}$ is an operation symbol and f is a partial function from natural numbers to processes, with $f(i) = P_i$. In our just introduced notation f has type $(\mathit{index},\mathit{qterm})\,\mathit{input}$, where we have taken *index* to be the type of natural numbers and *qterm* is the type of quasi-terms for processes (for more details see Section 3.5.1, where sorting and binding signatures are introduced).

Note that, so far, we consider sorts of variables but not sorts of terms. The latter will come much later, in Section 3.5, when we introduce signatures. Then, we will gain control (1) of which varsorts should be embedded in which term sorts and (2) of which operation symbols are allowed to be applied to which sorts of terms. But, until then, we will develop the interesting part of the theory of bindings without sorting the terms.

$$
\begin{array}{rcl}
\mathsf{alpha}\ (\mathsf{qVar}\ xs\ x)\ (\mathsf{qVar}\ xs'\ x') & \longleftrightarrow & xs = xs' \wedge x = x' \\
\mathsf{alpha}\ (\mathsf{qOp}\ \delta\ inp\ binp)\ (\mathsf{qOp}\ \delta'\ inp'\ binp') & \longleftrightarrow & \delta = \delta' \wedge \uparrow \mathsf{alpha}\ inp\ inp' \wedge \uparrow \mathsf{alphaAbs}\ binp\ binp' \\
\mathsf{alpha}\ (\mathsf{qVar}\ xs\ x)\ (\mathsf{qOp}\ \delta'\ inp'\ binp') & \longleftrightarrow & \mathsf{False} \\
\mathsf{alpha}\ (\mathsf{qOp}\ \delta\ inp\ binp)\ (\mathsf{qVar}\ xs'\ x') & \longleftrightarrow & \mathsf{False} \\
\mathsf{alphaAbs}\ (\mathsf{qAbs}\ xs\ x\ X)\ (\mathsf{qAbs}\ xs'\ x'\ X') & \longleftrightarrow & xs = xs' \wedge (\exists y \notin \{x, x'\}.\ \mathsf{qFresh}\ xs\ y\ X \wedge \\
 & & \mathsf{qFresh}\ xs\ y\ X' \wedge \mathsf{alpha}\ (X[y \wedge x]_{xs})\ (X'[y \wedge x']_{xs}))
\end{array}
$$

FIGURE 3.1: Alpha-Equivalence

On quasiterms, we define freshness, $\mathsf{qFresh} : varsort \to var \to qterm \to bool$, substitution, $\_[\_/\_]\_ : qterm \to qterm \to var \to varsort \to qterm$, parallel substitution, $\_[\_] : qterm \to (varsort \to var \to qterm\ option) \to qterm$, swapping, $\_[\_ \wedge \_]\_ : qterm \to var \to var \to varsort \to qterm$, and alpha-equivalence, $\mathsf{alpha} : qterm \to qterm \to bool$—and corresponding operators on quasiabstractions: qFreshAbs, alphaAbs, etc.

The definitions proceed as expected, with picking suitable fresh variables in the case of substitutions and alpha. For parallel substitution, given a (partial) variable-to-quasiterm assignment $\rho : varsort \to var \to qterm\ option$, the quasiterm $X[\rho]$ is obtained by substituting, for each free variable x of sort xs in X for which ρ is defined, the quasiterm Y where $\rho\ xs\ x = \mathsf{Some}\ Y$. We only show the formal definition of alpha-equivalence in the following subsection. In Subsection 3.2.4 however we will show the lifted (and more significant) versions of these for terms, after the quotienting to alpha-equivalence has already happened. In each case the standard (and quite tedious process) works like this: we define the operator on quasi-terms, we prove that the defined concept respects alpha-equivalence classes, we lift it to terms and prove lemmas that can be used as if the concept had been defined directly on terms.

3.2.2 Alpha-Equivalence

We define the predicates alpha (on quasiterms) and alphaAbs (on quasiabstractions) mutually recursively, as shown in Fig. 3.1. For variable quasiterms, we require equality on both the variables and their sorts. For qOp quasiterms, we recurse through the components, *inp* and *binp*. Given any predicate $P : \beta^2 \to bool$, we write $\uparrow P$ for its lifting to $(\alpha,\beta)\ input^2 \to bool$, defined as

$\uparrow P\ inp\ inp' \longleftrightarrow$

$\forall i.\ \mathsf{case}\ (inp\ i, inp'\ i)\ \mathsf{of}\ (\mathsf{None}, \mathsf{None}) \Rightarrow \mathsf{True} \mid (\mathsf{Some}\ b, \mathsf{Some}\ b') \Rightarrow P\ b\ b' \mid \_ \Rightarrow \mathsf{False}$

Thus, $\uparrow P$ relates two inputs just in case they have the same domain and their results are componentwise related.

Convention 5. Throughout this chapter, without further notice we write $\uparrow$ for the natural lifting of the various operators from terms and abstractions to free or bound inputs.

In Fig. 3.1's clause for quasiabstractions, we require that the bound variables are of the same sort and there exists some fresh y such that alpha holds for the terms where y is swapped with the bound variable. Following Nominal Logic, we prefer to use swapping instead of substitution in alpha-equivalence, since this leads to simpler proofs [58].

3.2.3 Good Quasiterms and Regularity of Variables

In general, alpha will not be an equivalence, namely, will not be transitive: Due to the arbitrarily wide branching of the constructors, we may not always have fresh variables y available in an attempt to prove transitivity by induction. To remedy this, we restrict ourselves to "good" quasiterms, whose constructors do not branch beyond the cardinality of *var*. Goodness is defined as the mutually recursive predicates qGood and qGoodAbs:

$$\begin{array}{rcl} \mathsf{qGood}\ (\mathsf{qVar}\ xs\ x) & \iff & \mathsf{True} \\ \mathsf{qGood}\ (\mathsf{qOp}\ \delta\ inp\ binp) & \iff & \uparrow\mathsf{qGood}\ inp\ \wedge\ \uparrow\mathsf{qGoodAbs}\ binp\ \wedge \\ & & |\mathsf{dom}\ inp| < |var|\ \wedge\ |\mathsf{dom}\ binp| < |var| \\ \mathsf{qGoodAbs}\ (\mathsf{qAbs}\ xs\ x\ X) & \iff & \mathsf{qGood}\ X \end{array}$$

where, given a partial function f, we write dom f for its domain.

Thus, for good items, we hope to always have a supply of fresh variables. Namely, we hope to prove $\mathsf{qGood}\ X \Longrightarrow \forall xs.\ \exists x.\ \mathsf{qFresh}\ xs\ x\ X$. But goodness is not enough. We also need a special property for the type *var* of variables. In the case of finitary syntax, it suffices to take *var* to be countably infinite, since a finitely branching term will contain fewer than $|var|$ variables (here, meaning a finite number of them)—this can be proved by induction on terms, using the fact that a finite union of finite sets is finite.

So let us attempt to prove the same in our general case. In the inductive qOp case, we know from goodness that the branching is smaller than $|var|$, so to conclude we would need the following: *A union of sets smaller than* $|var|$ *indexed by a set smaller than* $|var|$ *stays smaller than* $|var|$. It turns out that this is a well-studied property of cardinals, called *regularity*—with $|nat|$ being the smallest regular cardinal. (In addition, we know that for every cardinal k there exists a regular cardinal larger than k.) Thus, the desirable generalization of countability is regularity (which is available from Isabelle's cardinal library [12]). Henceforth, we will assume:

Assumption 6. $|var|$ is a regular cardinal.

We will thus have not only one, but a $|var|$ number of fresh variables:

Proposition 7. $\mathsf{qGood}\ X \Longrightarrow \forall xs.\ |\{x.\ \mathsf{qFresh}\ xs\ x\ X\}| = |var|$

Now we can prove, for good items, the properties of alpha familiar from the λ-calculus, including it being an equivalence and an alternative formulation of the abstraction case, where "there exists a fresh y" is replaced with "for all fresh y." While the "exists" variant is useful when proving that two terms are alpha-equivalent, the "forall" variant gives stronger inversion and induction rules for proving implications from alpha. (Such fruitful "exsist-fresh/forall-fresh," or "some-any" dychotomies have been previously discussed in the context of bindings, e.g, in [54, 6, 48].)

Proposition 8. The following hold:
(1) alpha and alphaAbs are equivalences on good quasiterms and quasiabstractions
(2) The predicates defined by replacing, in Fig. 3.1's definition, the abstraction case with

$$\mathsf{alphaAbs}\ (\mathsf{qAbs}\ xs\ x\ X)\ (\mathsf{qAbs}\ xs'\ x'\ X') \iff$$
$$xs = xs' \wedge (\forall y \notin \{x, x'\}.\ \mathsf{qFresh}\ xs\ y\ X \wedge \mathsf{qFresh}\ xs\ y\ X' \Longrightarrow \mathsf{alpha}\ (X[y \wedge x]_{xs})\ (X'[y \wedge x']_{xs}))$$

Constructors
Var : *varsort* → *var* → *term*
Op : *opsym* → (*index*, *term*) *input* → (*bindex*, *abs*) *input* → *term*
Abs : *varsort* → *var* → *term* → *abs*

Operators on terms and abstractions		
Freshness	terms	fresh : *varsort* → *var* → *term* → *bool*
	abstractions	freshAbs : *varsort* → *var* → *abs* → *bool*
Substitution	terms	_[_/_]_ : *term* → *term* → *var* → *varsort* → *term*
	abstractions	_[_/_]_ : *abs* → *term* → *var* → *varsort* → *abs*
Parallel Substitution	terms	_[_] : *term* → (*varsort* → *var* → *term option*) → *term*
	abstractions	_[_] : *abs* → (*varsort* → *var* → *term option*) → *abs*
Swapping	terms	_[_ ∧ _]_ : *term* → *var* → *var* → *varsort* → *term*
	abstractions	_[_ ∧ _]_ : *abs* → *var* → *var* → *varsort* → *abs*

FIGURE 3.2: Constructors and operators on terms and abstractions

coincide with alpha and alphaAbs.

3.2.4 Terms and Their Properties

We define *term* and *abs* as collections of alpha- and alphaAbs- equivalence classes of *qterm* and *qabs*. Since qGood and qGoodAbs are compatible with alpha and alphaAbs, we lift them to corresponding predicates on terms and abstractions, good and goodAbs.

We also prove that all constructors and operators are alpha-compatible, which allows lifting them to terms. Figure 3.2 shows the types of all these term constructors and operators.

To establish an abstraction barrier that sets terms free from their quasiterm origin, we prove that the syntactic constructors mostly behave like free constructors, in that Var, Op and Abs are exhaustive and Var and Op are injective and nonoverlapping. True to the quarantine principle expressed in Subsection 3.1.1, the only nonfreeness incident occurs for Abs. Its equality behavior is regulated by the "exists fresh" and "forall fresh" properties inferred from the definition of alphaAbs and Prop. 8(2), respectively:

Proposition 9. Assume good X and good X'. Then the following are equivalent:
(1) Abs $xs\ x\ X$ = Abs $xs'\ x'\ X'$
(2) $xs = xs' \wedge (\exists y \notin \{x, x'\}.\ \mathsf{fresh}\ xs\ y\ X \wedge \mathsf{fresh}\ xs\ y\ X' \wedge X[y \wedge x]_{xs} = X'[y \wedge x']_{xs})$
(3) $xs = xs' \wedge (\forall y \notin \{x, x'\}.\ \mathsf{fresh}\ xs\ y\ X \wedge \mathsf{fresh}\ xs\ y\ X' \Longrightarrow X[y \wedge x]_{xs} = X'[y \wedge x']_{xs})$

Useful rules for abstraction equality also hold with substitution:

Proposition 10. Assume good X and good X'. Then the following hold:
(1) $y \notin \{x, x'\} \wedge \mathsf{fresh}\ xs\ y\ X \wedge \mathsf{fresh}\ xs\ y\ X' \wedge X\,[(\mathsf{Var}\ xs\ y)\ /\ x]_{xs} = X'\,[(\mathsf{Var}\ xs\ y)\ /\ x']_{xs} \Longrightarrow$
Abs $xs\ x\ X$ = Abs $xs\ x'\ X'$
(2) fresh $xs\ y\ X \implies$ Abs $xs\ x\ X$ = Abs $xs\ y\ (X\,[(\mathsf{Var}\ xs\ y)\ /\ x]_{xs})$

To completely seal the abstraction barrier, for all the standard operators we prove simplification rules regarding their interaction with the constructors, which makes the former behave as if they had been defined in terms of the latter.

The following facts resemble an inductive definition of freshness (as a predicate):

Proposition 11. Assume good X, $\uparrow$good *inp*, $\uparrow$good *binp*, |dom *inp*| < |*var*| and |dom *binp*| < |*var*|. The following hold:
(1) $(ys, y) \neq (xs, x) \Longrightarrow \mathsf{fresh}\ ys\ y\ (\mathsf{Var}\ xs\ x)$
(2) $\uparrow(\mathsf{fresh}\ ys\ y)\ inp \wedge \uparrow(\mathsf{freshAbs}\ ys\ y)\ binp \Longrightarrow \mathsf{fresh}\ ys\ y\ (\mathsf{Op}\ \delta\ inp\ binp)$
(3) $(ys, y) = (xs, x) \vee \mathsf{fresh}\ ys\ y\ X \Longrightarrow \mathsf{freshAbs}\ ys\ y\ (\mathsf{Abs}\ xs\ x\ X)$

Here and elsewhere, when dealing with Op, we make cardinality assumptions on the domains of the inputs to make sure the terms Op δ *inp binp* are good.

We can further improve on Prop. 11, obtaining "iff" facts that resemble a primitively recursive definition of freshness (as a function):

Proposition 12. Prop. 11 stays true if the implications are replaced by equivalences ($\Longleftrightarrow$).

For the swapping and substitution operators, we prove the following simplification rules, with a similar primitive recursion flavor.

Proposition 13. Assume good X, $\uparrow$good *inp*, $\uparrow$good *binp*, |dom *inp*| < |*var*| and |dom *binp*| < |*var*|. The following hold:
(1) $(\mathsf{Var}\ xs\ x)\,[y \wedge z]_{ys} = \mathsf{Var}\,(x\,[y \wedge z]_{xs,ys})$
(2) $(\mathsf{Op}\ \delta\ inp\ binp)\,[y \wedge z]_{ys} = \mathsf{Op}\ \delta\ (\uparrow(\_[y \wedge z]_{ys})\ inp)\ (\uparrow(\_[y \wedge z]_{ys})\ binp)$
(3) $(\mathsf{Abs}\ xs\ x\ X)\,[y \wedge z]_{ys} = \mathsf{Abs}\ xs\ (x\,[y \wedge z]_{xs,ys})\ (X\,[y \wedge z]_{ys})$
where $x\,[y \wedge z]_{xs,ys}$ is the (sorted) swapping on variables, defined as

$$\text{if } (xs, x) = (ys, y) \text{ then } y \text{ else if } (xs, x) = (ys, z) \text{ then } z \text{ else } x$$

Proposition 14. Assume good X, good Y, $\uparrow$good *inp*, $\uparrow$good *binp*, |dom *inp*| < |*var*| and |dom *binp*| < |*var*|. The following hold:
(1) $(\mathsf{Var}\ xs\ x)\,[Y/y]_{ys} = (\text{if } (xs, x) = (ys, y) \text{ then } Y \text{ else } \mathsf{Var}\ xs\ x)$
(2) $(\mathsf{Op}\ \delta\ inp\ binp)\,[Y/y]_{ys} = \mathsf{Op}\ \delta\ (\uparrow(\_[Y/y]_{ys})\ inp)\ (\uparrow(\_[Y/y]_{ys})\ binp)$
(3) $(xs, x) \neq (ys, y) \wedge \mathsf{fresh}\ xs\ x\ Y \Longrightarrow (\mathsf{Abs}\ xs\ x\ X)\,[Y/y]_{ys} = \mathsf{Abs}\ xs\ x\ (X\,[Y/y]_{ys})$

Since unary substitution is a particular case of parallel substitution, the previous lemma is a corollary of the following:

Proposition 15. Assume good X, $\uparrow$good *inp*, $\uparrow$good *binp*, $\uparrow$good ρ, |dom *inp*| < |*var*|, |dom *binp*| < |*var*| and |dom ρ| < |*var*|. The following hold:
(1) $(\mathsf{Var}\ xs\ x)\,[\rho] = (\text{if } \rho\ xs\ x = \mathsf{Some}\ Y \text{ then } Y \text{ else } \mathsf{Var}\ xs\ x)$
(2) $(\mathsf{Op}\ \delta\ inp\ binp)\,[\rho] = \mathsf{Op}\ \delta\ (\uparrow(\_[\rho])\ inp)\ (\uparrow(\_[\rho])\ binp)$
(3) $\uparrow(\mathsf{fresh}\ xs\ x)\ \rho \Longrightarrow (\mathsf{Abs}\ xs\ x\ X)\,[\rho] = \mathsf{Abs}\ xs\ x\ (X\,[\rho])$

Above, the notation $\uparrow(\mathsf{fresh}\ xs\ x)\ \rho$ follows the spirit of Convention 5, in that it lifts the freshness predicate from terms to environments for parallel substitution, i.e., partial variable-to-term assignments $\rho : varsort \to var \to term\ option$. However, it must be noted that this is a non-standard lifting process, referring not only to the freshness on ρ's image terms, but also to *distinctness* on ρ's domain variables. Namely, $\uparrow(\mathsf{fresh}\ xs\ x)\ \rho$ is defined as

$$\forall\, ys, y, Y.\ \rho\ ys\ y = \mathsf{Some}\ Y \Longrightarrow (xs, x) \neq (ys, y) \wedge \mathsf{fresh}\ xs\ x\ Y$$

Thus, (xs, x) must be fresh for the graph of (the uncurried version of) the partial function ρ.

Note that, when it comes to the interaction of freshness and substitution with Abs, the simplification rules require freshness of the bound variable. Thus, $\mathsf{freshAbs}\ ys\ y\ (\mathsf{Abs}\ xs\ x\ X)$ is reducible to $\mathsf{fresh}\ ys\ y\ X$ only if (xs, x) is distinct from (ys, y). Moreover, $(\mathsf{Abs}\ xs\ x\ X)\ [Y/y]_{ys}$ is expressible in terms of $X\,[Y/y]_{ys}$ only if (xs, x) is distinct from (ys, y) and fresh for Y. And similarly for parallel substitution. By contrast, swapping does not suffer from this restriction, which makes it significantly more manageable in proofs.

In addition to the simplification rules, we prove a comprehensive collection of lemmas describing the interaction between any pair of operators, including the interaction of each operator with itself (the latter being typically a form of compositionality property). Below we only list these properties for terms, omitting the corresponding ones for abstractions.

Proposition 16 (Properties of Swapping)**.** Assume $\mathsf{good}\ X$. The following hold:
(1) Swapping the same variable is identity:

$$X\,[x \wedge x]_{xs} = X$$

(2) Swapping is compositional:

$$(X\,[x_1 \wedge x_2]_{xs})\,[y_1 \wedge y_2]_{ys} \;=\; (X\,[y_1 \wedge y_2]_{ys})\,[(x_1\,[y_1 \wedge y_2]_{xs,ys}) \wedge (x_2\,[y_1 \wedge y_2]_{xs,ys})]_{xs}$$

(3) Swapping commutes if the variables are disjoint or the varsorts are different:

$$xs \neq ys \;\vee\; \{x_1, x_2\} \cap \{y_1, y_2\} = \emptyset \;\Longrightarrow\; (X\,[x_1 \wedge x_2]_{xs})\,[y_1 \wedge y_2]_{ys} = (X\,[y_1 \wedge y_2]_{ys})\,[x_1 \wedge x_2]_{xs}$$

(4) Swapping is involutive:

$$(X\,[x \wedge y]_{xs})\,[x \wedge y]_{xs} = X$$

(5) Swapping is symmetric:

$$X\,[x \wedge y]_{xs} = X\,[y \wedge x]_{xs}$$

Proposition 17 (Swapping versus Freshness)**.** Assume $\mathsf{good}\ X$. The following hold:
(1) Swapping preserves freshness:

$$xs \neq ys \;\vee\; x \notin \{y_1, y_2\} \;\Longrightarrow\; \mathsf{fresh}\ xs\ (x\,[y_1 \wedge y_2]_{xs,ys})\ (X\,[y_1 \wedge y_2]_{ys}) = \mathsf{fresh}\ xs\ x\ X$$

(2) Swapping fresh variables is identity:

$$\mathsf{fresh}\ xs\ x_1\ X \;\wedge\; \mathsf{fresh}\ xs\ x_2\ X \;\Longrightarrow\; X\,[x_1 \wedge x_2]_{xs} = X$$

(3) Swapping fresh variables composes:

$$\mathsf{fresh}\ xs\ y\ X \;\wedge\; \mathsf{fresh}\ xs\ z\ X \;\Longrightarrow\; (X\,[y \wedge x]_{xs})\,[z \wedge y]_{xs} = X\,[z \wedge x]_{xs}$$

The following lemmas describe the basic properties of substitution. The results for unary substitution follow routinely from those of parallel substitution. However, to support concrete formalizations it is useful to have both versions. Indeed, the majority of formalizations will only need unary substitution—and, in such cases, the user should not be bothered with having to work with the much heavier parallel substitution properties.

Proposition 18 (Swapping versus Substitution)**.** Assume $\mathsf{good}\ X$, $\mathsf{good}\ Y$ and $\uparrow\mathsf{good}\ \rho$. The following hold:
(1) $(X\,[\rho])\,[z_1 \wedge z_2]_{zs} = (X\,[z_1 \wedge z_2]_{zs})\,[\uparrow(\_\,[z_1 \wedge z_2]_{zs})\,\rho]$
(2) $Y\,[X/x]_{xs}\,[z_1 \wedge z_2]_{zs} \;=\; (Y\,[z_1 \wedge z_2]_{zs})\,[(X[z_1 \wedge z_2]_{zs})\,/\,(x[z_1 \wedge z_2]_{xs,zs})]_{xs}$

Note that, at point (1) above, $\uparrow(\_\,[z_1 \wedge z_2]_{zs})\ \rho$ is the lifting of the $(z_1, z_2.zs)$-swapping operator (which swaps z_1 with z_2 on varsort zs) to parallel-substitution environments ρ : *varsort* $\to$ *var* $\to$ *term option*. Point (2) follows easily from point (1).

Proposition 19 (Parallel Substitution versus Freshness)**.** Assume good X and $\uparrow$good ρ. The following hold:

$$\begin{array}{l}\mathsf{fresh}\ xs\ x\ (X\ [\rho]) \iff (\ \forall\ ys\ y.\ \mathsf{fresh}\ ys\ y\ X \vee \\ ((\rho\ ys\ y = \mathsf{None} \wedge (ys, y) \neq (xs, x))\ \vee\ (\exists\ Y.\ \rho\ ys\ y = \mathsf{Some}\ Y \wedge \mathsf{fresh}\ xs\ x\ Y))\ )\end{array}$$

In the unary case we obtain three separate properties:

Proposition 20 (Substitution versus Freshness)**.** Assume good X and good Y. The following hold:

(1) Freshness for a unary substitution decomposes into freshness for its participants:

$$\mathsf{fresh}\ zs\ z\ (X[Y/y]_{ys}) \iff ((zs, z) = (ys, y) \vee \mathsf{fresh}\ zs\ z\ X) \wedge (\mathsf{fresh}\ ys\ y\ X \vee \mathsf{fresh}\ zs\ z\ Y)$$

(2) Substitution preserves freshness:

$$\mathsf{fresh}\ zs\ z\ X \wedge \mathsf{fresh}\ zs\ z\ Y \implies \mathsf{fresh}\ zs\ z\ (X\ [Y/y]_{ys})$$

(3) The substituted variable is fresh for the substitution:

$$\mathsf{fresh}\ ys\ y\ Y \implies \mathsf{fresh}\ ys\ y\ (X\ [Y/y]_{ys})$$

Proposition 21 (Properties of Substitution)**.** Assume good X, good Y, $\uparrow$good ρ and $\uparrow$good ρ'. The following hold:

(1) Parallel substitution in environment ρ only depends on ρ's action on the free (non-fresh) variables:

$$(\forall\ ys\ y.\ \neg\ \mathsf{fresh}\ ys\ y\ X \implies \rho\ ys\ y = \rho'\ ys\ y) \implies X\ [\rho] = X\ [\rho']$$

(2) Parallel substitution is the identity if the free variables of the environment ρ are disjoint from those of the target term X:

$$(\forall\ zs\ z.\ \uparrow(\mathsf{fresh}\ zs\ z)\ \rho\ \vee\ \mathsf{fresh}\ zs\ z\ X) \implies X\ [\rho] = X$$

(3) Unary substitution is the identity if the substituted variable is fresh for the target term (corollary of point (2)):

$$\mathsf{fresh}\ ys\ y\ X \implies (X\ [Y/y]_{ys}) = X$$

As for compositionality of substitution we give different versions of the lemma, all of which are consequences of the first, most general one.

Proposition 22 (Substitution Compositionality)**.** Assume good X, good Y, $\uparrow$good ρ and $\uparrow$good ρ'. The following hold:

(1) Parallel substitution is compositional:

$$X\,[\rho]\,[\rho'] \ =\ X\,[\rho \bullet \rho']$$

where $\rho \bullet \rho'$ is the monadic composition of ρ and ρ', defined as

$$(\rho \bullet \rho')\ xs\ x = \ \mathsf{case}\ \rho\ xs\ x\ \mathsf{of}\ \mathsf{None} \Rightarrow \rho'\ xs\ x \mid \mathsf{Some}\ X \Rightarrow X[\rho']$$

(2) Parallel substitution distributes over unary substitution:

$$(X\,[Y/y]_{ys})\,[\rho] \ =\ X\,[\rho\,[y \leftarrow Y[\rho]]_{ys}]$$

where $\rho[y \leftarrow Y[\rho]]_{ys}$ is the assignment ρ updated with value $\mathsf{Some}\,(Y[\rho])$ for y
(3) Unary substitution composes with parallel substitution (via monadic composition)

$$(X[\rho])\,[Y/y]_{ys} = X[\rho \bullet [Y/y]_{ys}]$$

where we use the notation $[Y/y]_{ys}$ also for that environment that maps everything to None, but (ys, y) which is instead mapped to Y
(4) Substitution of the same variable (and of the same varsort) distributes over itself:

$$X\,[Y_1/y]_{ys}\,[Y_2/y]_{ys} = X\,[(Y_1\,[Y_2/y]_{ys})/y]_{ys}$$

(5) Substitution of different variables distributes over itself, assuming freshness:

$$(ys \neq zs \vee y \neq z) \wedge \mathsf{fresh}\; ys\; y\; Z \implies X\,[Y/y]_{ys}\,[Z/z]_{zs} = (X\,[Z/z]_{zs})\;[(Y\,[Z/z]_{zs})/y]_{ys}$$

In summary, we have formalized quite exhaustively the general-purpose properties of the syntactic constructors and the standard operators. Some of these properties are subtle. During the formalization of concrete results for particular syntaxes, they are likely to require a lot of time to even formulate them correctly, let alone prove them—which would be wasteful, since they are independent of the particular syntax.

3.3 Operator-Sensitive Recursion

In this section we present several definition principles for functions having terms and abstractions as their domains. The principles we formalize are generalizations to an arbitrary syntax of results that have been previously described for the particular syntax of λ-calculus [63, 52]. The main characteristic of the principles will be that the functions they introduce have defining clauses not only for the constructors (as customary in recursive definitions on free datatypes), but also for the freshness, substitution and/or swapping operators.

In this section there is our very first approach to recursion in the presence of bindings. In the context of this theory we have formalized definition principles in a style that we have been able to reuse in later formalizations; in particular in Section 6.5 of Chapter 6 we have transported the same in our most recent and general framework, based on functors.

We start with the simpler-structured *iteration* principles (3.3.1) followed by their extension to primitive recursion (3.3.2). We also show two examples of using our principles. The first defines the skeleton of a term (a generalization of the notion of depth) using freshness-swapping-based iteration (3.3.3). The second employs freshness-substitution-based iteration to produce a whole class of instances: the interpretation of syntax in semantic domains (3.3.4).

3.3.1 Iteration

A *freshness-substitution (FSb) model* consists of two collections of elements endowed with term- and abstraction- like operators satisfying some characteristic properties of terms. More precisely, it consists of:

- two types, T and A
- operations corresponding to the constructors:

F1: $(ys, y) \neq (xs, x) \implies \mathsf{FRESH}\ ys\ y\ (\mathsf{VAR}\ xs\ x)$
F2: $\uparrow(\mathsf{FRESH}\ ys\ y)\ inp$ and $\uparrow(\mathsf{FRESHABS}\ ys\ y)\ binp \implies \mathsf{FRESH}\ ys\ y\ (\mathsf{OP}\ \delta\ inp\ binp)$
F3: $\mathsf{FRESHABS}\ ys\ y\ (\mathsf{ABS}\ ys\ y\ X)$
F4: $\mathsf{FRESH}\ ys\ y\ X \implies \mathsf{FRESHABS}\ ys\ y\ (\mathsf{ABS}\ xs\ x\ X)$

FIGURE 3.3: The freshness clauses

Sb1: $\mathsf{SUBST}\ (\mathsf{VAR}\ zs\ z)\ Z\ z\ zs = Z$
Sb2: $(xs, x) \neq (zs, z) \implies \mathsf{SUBST}\ (\mathsf{VAR}\ xs\ x)\ Z\ z\ zs = \mathsf{VAR}\ xs\ x$
Sb3: $\mathsf{SUBST}\ (\mathsf{OP}\ \delta\ inp\ binp)\ Z\ z\ zs = \mathsf{OP}\ \delta\ (\uparrow(\mathsf{SUBST}\ \_\ Z\ z\ zs)\ inp)\ (\uparrow(\mathsf{SUBSTABS}\ \_\ Z\ z\ zs)\ binp)$
Sb4: $(xs, x) \neq (zs, z) \wedge \mathsf{FRESH}\ xs\ x\ Z \implies \mathsf{SUBSTABS}\ (\mathsf{ABS}\ xs\ x\ X)\ Z\ z\ zs = \mathsf{ABS}\ xs\ x\ (\mathsf{SUBST}\ X\ Z\ z\ zs)$
SbRn: $\mathsf{FRESH}\ xs\ y\ X \implies \mathsf{ABS}\ xs\ x\ X = \mathsf{ABS}\ xs\ y\ (\mathsf{SUBST}\ (\mathsf{VAR}\ xs\ y)\ x\ xs\ X)$

FIGURE 3.4: The substitution and substitution-renaming clauses

$$\mathsf{VAR} : varsort \to var \to T$$

$$\mathsf{OP} : opsym \to (index, T)\ input \to (bindex, A)\ input \to T$$

$$\mathsf{ABS} : varsort \to var \to T \to A$$

- operations corresponding to freshness and substitution:

$$\mathsf{FRESH} : varsort \to var \to T \to bool$$

$$\mathsf{FRESHABS} : varsort \to var \to A \to bool$$

$$\mathsf{SUBST} : T \to T \to var \to varsort \to T$$

$$\mathsf{SUBSTABS} : A \to T \to var \to varsort \to A$$

and it is required to satisfy:

- the freshness clauses F1–F5 shown in Figure 3.3 (analogous to the implicational simplification rules for freshness in Prop. 11)
- substitution clauses Sb1–Sb4 shown in Figure 3.4 (analogous to the simplification rules for substitution in Prop. 14)
- the substitution-renaming clause SbRn also shown in Figure 3.4 (analogous to the substitution-based abstraction equality rule in Prop. 10(2))

Theorem 23. The good terms and abstractions form the *initial FSb model*. Namely, for any FSb model as above, there exist the functions $f : term \to T$ and $fAbs : abs \to A$ that commute, on good terms, with the constructors and with substitution and preserve freshness:

$$f\ (\mathsf{Var}\ xs\ x) = \mathsf{VAR}\ xs\ x$$
$$f\ (\mathsf{Op}\ \delta\ inp\ binp) = \mathsf{OP}\ \delta\ (\uparrow f\ inp)\ (\uparrow fAbs\ binp)$$
$$fAbs\ (\mathsf{Abs}\ xs\ x\ X) = \mathsf{ABS}\ xs\ x\ (f\ X)$$
$$f\ (X\,[Y/y]_{ys}) = \mathsf{SUBST}\ (f\ X)\ (f\ Y)\ y\ ys$$
$$fAbs\ (A\,[Y/y]_{ys}) = \mathsf{SUBSTABS}\ (fAbs\ A)\ (f\ Y)\ y\ ys$$
$$\mathsf{fresh}\ xs\ x\ X \implies \mathsf{FRESH}\ xs\ x\ (f\ X)$$
$$\mathsf{freshAbs}\ xs\ x\ A \implies \mathsf{FRESHABS}\ xs\ x\ (fAbs\ A)$$

Sw1: $\mathsf{SWAP}\ (\mathsf{VAR}\ xs\ x)\ z_1\ z_2\ zs\ =\ \mathsf{VAR}\ xs\ (x\,[z_1 \wedge z_2]_{xs,zs})$
Sw2: $\mathsf{SWAP}\ (\mathsf{OP}\ \delta\ inp\ binp)\ z_1\ z_2\ zs\ =\ \mathsf{OP}\ \delta\ (\uparrow(\mathsf{SWAP}\ \_\ z_1\ z_2\ zs)\ inp)\ (\uparrow(\mathsf{SWAPABS}\ \_\ z_1\ z_2\ zs)\ binp)$
Sw3: $\mathsf{SWAPABS}\ (\mathsf{ABS}\ xs\ x\ X)\ z_1\ z_2\ zs\ =\ \mathsf{ABS}\ xs\ (x\,[z_1 \wedge z_2]_{xs,zs})\ (\mathsf{SWAP}\ X\ z_1\ z_2\ zs)$
SwCong: $\mathsf{FRESH}\ xs\ y\ X\ \wedge\ \mathsf{FRESH}\ xs\ y\ X'\ \wedge\ \mathsf{SWAP}\ X\ y\ x\ ys = \mathsf{SWAP}\ X'\ y\ x'\ ys$
$\Longrightarrow\ \mathsf{ABS}\ xs\ x\ X = \mathsf{ABS}\ xs\ x'\ X'$

FIGURE 3.5: The swapping and swapping-based congruence clauses

In addition, the two functions are uniquely determined on good terms and abstractions, in that, for all other functions $g : term \to T$ and $gAbs : abs \to A$ satisfying the same commutation and preservation properties, it holds that f and g are equal on good terms and $fAbs$ and $gAbs$ are equal on good abstractions.

Like any initiality property, this theorem represents an iteration principle. To comprehend the connection between initiality and iteration, let us first look at the simpler case of lists over a type G, with constructors $\mathsf{Nil} : G\ list$ and $\mathsf{Cons} : G \to G\ list \to G\ list$. To define, by iteration, a function from lists, say, $\mathsf{length} : G\ list \to nat$, we need to indicate what is Nil mapped to, here $\mathsf{length}\ \mathsf{Nil} = 0$, and, recursively, what is Cons mapped to, here $\mathsf{length}\ (\mathsf{Cons}\ a\ as) = 1 + \mathsf{length}\ as$. We can rephrase this by saying: If we define "list-like" operators on the target domain— here, taking $\mathsf{NIL} : nat$ to be 0 and $\mathsf{CONS} : G \to nat \to nat$ to be $\lambda g, n.\ 1 + n$—then the iteration offers us a function length that commutes with the constructors: $\mathsf{length}\ \mathsf{Nil} = \mathsf{NIL} = 0$ and $\mathsf{length}\ (\mathsf{Cons}\ a\ as) = \mathsf{CONS}\ a\ (\mathsf{length}\ as) = 1 + \mathsf{length}\ as$. For terms, we have a similar situation, except that (1) substitution and freshness are considered in addition to the constructors and (2) paying the price for lack of freeness, some conditions need to be verified to deem the operations "term-like."

The main feature of our iteration theorem is the ability to define functions in a manner that is compatible with alpha-equivalence. A byproduct of the theorem is that the defined functions also interact well with freshness and substitution, in that they map these concepts to corresponding concepts on the target domains.

Michael Norrish has developed a similar principle that employs swapping instead of substitution [52]. We have also formalized this in our framework—in a slightly restricted form, namely without factoring in fixed variables and parameters.

A *freshness-swapping (FSw) model* is a structure similar to a freshness-substitution model, just that instead of the substitution-like operators, SUBST and SUBSTABS, it features swapping-like operators:

$$\mathsf{SWAP} : T \to var \to var \to varsort \to T$$

$$\mathsf{SWAPABS} : A \to var \to var \to varsort \to A$$

assumed to satisfy the clauses Sw1–Sw3 corresponding to those for simplifying term swapping and, instead of the substitution-based variable-renaming property, a swapping-based congruence rule for abstractions SwCong—all shown in Figure 3.5.

Then a swapping-aware version of the iteration theorem holds:

Theorem 24. The good terms and abstractions form the initial FSw model. Namely, there exists a pair of functions $f : term \to T$ and $fAbs : abs \to A$ that commute, on good terms, with

the constructors and preserves freshness—similarly to how it is described in Theorem 23, the only difference being that they are not guaranteed to commute with substitution, but with swapping, namely:

$$f\,(X\,[z_1 \wedge z_2]_{zs}) = \mathsf{SWAP}\ (f\ X)\ z_1\ z_2\ zs$$
$$fAbs\,(A\,[z_1 \wedge z_2]_{zs}) = \mathsf{SWAPABS}\ (fAbs\ A)\ z_1\ z_2\ zs$$

In addition, the two functions are uniquely determined on good terms and abstractions (just like in Theorem 23).

Finally, we combine both notions, obtaining *freshness-substitution-swapping (FSbSw) models*. These are required to have both substitution-like and swapping-like operators and to satisfy the union of the FSb and FSw clauses, except for the swapping congruence clause SwCong—namely, clauses F1-F4, Sb1-Sb4, Sw1-Sw3 and SbRn. (Interestingly, SwCong was not needed for proving the iteration theorem; the proof needs either SbRn and SwCong, i.e., only one of the two.)

Theorem 25. The good terms and abstractions form the *initial* FSbSw model. Namely, there exists a pair of functions $f : term \to T$ and $fAbs : abs \to A$ that commute, on good terms, with the constructors, substitution, swapping and preserves freshness, as described in Theorems 23 and 24. In addition, the two functions are uniquely determined on good terms and abstractions.

In summary, we have three variants of models (and iteration principles), corresponding to three combinations of the fundamental term operations:

- freshness-substitution (FSb) models
- freshness-swapping (FSw) models
- freshness-substitution-swapping (FSbSw) models

Having formalized all these variants, the user can decide on the desired "contract:" With more operators factored in, there are more proof obligations that need to be discharged for the definition to succeed, but then the defined functions satisfy more desirable properties.

3.3.2 Primitive Recursion

Iteration is a simplified form of primitive recursion. The difference between the two is illustrated by the following simple example:[1] The predecessor function $\mathsf{pred} : nat \to nat$ is defined by $\mathsf{pred}\ 0 = 0$ and $\mathsf{pred}\,(\mathsf{Suc}\ n) = n$. This does not fit an iteration scheme, where only the value of the function on smaller arguments, and not the arguments themselves, can be used. In the example, iteration would allow $\mathsf{pred}\,(\mathsf{Suc}\ n)$ to invoke recursively $\mathsf{pred}\ n$, but not n. Of course, we can simulate recursion by iteration if we are allowed an auxiliary output: defining $\mathsf{pred}' : nat \to nat \times nat$ by iteration, $\mathsf{pred}'\ 0 = (0,0)$ and $\mathsf{pred}'\,(\mathsf{Suc}\ n) = \mathsf{case}\ \mathsf{pred}'\ n$ of $(n_1,n_2) \Rightarrow (\mathsf{Suc}\ n_1, n_1)$, and then taking $\mathsf{pred}\ n$ to be the second component of $\mathsf{pred}'\ n$.

In our framework, primitive recursion can also be reduced to iteration—see [62, §1.4.2] for a description of this phenomenon for the general case of initial models in Horn theories.

[1] This is a contrived example, where no "real" recursion occurs—but it illustrates the point.

Initially, we had only formalized the iteration theorems. However, we soon realized that several applications (for the particular syntaxes of λ-calculus and many-sorted first-order logic) required the full power of primitive recursion, and it was very tedious to perform the recursion-to-iteration encoding over and over again, with each new definition. We therefore decided to formalize this reduction for an arbitrary syntax, obtaining primitive recursion theorems in all three variants, that is, factoring in substitution, swapping or both. We only show here the primitive recursion variant of Theorem 23, where we highlight the additions compared to iteration. (The other two primitive recursion theorems are obtained similarly from Theorems 24 and 25.)

A *FSb recursion model* has the same components as an FSb model, except that:

- OP takes term and abstraction inputs in addition to inputs from the model, i.e., has type $opsym \to (index, term)\ input \to (index, T)\ input \to (bindex, abs)\ input \to (bindex, A)\ input \to T$
- ABS takes an additional term argument, i.e., has type $varsort \to var \to term \to T \to A$
- The freshness and substitution operators take additional term and/or abstraction arguments; e.g., the types for the term versions of these are:

  $$\mathsf{FRESH} : varsort \to var \to term \to T \to bool$$

  $$\mathsf{SUBST} : term \to T \to term \to T \to var \to varsort \to T$$

- The clauses F1–F4, Sb1–Sb4 and SbRn are updated to factor in the additional structure, e.g., Sb4 becomes:

  $$(xs, x) \neq (zs, z) \wedge \mathsf{fresh}\ xs\ x\ Z' \wedge \mathsf{FRESH}\ xs\ x\ Z'\ Z \Longrightarrow$$
  $$\mathsf{SUBSTABS}\ (\mathsf{Abs}\ xs\ x\ X')\ (\mathsf{ABS}\ xs\ x\ X'\ X)\ \ Z'\ Z\ z\ zs\ =$$
  $$\mathsf{ABS}\ xs\ x\ (X'[Z'/z]_{zs})\ (\mathsf{SUBST}\ X'\ X\ Z'\ Z\ z\ zs)$$

  where X and Z are (as before) elements of T, whereas X' and Z' are terms.

Theorem 26. For any FSb recursion model as above, there exist the functions $f : term \to T$ and $fAbs : abs \to A$ that commute, on good terms, with the constructors and with substitution and preserve freshness, in the same manner as in Theorem 23, *mutatis mutandis*. For example:

- $f\ (\mathsf{Var}\ xs\ x) = \mathsf{VAR}\ xs\ x$
- $f\ (\mathsf{Op}\ \delta\ inp\ binp) = \mathsf{OP}\ \delta\ inp\ (\uparrow f\ inp)\ binp\ (\uparrow fAbs\ binp)$
- $\mathsf{fresh}\ xs\ x\ X \Longrightarrow \mathsf{FRESH}\ xs\ x\ X\ (f\ X)$
- $f\ (X[Y/y]_{ys}) = \mathsf{SUBST}\ X\ (f\ X)\ Y\ (f\ Y)\ y\ ys$

3.3.3 Iteration Example: the Skeleton of a Term

Since terms are possibly infinitely branching, they have no notion of finite depth. While we could generalize the depth to return a transfinite ordinal, we opt for a simpler solution: Instead of depth, we use a slightly more informative entity, the "skeleton," which models a term's bare-bones structure.

We define the (free) datatypes of trees and "abstraction trees" branching over the free and bound indexes we use for terms. Unlike terms and abstractions, these store no operation symbols or variables, but only placeholders indicating their presence.

$$\begin{array}{rrcl} \mathsf{datatype} & \mathit{tree} & = & \mathsf{tVar} \mid \\ & & & \mathsf{tOp}\ ((\mathit{index},\mathit{tree})\,\mathit{input})\ ((\mathit{bindex},\mathit{atree})\,\mathit{input}) \\ \mathsf{and} & \mathit{atree} & = & \mathsf{tAbs}\ \mathit{tree} \end{array}$$

Our aim is to introduce the *skeleton* of a term (or of an abstraction), as the tree obtained from it by retaining only branching information and forgetting about the occurrences of operation symbols and variables and their sorts. Namely, we wish to define $\mathsf{skel} : \mathit{term} \to (\mathit{index},\mathit{bindex})\,\mathit{tree}$ and $\mathsf{skelAbs} : \mathit{abs} \to (\mathit{index},\mathit{bindex})\,\mathit{tree}$ by the following mutually recursive clauses:

$$\begin{array}{rcl} \mathsf{skel}\ (\mathsf{Var}\ \mathit{xs}\ x) & = & \mathsf{tVar} \\ \mathsf{skel}\ (\mathsf{Op}\ \delta\ \mathit{inp}\ \mathit{binp}) & = & \mathsf{tOp}\ (\uparrow\mathsf{skel}\ \mathit{inp})\ (\uparrow\mathsf{skelAbs}\ \mathit{binp}) \\ \mathsf{skelAbs}\ (\mathsf{Abs}\ \mathit{xs}\ x\ X) & = & \mathsf{tAbs}\ (\mathsf{skel}\ X) \end{array}$$

To this end, we wish to make use of one of our iteration/recursion principles to guarantee that the above represents a valid definition, in that there exist the functions skel and skelAbs satisfying the above equations. So we look into "completing" the above definition by indicating how these presumptive functions are supposed to behave with respect to the standard operators. In other words, we try to define tree versions of the standard term operators

- $\mathsf{FRESH} : \mathit{varsort} \to \mathit{var} \to \mathit{tree} \to \mathit{bool}$
  $\mathsf{FRESHABS} : \mathit{varsort} \to \mathit{var} \to \mathit{atree} \to \mathit{bool}$
- $\mathsf{SWAP} : \mathit{tree} \to \mathit{var} \to \mathit{var} \to \mathit{varsort} \to \mathit{tree}$
  $\mathsf{SWAPABS} : \mathit{atree} \to \mathit{var} \to \mathit{var} \to \mathit{varsort} \to \mathit{atree}$
  and/or
- $\mathsf{SUBST} : \mathit{tree} \to \mathit{tree} \to \mathit{var} \to \mathit{varsort} \to \mathit{tree}$
  $\mathsf{SUBSTABS} : \mathit{atree} \to \mathit{tree} \to \mathit{var} \to \mathit{varsort} \to \mathit{atree}$

while keeping in mind that skel and skelAbs must commute with these. Since trees have no actual variables in them, the only sensible choices are the trivial ones:

- $\mathsf{FRESH}\ \mathit{xs}\ x\ X = \mathsf{True},\ \mathsf{FRESHABS}\ \mathit{xs}\ x\ A = \mathsf{True}$
- $\mathsf{SWAP}\ X\ x_1\ x_2\ \mathit{xs} = X,\ \mathsf{SWAPABS}\ A\ x_1\ x_2\ \mathit{xs} = A$
- $\mathsf{SUBST}\ Y\ X\ x\ \mathit{xs} = Y,\ \mathsf{SUBSTABS}\ A\ X\ x\ \mathit{xs} = A$

Thus, commutation with the operators will mean the following (where we omit the properties for the abstraction versions of the operators, which are similar):

$$\begin{array}{rcl} \mathsf{fresh}\ \mathit{xs}\ x\ X & \Longrightarrow & \mathsf{True} \\ \mathsf{skel}\ (X\,[x_1 \wedge x_2]_{\mathit{xs}}) & = & \mathsf{skel}\ X \\ \mathsf{skel}\ (Y\,[X/x]_{\mathit{xs}}) & = & \mathsf{skel}\ Y \end{array}$$

Of these, the intended freshness and swapping properties are clearly suitable: The former is vacuously true, and the latter states that the skeleton does not change after swapping two

variables. However, the substitution property cannot work, since it states the wrong/undesired property that the skeleton of a term Y does not change after substituting a term X for one of its variables x—which contradicts our intuition that the skeleton may in fact grow (specifically, at the tVar leaves that correspond to free occurrences of Var xs x in Y).

In summary, for making the skeleton definition work we must focus on freshness and swapping rather than substitution. We thus employ the iteration Theorem 24, where the required FSw model is defined taking FRESH, FRESHABS, SWAP and SWAPABS as above and taking VAR, OP, ABS to be given by tVar, tOp and tAbs, respectively. (Namely, VAR xs x = tVar, OP δ = tOp and ABS xs x = tAbs.) That this indeed forms an FSw model, i.e., satisfies the desired clauses, is immediate to check: F1–F4 hold trivially since FRESH and FRESHABS are vacuously true, while Sw1–Sw4 and SwCong hold trivially since SWAP and SWAPABS are the identity functions.

Thus, Theorem 24 gives us the functions skel and skelAbs that are uniquely characterized by the following properties (where we omit the freshness preservation property, which in this case is just a tautology):

$$\begin{aligned}
\mathsf{skel}\ (\mathsf{Var}\ xs\ x) &= \mathsf{tVar} \\
\mathsf{skel}\ (\mathsf{Op}\ \delta\ inp\ binp) &= \mathsf{tOp}\ (\uparrow\mathsf{skel}\ inp)\ (\uparrow\mathsf{skelAbs}\ binp) \\
\mathsf{skelAbs}\ (\mathsf{Abs}\ xs\ x\ X) &= \mathsf{tAbs}\ (\mathsf{skel}\ X) \\
\mathsf{skel}\ (X\,[x_1 \wedge x_2]_{xs}) &= \mathsf{skel}\ X \\
\mathsf{skelAbs}\ (A\,[x_1 \wedge x_2]_{xs}) &= \mathsf{skelAbs}\ A
\end{aligned}$$

Besides offering a simple instance of freshness-swapping-based iteration, the skeleton operator provides a generalization of depth that turned out to be sufficient for proving important properties requiring renaming variables in terms—notably the fresh induction principle we discuss in Section 3.4.

3.3.4 Interpretation of Syntax in Semantic Domains

Perhaps the most useful application of our iteration principles is the seamless interpretation of syntax in semantic domains, in a manner that is guaranteed to be compatible with alpha, substitution and freshness. This construction shows up commonly in the literature, for different notions of semantic domain. However, the construction is essentially the same, and can be expressed for an arbitrary syntax—for which reason we have formalized it in our framework.

A *semantic domain* consists of two collections of elements endowed with interpretations of the Op and Abs constructors, the latter in a higher-order fashion—interpreting variable binding as (meta-level) functional binding. Namely, it consists of:

- two types, Dt and Da
- a function op : $opsym \to (index, Dt)\ input \to (bindex, Da)\ input \to Dt$
- a function abs : $varsort \to (Dt \to Dt) \to Da$

Theorem 27. The terms and abstractions are interpretable in any semantic domain. Namely, if val is the type of valuations of variables in the domain, $varsort \to var \to Dt$, there exist the functions sem : $term \to val \to Dt$ and semAbs : $abs \to val \to Da$ such that:

- sem (Var xs x) $\rho = \rho\ xs\ x$

- $\mathsf{sem}\,(\mathsf{Op}\,\delta\;inp\;binp)\,\rho = \mathsf{op}\,\delta\;(\uparrow(\mathsf{sem}\;\_\;\rho)\;inp)\;(\uparrow(\mathsf{semAbs}\;\_\;\rho)\;binp)$
- $\mathsf{semAbs}\,(\mathsf{Abs}\;xs\;x\;X)\,\rho = \mathsf{abs}\;xs\,(\lambda d.\;\mathsf{sem}\;X\;(\rho[(xs,x) \leftarrow d]))$,
  where $\rho[(xs,x) \leftarrow d]$ is the function ρ updated at (xs,x) with d—which sends (xs,x) to d and any other (ys,y) to $\rho\;ys\;y$.

In addition, the interpretation functions map syntactic substitution and freshness to semantic versions of the concepts:

- $\mathsf{sem}\,(X[Y/y]_{ys})\,\rho = \mathsf{sem}\;X\;(\rho[(ys,y) \leftarrow \mathsf{sem}\;Y\;\rho])$
- $\mathsf{fresh}\;xs\;x\;X \implies (\forall \rho,\rho'.\;\rho =_{(xs,x)} \rho' \implies \mathsf{sem}\;X\;\rho = \mathsf{sem}\;X\;\rho')$,
  where "$=_{(xs,x)}$" means "equal everywhere except perhaps on (xs,x)"—namely $\rho =_{(xs,x)} \rho'$ holds iff $\rho\;ys\;y = \rho'\;ys\;y$ for all $(ys,y) \neq (xs,x)$.

Theorem 27 is the foundation for many particular semantic interpretations, including that of λ-terms in Henkin models and that of FOL terms and formulas in FOL models. It guarantees compatibility with alpha and proves, as bonuses, a freshness and a substitution property. The freshness property is the familiar notion that the interpretation only depends on the free variables, and the substitution property generalizes what is usually called *the substitution lemma*, stating that interpreting a substituted term is the same as interpreting the original term in a "substituted" environment. Both properties are essential lemmas in most developments that involve semantics.

This theorem follows by an instantiation of the iteration Theorem 23: taking T and A to be $val \to Dt$ and $val \to Da$ and taking the term/abstraction-like operations as prescribed by the desired clauses for sem and semAbs (in the following we omit the abstraction versions of the freshness and substitution operators):

- $\mathsf{VAR}\;xs\;x \;=\; \lambda\rho.\;\rho\;xs\;x$
- $\mathsf{OP}\;\delta\;inp\;binp \;=\; \lambda\rho.\;\mathsf{op}\;\delta\;(\uparrow(\_\;\rho)\;inp)\;(\uparrow(\_\;\rho)\;binp)$
  where $(\_\;\rho)$ denotes the "application to ρ" operator $\lambda\,u.\,u\,\rho$
- $\mathsf{ABS}\;xs\;x\;X \;=\; \lambda\rho.\;\mathsf{abs}\;xs\;(\lambda d.\;X\;(\rho[(xs,x) \leftarrow d]))$
- $\mathsf{FRESH}\;xs\;x\;X \;=\; (\forall \rho,\rho'.\;\rho =_{(xs,x)} \rho' \implies X\,\rho = X\,\rho')$
- $\mathsf{SUBST}\;X\;Y\;y\;ys = \;\lambda\rho.\;X\;(\rho[(ys,y) \leftarrow Y\,\rho])$

Note that the above definitions are *completely determined* by the intended properties listed in Theorem 27 (which we set out to prove). For example, FRESH was defined so that the freshness property listed in Theorem 27 becomes the freshness commutation property $\mathsf{fresh}\;xs\;x\;X \implies \mathsf{FRESH}\;xs\;x\;(\mathsf{sem}\;X)$. Thus, according to our freshness-substitution-based iteration Theorem 23, what we are left to check in order to prove Theorem 27 is that the above structure is an FSb model, i.e., satisfies the clauses F1-F4, Sb1-Sb4 and SbRn. This amounts to checking the following:

F1: $(xs,x) \neq (ys,y) \;\wedge\; \rho =_{(ys,y)} \rho' \implies \rho\;xs\;x = \rho'\;xs\;x$

F2: A trivial implication, of the form "A implies A"

F3: If $\rho =_{(ys,y)} \rho' \implies \rho[(ys,y) \leftarrow d] = \rho'[(ys,y) \leftarrow d]$

F4: If $\rho =_{(ys,y)} \rho' \implies \rho[(xs,x) \leftarrow d] =_{(ys,y)} \rho'[(xs,x) \leftarrow d]$

Sb1: $\rho[(xs,x) \leftarrow d]\ xs\ x = d$

Sb2: If $(xs,x) \neq (ys,y) \implies \rho[(ys,y) \leftarrow d]\ xs\ x = \rho\ xs\ x$

Sb3: Some trivial equalities, of the form "A = A"

Sb4: $\rho =_{(xs,x)} \rho[(xs,x) \leftarrow d]$ and
$(xs,x) \neq (zs,z) \implies \rho[(zs,z) \leftarrow d'][(xs,x) \leftarrow d] = \rho[(xs,x) \leftarrow d]\,[(zs,z) \leftarrow d']$

SbRn: $\rho[(xs,y) \leftarrow d]\,[(xs,x) \leftarrow d] =_{(xs,y)} \rho[(xs,x) \leftarrow d]$

All the above are straightforward properties of function update. Indeed, Isabelle/HOL's *auto* method was able to prove all of them.

3.4 Induction Principle

We formalize a scheme for "fresh" induction in the style of Nominal Logic, which realizes the Barendregt convention. We introduce and motivate this scheme by an example. To prove Prop. 20(a), we use (mutual) structural induction over terms and abstractions, proving the statement together with the corresponding statement for abstractions,

$$\begin{aligned}&\mathsf{freshAbs}\ zs\ z\ (A[Y/y]_{ys}) \iff\\ &((zs,z) = (ys,y) \vee \mathsf{freshAbs}\ zs\ z\ A) \wedge (\mathsf{freshAbs}\ ys\ y\ A \vee \mathsf{fresh}\ zs\ z\ Y)\end{aligned}$$

The proof's only interesting case is the Abs case, say, for abstractions of the form $\mathsf{Abs}\ xs\ x\ X$. However, if we were able to assume freshness of (xs,x) for all the statement's parameters, namely Y, (ys,y) and (zs,z), this case would also become "uninteresting," following automatically from the induction hypothesis by mere simplification, as shown below (with the freshness assumptions highlighted):

$\mathsf{freshAbs}\ zs\ z\ ((\mathsf{Abs}\ xs\ x\ X)\,[Y/y]_{ys})$
$\Updownarrow$ (by Prop. 14(3), since $(xs,x) \neq (ys,y)$ and $\mathsf{fresh}\ xs\ x\ Y$)
$\mathsf{freshAbs}\ zs\ z\ (\mathsf{Abs}\ xs\ x\ (X\,[Y/y]_{ys}))$
$\Updownarrow$ (by Prop. 12(3), since $(xs,x) \neq (zs,z)$)
$\mathsf{fresh}\ zs\ z\ (X\,[Y/y]_{ys})$
$\Updownarrow$ (by Induction Hypothesis)
$((zs,z) = (ys,y) \vee \mathsf{fresh}\ zs\ z\ X) \wedge (\mathsf{fresh}\ ys\ y\ X \vee \mathsf{fresh}\ zs\ z\ Y)$
$\Updownarrow$ (by Prop. 12(3) applied twice, since $(xs,x) \neq (zs,z)$ and $(xs,x) \neq (ys,y)$)
$((zs,z) = (ys,y) \vee \mathsf{freshAbs}\ zs\ z\ (\mathsf{Abs}\ xs\ x\ X)) \wedge (\mathsf{freshAbs}\ ys\ y\ (\mathsf{Abs}\ xs\ x\ X) \vee \mathsf{fresh}\ zs\ z\ Y)$

The practice of assuming freshness, known in the literature as the Barendregt convention, is a hallmark in informal reasoning about bindings. Thanks to insight from Nominal Logic [58, 76, 73], we also know how to apply this morally correct convention fully rigorously. To capture it in our formalization, we model parameters $p : \mathit{param}$ as anything that allows for a notion of freshness, or, alternatively, provides a set of (free) variables for each varsort, $\mathsf{varsOf} : \mathit{param} \to \mathit{varsort} \to \mathit{var\ set}$. With this, a "fresh induction" principle can be formulated, if all parameters have fewer variables than $|\mathit{var}|$ (in particular, if they have only finitely many).

Theorem 28. Let $\varphi : term \to param \to bool$ and $\varphi Abs : abs \to param \to bool$. Assume:
(1) $\forall xs, p.\ |\mathsf{varsOf}\ p\ xs| < |var|$
(2) $\forall xs, x, p.\ \varphi\ (\mathsf{Var}\ xs\ x)\ p$
(3) $\forall \delta, inp, binp, p.\ |\mathsf{dom}\ inp| < |var| \wedge |\mathsf{dom}\ binp| < |var| \wedge \uparrow(\lambda X.\ \mathsf{good}\ X \wedge (\forall q.\varphi\ X\ q))\ inp \wedge \uparrow(\lambda A.\ \mathsf{goodAbs}\ A \wedge (\forall q.\varphi Abs\ A\ q))\ binp \Longrightarrow \varphi\,(\mathsf{Op}\ \delta\ inp\ binp)\ p$
(4) $\forall xs, x, X, p.\ \mathsf{good}\ X \wedge \varphi\ X\ p \wedge x \notin \mathsf{varsOf}\ p\ xs \Longrightarrow \varphi Abs\,(\mathsf{Abs}\ xs\ x\ X)\ p$
Then $\forall X, p.\ \mathsf{good}\ X \Longrightarrow \varphi\ X\ p$ and $\forall A, p.\ \mathsf{goodAbs}\ A \Longrightarrow \varphi Abs\ A\ p$.

Highlighted is the essential difference from the usual structural induction: The bound variable x can be assumed fresh for the parameter p (on its varsort, xs). Note also that, in the Op case, we lift to inputs the predicate as quantified universally over all parameters.

Back to Prop. 20(a), this follows automatically by fresh induction (plus the shown simplifications), after recognizing as parameters the variables (ys, y) and (zs, z) and the term Y—formally, taking $param = (varsort \times var)^2 \times term$ and $\mathsf{varsOf}\ ((ys, y), (zs, z), Y)\ xs = \{y \mid xs = ys\} \cup \{z \mid xs = zs\} \cup \{x \mid \neg\,\mathsf{fresh}\ xs\ x\ Y\}$.

Fresh induction is based on the possibility to rename bound variables in abstractions without loss of generality. To prove this principle, we employed standard induction over the skeleton of terms—using the crucial fact that the skeleton is invariant under swapping.

3.5 Sorting the Terms

So far, we have a framework where the operations take as free and bound inputs partial families of terms and abstractions. All theorems refer to good (i.e., sufficiently low-branching) terms and abstractions. However, we promised a theory that is applicable to terms over many-sorted binding signatures. Thanks to the choice of a flexible notion of input, it is not difficult to cast our results into such a many-sorted setting. Given a suitable notion of signature (3.5.1), we classify terms according to sorts (3.5.2) and prove that well-sorted terms are good (3.5.3)—this gives us sorted versions of all theorems (3.5.5).

3.5.1 Binding Signatures

A *(binding) signature* is a tuple $(index, bindex, varsort, sort, opsym, \mathsf{asSort}, \mathsf{stOf}, \mathsf{arOf}, \mathsf{barOf})$, where *index*, *bindex*, *varsort* and *opsym* are types (with the previously discussed intuitions) and *sort* is a new type, of sorts for terms. Moreover:

- $\mathsf{asSort} : varsort \to sort$ is an injective map, embedding varsorts into sorts
- $\mathsf{stOf} : opsym \to sort$, read "the (result) sort of"
- $\mathsf{arOf} : opsym \to (index, sort)\ input$, read "the (free) arity of"
- $\mathsf{barOf} : opsym \to (bindex, varsort \times sort)\ input$, read "the bound arity of"

Thus, a signature prescribes which varsorts correspond to which sorts (as discussed in Section 3.1.4) and, for each operation symbol, which are the sorts of its free inputs (the arity), of its bound (abstraction) inputs (the bound arity), and of its result.

When we give examples for our concrete syntaxes described in Section 3.1, we will write $(i_1 \mapsto a_1, \ldots, i_n \mapsto a_n)$ for the partial function that sends each i_k to Some a_k and everything else to None. In particular, $()$ denotes the totally undefined function.

For the λ-calculus syntax, we take $\mathit{index} = \mathit{bindex} = \mathit{nat}$, $\mathit{varsort} = \mathit{sort} = \{\mathsf{lam}\}$ (a singleton datatype), $\mathit{opsym} = \{\mathsf{App}, \mathsf{Lam}\}$, asSort to be the identity and stOf to be the unique function to $\{\mathsf{lam}\}$. Since App has two free inputs and no bound input, we use the first two elements of *nat* as free arity and nothing for the bound arity: $\mathsf{arOf\ App} = (0 \mapsto \mathsf{lam}, 1 \mapsto \mathsf{lam})$, $\mathsf{barOf\ App} = ()$. By contrast, since Lam has no free input and one bound input, we use nothing for the free arity, and the first element of *nat* for the bound arity: $\mathsf{arOf\ Lam} = ()$, $\mathsf{barOf\ Lam} = (0 \mapsto (\mathsf{lam}, \mathsf{lam}))$.

For the CCS example in Section 3.1.5, we fix a type *chan* of channels. We choose a cardinal upper bound κ for the branching of sum ($\sum$), and choose a type *index* of cardinality κ. For *bindex*, we do not need anything special, so we take it to be *nat*. We have two sorts, of expressions and processes, so we take $\mathit{sort} = \{\mathsf{exp}, \mathsf{proc}\}$. Since we have expression variables but no process variables, we take $\mathit{varsort} = \{\mathsf{varexp}\}$ and asSort to send varexp to exp. We define *opsym* as the following datatype: $\mathit{opsym} = \mathsf{Zero} \mid \mathsf{Plus} \mid \mathsf{Inp}\ \mathit{chan} \mid \mathsf{Out}\ \mathit{chan} \mid \sum(\mathit{index\ set})$. The free and bound arities and sorts of the operation symbols are as expected. For example, $\mathsf{Inp}\ c$ acts similarly to λ-abstraction, but binds, in proc terms, variables of a different sort, varexp: $\mathsf{arOf}\,(\mathsf{Inp}\ c) = ()$, $\mathsf{barOf}\,(\mathsf{Inp}\ c) = (0 \mapsto (\mathsf{varexp}, \mathsf{proc}))$. For $\sum I$ with $I : \mathit{index\ set}$, the arity is only defined for elements of I, namely $\mathsf{arOf}\,(\sum I) = ((i \in I) \mapsto \mathsf{proc})$.

3.5.2 Well-Sorted Terms over a Signature

Based on the information from a signature, we can distinguish our terms of interest, namely those that are well-sorted in the sense that:

- all variables are embedded into terms of sorts compatible with their varsorts
- all operation symbols are applied according their free and bound arities

This is modeled by well-sortedness predicates $\mathsf{wls} : \mathit{sort} \to \mathit{term} \to \mathit{bool}$ and $\mathsf{wlsAbs} : \mathit{varsort} \times \mathit{sort} \to \mathit{abs} \to \mathit{bool}$, where $\mathsf{wls}\ s\ X$ states that X is a well-sorted term of sort s and $\mathsf{wlsAbs}\ (xs, s)\ A$ states that A is a well-sorted abstraction binding an xs-variable in an s-term. They are defined mutually inductively by the following clauses:

$$\begin{array}{rcl} & & \mathsf{wls}\ (\mathsf{asSort}\ xs)\ (\mathsf{Var}\ xs\ x) \\ \uparrow\mathsf{wls}\ (\mathsf{arOf}\ \delta)\ \mathit{inp} \wedge \uparrow\mathsf{wlsAbs}\ (\mathsf{barOf}\ \delta)\ \mathit{binp} & \Longrightarrow & \mathsf{wls}\ (\mathsf{stOf}\ \delta)\ (\mathsf{Op}\ \delta\ \mathit{inp}\ \mathit{binp}) \\ \mathsf{isInBar}\ (xs, s) \wedge \mathsf{wls}\ s\ X & \Longrightarrow & \mathsf{wlsAbs}\ (xs, s)\ (\mathsf{Abs}\ xs\ x\ X) \end{array}$$

where $\mathsf{isInBar}\,(xs, s)$ states that the pair (xs, s) is in the bound arity of at least one operation symbol δ, i.e., $\mathsf{barOf}\ \delta\ i = (xs, s)$ for some i— this rules out unneeded abstractions.

Let us illustrate sorting for our running examples. In the λ-calculus syntax, let $X = \mathsf{Var\ lam}\ x$, $A = \mathsf{Abs\ lam}\ x\ X$, and $Y = \mathsf{Op\ Lam}\ ()\ (0 \mapsto A)$. These correspond to what, in the unsorted BNF notation from Section 3.1.1, we would write $\mathsf{Var}\ x$, $\mathsf{Abs}\ x\ X$ and $\mathsf{Lam}\ (\mathsf{Abs}\ x\ X)$. In our sorting system, X and Y are both well-sorted terms at sort lam (written $\mathsf{wls\ lam}\ X$ and $\mathsf{wls\ lam}\ Y$) and A is a well-sorted abstraction at sort $(\mathsf{lam}, \mathsf{lam})$ (written $\mathsf{wlsAbs}\,(\mathsf{lam}, \mathsf{lam})\,A$).

For CCS, we have that $E = \mathsf{Op\ Zero}\,()\,()$ and $F = \mathsf{Op\ Plus}\ (0 \mapsto E,\ 1 \mapsto E)\,()$ are well-sorted terms of sort exp. Moreover, $P = \mathsf{Op}\,(\sum \emptyset)\,()\,()$ and $Q = \mathsf{Op}\,(\mathsf{Out}\ c)\,(0 \mapsto F, 1 \mapsto P)\,()$

are well-sorted terms of sort proc. (Note that P is a sum over the empty set of choices, i.e., the null process, whereas Q represents a process that outputs the value of $0+0$ on channel c and then stops.) If, e.g., we swap the arguments of $\mathsf{Out}\ c$ in Q, we obtain $\mathsf{Op}\,(\mathsf{Out}\ c)\,(0 \mapsto P, 1 \mapsto F)\,()$, which is not well-sorted: In the inductive clause for wls, the input $(0 \mapsto P, 1 \mapsto F)$ fails to match the arity of $\mathsf{Out}\ c$, $(0 \mapsto \mathsf{exp}, 1 \mapsto \mathsf{proc})$.

3.5.3 From Good to Well-Sorted

Recall that goodness means "does not branch beyond $|var|$." On the other hand, well-sortedness imposes that, for each applied operation symbol δ, its inputs have same domains, i.e., *only branch as much*, as the arities of δ. Thus, it suffices to assume the arity domains smaller than $|var|$. We will more strongly assume that the types of sorts and indexes (the latter subsuming the arity domains) are all smaller than $|var|$:

Assumption 29. $|sort| < |var| \ \wedge\ |index| < |var| \ \wedge\ |bindex| < |var|$

Now we can prove:

Proposition 30. $(\mathsf{wls}\ s\ X \Longrightarrow \mathsf{good}\ X) \ \wedge\ (\mathsf{wls}\ (xs, s)\ A \Longrightarrow \mathsf{goodAbs}\ A)$

In addition, we prove that all the standard operators preserve well-sortedness. For example, we prove that if we substitute, in the well-sorted term X of sort s, for the variable y of varsort ys, the well-sorted term Y of sort corresponding to ys, then we obtain a well-sorted term of sort s: $\mathsf{wls}\ s\ X \wedge \mathsf{wls}\ (\mathsf{asSort}\ ys)\ Y \implies \mathsf{wls}\ s\ (X\,[Y/y]_{ys})$.

Using the preservation properties and Prop. 30, we transfer the entire theory of Sections 3.2.4, 3.4 and 3.3 from good terms to well-sorted terms—e.g., Prop. 22(d) becomes:

$$\mathsf{wls}\ s\ X \wedge \mathsf{wls}\ (\mathsf{asSort}\ ys)\ Y_1 \wedge \mathsf{wls}\ (\mathsf{asSort}\ ys)\ Y_2 \Longrightarrow X\,[Y_1/y]_{ys}\,[Y_2/y]_{ys} = \ldots$$

The transfer is mostly straightforward for all facts, including the induction theorem. For stating the sorted version of the recursion and semantic interpretation theorems, there is some additional bureaucracy since we also need sorting predicates on the target domain; we will dedicate to this the next subsection 3.5.4.

There is an important remaining question: Are our two Assumptions (6 and 29) satisfiable? That is, can we find, for any types *sort*, *index* and *bindex*, a type *var* larger than these such that $|var|$ is regular? Fortunately, the theory of cardinals again provides us with a positive answer: Let $G = nat + sort + index + bindex$. Since any successor of an infinite cardinal is regular, we can take *var* to have the same cardinality as the successor of $|G|$, by defining *var* as a suitable subtype of G *set*. In the case of all operation symbols being finitary, i.e., with their arities having finite domains, we do not need the above fancy construction, but can simply take *var* to be a copy of *nat*.

3.5.4 Many-Sorted Recursion

As mentioned in the previous subsection, adapting the theorems from good items to well-sorted items is a routine process. For recursion, the process is more bureaucratic, since it involves the sorting of the target domain as well. We obtain the well-sorted versions of all

the iteration and recursion theorems. We only show here the case of FSb primitive recursion. (The others are similar.)

A *sorted FSb recursion model* is an extension of the concept of FSb model with the following:

- the sorting predicates $\mathsf{wls}^T : \mathit{sort} \to T \to \mathit{bool}$ and $\mathsf{wlsAbs}^T : \mathit{varsort} \times \mathit{sort} \to A \to \mathit{bool}$
- the assumption that all operators preserve sorting, e.g.,

$$\mathsf{wls}\ s\ X' \wedge \mathsf{wls}\ (\mathsf{asSort}\ ys)\ Y' \wedge \mathsf{wls}^T\ s\ X \wedge \mathsf{wls}^T\ (\mathsf{asSort}\ ys)\ Y \\ \Longrightarrow \mathsf{wls}^T\ s\ (\mathsf{SUBST}\ X'\ X\ Y'\ Y\ y\ ys)$$

The recursion Theorem 26 is now extended to take sorting into account:

Theorem 31. For any sorted FSb recursion model, there exist the functions $f : \mathit{term} \to T$ and $\mathit{fAbs} : \mathit{abs} \to A$ that satisfy the same properties as in Theorem 26 and additionally preserve sorting:

- $\mathsf{wls}\ s\ X \Longrightarrow \mathsf{wls}^T\ s\ (f\ X)$
- $\mathsf{wlsAbs}\ (xs, s)\ A \Longrightarrow \mathsf{wlsAbs}^T\ (xs, s)\ (\mathit{fAbs}\ A)$

Similarly, we obtain a sorted version of the semantic interpretation theorem. We define a *sorted semantic domain* to have the same components as a semantic domain from Section 3.3.4, plus sorting predicates wls^{Dt} and wls^{Da}. Again, it is assumed that the semantic operators preserve sorting. Then Theorem 27 is adapted to sorted domains, additionally ensuring the sort preservation of sem and semAbs.

3.5.5 End Product

All in all, the formalization of our first framework provides a theory of syntax with bindings over an arbitrary many-sorted signature. The signature is formalized as an Isabelle locale [39] that fixes the types *var*, *sort*, *varsort*, *index*, *bindex* and *opsym* and the constants asSort, arOf and barOf and assumes the injectivity of asSort and the *var* properties (Assumptions 6 and 29). All end-product theorems are placed in this locale.

The whole formalization consists of 22700 lines of code (LOC). Of these, 3300 LOC are dedicated to quasiterms, their standard operators and alpha-equivalence. 3700 LOC are dedicated to the definition of terms and the lifting of results from quasiterms. Of the latter, the properties of substitution were the most extensive—2500 LOC out of the whole 3700—since substitution, unlike freshness and swapping, requires heavy variable renaming, which complicates the proofs.

The induction scheme presented in Section 3.4 is not the only scheme we formalized (though it is the most useful). We also proved a variety of lower-level induction schemes based on the skeleton of the terms and schemes that are easier to instantiate—e.g., by pre-instantiating Theorem 28 with commonly used parameters such as variables, terms and environments. Induction and iteration/recursion principles constitute 8000 LOC altogether.

The remaining 7700 LOC of the formalization are dedicated to transiting from good terms to sorted terms. Of these, 3500 LOC are taken by the sheer statement of our many end-product theorems. Another fairly large part, 2000 LOC, is dedicated to transferring all the

variants of iteration and recursion (those from Sections 3.3.1 and 3.3.2) and the interpretation Theorem 27, which require conceptually straightforward but technically tedious moves back and forth between sorted terms and sorted elements of the target domain.

Chapter 4

A Formalization of the Church-Rosser and Standardization Theorems

In this chapter, we instantiate the general theory from Chapter 3 to the syntax of lambda-calculus and formalize the development leading to two major λ-calculus results: the Church-Rosser and Standardization theorems for β-reduction. Our work[1] covers both the call-by-name and call-by-value versions of the calculus, following a classic paper by Gordon Plotkin [60].

In our journey towards a better understanding of syntaxes and their binding mechanisms, we wanted some case study, that highlighted the role that a formalized general theory like ours can assume and the good it can do. Indeed, during the formalization, we were able to stay focused on the high-level ideas of the development—thanks to the arsenal provided by our framework: a wealth of basic facts about the substitution, swapping and freshness operators, as well as recursive-definition and reasoning principles.

The first step we take is instantiating the framework to the syntaxes of call-by-name and call-by-value λ-calculus, the latter differing from the former by the existence of an additional syntactic category of special terms called values. This instantiation provides us with a rich theory of the standard operators on terms, namely freshness, substitution and swapping, as well as a freshness-aware induction proof principle and operators-aware recursive definition principles (Section 4.1; see also the previous Chapter 3).

Then we proceed with the formal development of our specific target results. We show in detail the development for the call-by-name calculus (Section 4.2). The similar development for the call-by-value calculus is only sketched by pointing out the differences, including the use of a two-sorted instantiation of our framework (Section 4.3).

The theorems we want to prove require the definition of a range of β-reduction relations, including parallel and left, single-step and multi-step reductions. The Church-Rosser theorem (Section 4.2.2) is proved by formalizing the parallel-reduction technique of Tait [8], enhanced with the complete parallel reduction operator trick due to Takahashi [69]. For Standardization (Section 4.2.3), we follow closely Plotkin's original paper [60].

[1] Our formalization is publicly available from the website [32] associated to our paper [31], currently under review.

Our presentation emphasizes the use of the various principles provided by our general framework, as well as some difficulties arising from representing formally some informal definition and proof idioms—such as recursing over alpha-equated terms (or, equivalently, recursing in an alpha-equivalence preserving manner) and inversion rules obeying Barendregt's variable convention. Some of the lessons learned during the formalization effort, as well as some statistics, are presented in Section 4.4.

4.1 Instantiation of the General Framework

Our framework from the previous chapter is parametrized by an arbitrary binding signature, which is represented as an Isabelle locale [39] (Subsection 3.5.1, Chapter 3). The signature essentially specifies the following data: a collection of term sorts, a collection of variable sorts, an embedding relationship between variable sorts and term sorts, and a collection of (term) constructors, each with an assigned arity and an assigned result sorts. The arity consists of zero or more input sorts—where an input sort is either just a term sort (for free inputs) or a pair of a variable sort and a term sort (for bound inputs). The result sort refers to the output of the constructor. In this chapter we ignore standalone abstractions as presented in the previous chapter, since we have covered them in syntactic sugar, as one of the first steps of the instantiation process. To this aim the technique we used is the one described in Section 3.1.1, Chapter 3: we define $\mathsf{Lm}\ x\ X$ as $\mathsf{Lam}\ (\mathsf{Abs}\ x\ X)$ and from now on all the definitions and results will be written using Lm[2].

After fixing the signature, "quasi-terms" are defined as being freely generated by the constructors, then terms are defined by quotienting quasi-terms to the notion of alpha-equivalence obtained standardly from the signature-specified bindings of the term constructors. Thus, what we call "terms" here are as usual alpha-equivalence classes. Several standard operators are defined on terms, including capture-avoiding substitution of terms for variables, freshness of a variable for a term, and swapping of two variables in a term. The theory provides many properties of these operators, as well as binding-aware and standard-operator-aware structural recursion and induction principles and a principle for interpreting syntax in a semantic domain.

We have already given details about the general framework in the previous chapter 3; here we just give a self-contained description of two instances of the framework.

4.1.1 The Syntax of λ-Calculus

Our first instance is the paradigmatic syntax of λ-calculus (with constants), which is typically informally specified using a grammar such as

$$X \quad ::= \quad \mathsf{Var}\ x \mid \mathsf{Ct}\ c \mid \mathsf{App}\ X\ Y \mid \mathsf{Lm}\ x\ X$$

where X and Y range over terms (the ones generated by the grammar), x over a given infinite type *var* of variables of variables and c over a given type *const* of constants—where Var and

[2] In the framework we will develop in Chapters 5 and 6 there will be no need of this method, since we have not employed standalone abstractions in the first place.

Ct are the embeddings of variables and constants into terms, App is application and Lm is λ-abstraction. Terms are assumed to be equated modulo alpha-equivalence, defined standardly by assuming that, in Lm x X, the λ-constructor Lm binds the variable x in the term X. Thus, for example, Lm x (Var x) = Lm y (Var y) even if $x \neq y$.

We obtain the above syntax by instantiating our signature to consist of:

- a single variable sort, vs, and a single term sort, ts—assuming that vs is embedded in ts gives as the variable-constructor Var, which embeds vs-variables into ts-terms
- the (non-variable) constructors App, Lam, and Ct c for each c : *const*, such that:
  - each Ct c has empty arity and result sort ts, which we can write briefly[3] as $([], \mathsf{ts})$
  - App has its arity consisting of two occurrences of free ts and result sort ts, which we can write as $([\mathsf{ts}, \mathsf{ts}], \mathsf{ts})$
  - Lm has its arity consisting of one vs-bound ts and result sort ts, which we can write as $([(\mathsf{vs}, \mathsf{ts})], \mathsf{ts})$

(Above, we wrote $[a_1, \ldots, a_n]$ for the list consisting of the n elements $a_1, \ldots, a_n$.)

However, this raw instance is not convenient for the user, because it represents a too deep embedding. Thus, the constructors would have to be applied to terms using a generic operator, Op, which is first applied to the desired constructor symbol; sortedness information would have to be unnecessarily carried around; etc.—in other words, one would incur the usual inconvenience arising from instantiating universal algebra to fixed-signature algebras, such as groups or rings.

To address this, we have designed a systematic method for transferring a signature instance to the expected shallow embedding of the instance. This involves creating native Isabelle/HOL types of terms for each sort of the signature and transferring all the term constructors and operators and all facts about them to these native types. The process is conceptually straightforward, but is quite tedious, and must be done by hand since we have not yet automated it. (But [67] presents the automation of a similar kind of transfer.)

For our instance of interest (λ-calculus with constants), this results in the type *term* of λ-terms together with:

- the constructors, namely Var : $var \to term$, Ct : $const \to term$, App : $term \to term \to term$ and Lm : $var \to term \to term$
- and the standard operators:
  - depth (height) of a term, depth : $term \to nat$
  - freshness of a variable in a term,[4] fresh : $var \to term \to bool$
  - (capture-free) substitution of a term for a variable in a term, _[_/_] : $term \to term \to var \to term$
  - (capture-free) parallel substitution of multiple terms for multiple variables in a term, _[_] : $term \to (var \to term\ option) \to term$

[3] The framework uses a more elaborate notation, that exploits partial functions, in order to accommodate a possibly infinite number of inputs, but this is not relevant here.

[4] Other frameworks employ a free-variable operator, FVars : $term \to term\ set$. This is of course inter-definable with the freshness operator.

– swapping of two variables in a term,[5] $\_[\_ \wedge \_] : \mathit{term} \to \mathit{var} \to \mathit{var} \to \mathit{term}$

From our general theory, we also obtain for free:

- many basic facts proved about the constructors and operators
- and induction and recursion principles for proving new facts about terms and defining new functions on terms, respectively

Our framework provides a multitude of general-purpose properties of the constructors and operators, including properties about their mutual interactions (see Subsection 3.2.4 of Chapter 3. For example, the following are two essential properties of equality between λ-abstractions, reflecting the fact that terms are alpha-equivalence classes. The second allows us to rename bound variables with fresh ones, whenever needed.

Proposition 32. *The following hold:*
(1) If $y \notin \{x, x'\}$ *and* $\mathsf{fresh}\ y\ X$ *and* $\mathsf{fresh}\ y\ X'$ *and* $X\,[(\mathsf{Var}\ y)\ /\ x] = X'\,[(\mathsf{Var}\ y)\ /\ x']$ *then* $\mathsf{Lm}\ x\ X = \mathsf{Lm}\ x'\ X'$
(2) If $\mathsf{fresh}\ y\ X$ *then* $\mathsf{Lm}\ x\ X = \mathsf{Lm}\ y\ (X\,[(\mathsf{Var}\ y)\ /\ x])$.

Another example is the compositionality of substitution:

Proposition 33. *The following hold:*
(1) $X\,[Y_1/y]\,[Y_2/y] = X\,[(Y_1\,[Y_2/y])/y]$
(2) If $y \neq z$ *and* $\mathsf{fresh}\ y\ Z$ *then* $X\,[Y/y]\,[Z/z] = X\,[Z/z]\,[(Y\,[Z/z])/y]$

More precisely, the last two results are obtained from the instantiation, to the λ-calculus syntax, of Proposition 10 and of Proposition 22, points (4) and (5).

Fresh structural induction. Our framework also offers a structural induction principle in the style of nominal logic [58, 76, 73] (see Theorem 28 from Section 3.4). It differs from standard structural induction in that, in the inductive Lm-case, it allows one to additionally assume freshness of the Lm-bound variable with respect to any potential parameters of the to-be-proved statement. For the λ-calculus instance, it becomes:

Proposition 34. (Fresh structural induction principle) *Let param be a type (of items called parameters) endowed with a function* $\mathsf{varsOf} : \mathit{param} \to \mathit{var}\ \mathit{set}$ *such that* $\mathsf{varsOf}\ p$ *is finite for all* $p : \mathit{param}$. *Let* $\varphi : \mathit{term} \to \mathit{param} \to \mathit{bool}$ *be a predicate on terms and parameters.*

Assume the following four sentences are true for all $x : \mathit{var}$, $c : \mathit{const}$ *and* $X, Y : \mathit{term}$:
(1) $\varphi\ (\mathsf{Var}\ x)\ p$ *holds for all* $p : \mathit{param}$.
(2) $\varphi\ (\mathsf{Ct}\ c)\ p$ *holds for all* $p : \mathit{param}$.
(3) If $\varphi\ X\ p$ *and* $\varphi\ Y\ p$ *hold for all* $p : \mathit{param}$, *then* $\varphi\,(\mathsf{App}\ X\ Y)\ q$ *holds for all* $q : \mathit{param}$.
(4) If $\varphi\ X\ p$ *holds for all* $p : \mathit{param}$, *then* $\varphi\,(\mathsf{Lm}\ x\ X)\ q$ *holds for all* $q : \mathit{param}$ *such that* $x \notin \mathsf{varsOf}\ q$.

Then $\varphi\ X\ p$ *holds for all* $X : \mathit{term}$ *and* $p : \mathit{param}$.

[5] While not explicitly present in the traditional λ-calculus [8], swapping (as already pointed out in this dissertation) has been popularized by nominal logic as a very convenient operator in bootstrapping definitions—thanks to the fact that bijective renamings behave better than arbitrary renamings with respect to bindings [58].

This immediately implies the following fresh case distinction principle. It states that any term is either a variable, or a constant, or an application, or an abstraction whose bound variable can be taken to be fresh for a given parameter.

Proposition 35. (Fresh case distinction principle) *Let param and* varsOf *be like in the previous proposition and let* Z : *term and* p : *param. Then one of the following holds:*
(1) $Z = \mathsf{Var}\ x$ *for some* x : *var.*
(2) $Z = \mathsf{Ct}\ c$ *for some* c : *const.*
(3) $Z = \mathsf{App}\ X\ Y$ *for some* X, Y : *term.*
(4) $Z = \mathsf{Lm}\ x\ X$ *for some* x : *var and* X : *term such that* $x \notin \mathsf{varsOf}\ p$.

When employing the above principles, the type *param* is often instantiated to $(var + term)\ list$, in order to capture any number of variable and/or term parameters in the to-be-proved statements.

Operator-aware recursion. Our framework offers structural recursion principles (Section 3.3) for defining functions H from terms to any other target type, based on the following ingredients:

- a description of the recursive behavior of H with respect to the term constructors (as is common with primitive recursion on free datatypes)
- a description of the expected interaction of H with freshness on the one hand and substitution and/or swapping on the other hand

These are achieved by organizing the target type as a "model" that interprets the constructors and the operators in specific ways.

Definition 36. *A* freshness-substitution model (FSb model) *is a type* D *endowed with the following:*

- *functions on* D *having similar types as the term constructors (but with term replaced with* D *in their target type and with the pair of term and* D *in their source types), namely* $\mathsf{VAR} : var \to D$, $\mathsf{CT} : const \to D$, $\mathsf{APP} : term \to D \to term \to D \to D$ *and* $\mathsf{LM} : var \to term \to D \to D$
- *functions on* D *having similar types as the freshness and substitution operators (again, with term suitably replaced with* D *or with term and* D*), namely* $\mathsf{FRESH} : var \to term \to D \to bool$ *and* $\mathsf{SUBST} : term \to D \to term \to D \to var \to D$

The above functions are allowed to be defined in any way, provided they satisfy the following freshness clauses (F1)-(F5), substitution clauses (Sb1)–(Sb4) and substitution-renaming clause (SbRn):

F1: $\mathsf{FRESH}\ x\ (\mathsf{Ct}\ c)\ (\mathsf{CT}\ c)$

F2: $x \neq z$ *implies* $\mathsf{FRESH}\ z\ (\mathsf{Var}\ x)\ (\mathsf{VAR}\ x)$

F3: $\mathsf{FRESH}\ z\ X'\ X$ *and* $\mathsf{FRESH}\ z\ Y'\ Y$ *implies* $\mathsf{FRESH}\ z\ (\mathsf{App}\ X'\ Y')\ (\mathsf{APP}\ X'\ X\ Y'\ Y)$

F4: $\mathsf{FRESH}\ z\ (\mathsf{Lm}\ z\ X')\ (\mathsf{LM}\ z\ X'\ X)$

F5: $\mathsf{FRESH}\ z\ X'\ X$ *implies* $\mathsf{FRESH}\ z\ (\mathsf{Lm}\ x\ X')\ (\mathsf{LM}\ x\ X'\ X)$

Sb1: $\mathsf{SUBST}\ (\mathsf{Var}\ z)\ (\mathsf{VAR}\ z)\ Z'\ Z\ z = Z$

Sb2: $x \neq z$ *implies* $\mathsf{SUBST}\ (\mathsf{Var}\ x)\ (\mathsf{VAR}\ x)\ Z'\ Z\ z = \mathsf{VAR}\ x$

Sb3: $\mathsf{SUBST}\ (\mathsf{App}\ X'\ Y')\ (\mathsf{APP}\ X'\ X\ Y'\ Y)\ Z'\ Z\ z =$
$\mathsf{APP}\ (X'[Z'\ /\ z])\ (\mathsf{SUBST}\ X'\ X\ Z'\ Z\ z)\ (Y'[Z'\ /\ z])\ (\mathsf{SUBST}\ Y'\ Y\ Z'\ Z\ z)$

Sb4: $x \neq z$ *and* $\mathsf{FRESH}\ x\ Z'\ Z$ *implies*
$\mathsf{SUBST}\ (\mathsf{Lm}\ z\ X')\ (\mathsf{LM}\ x\ X'\ X)\ Z'\ Z\ z = \mathsf{LM}\ x\ (X'[Z'\ /\ z])\ (\mathsf{SUBST}\ X'\ X\ Z'\ Z\ z)$

SbRn: $x \neq y$ *and* $\mathsf{FRESH}\ y\ X'\ X$ *implies*
$\mathsf{LM}\ y\ (X'[(\mathsf{Var}\ y)\,/\,x])\ (\mathsf{SUBST}\ X'\ X\ (\mathsf{Var}\ y)\ (\mathsf{VAR}\ y)\ x) = \mathsf{LM}\ x\ X'\ X$

Definition 37. *A* freshness-swapping model (FSw model) *is similar to an FSb-model, except that it has a swapping-like function* $\mathsf{SWAP} : \mathit{term} \to D \to \mathit{var} \to \mathit{var} \to D$ *instead of the substitution-like function* SUBST *and satisfies the following swapping clauses (Sw1)–(Sw4) and swapping-congruence clause (SwCg) instead of the substitution-related clauses (Sb1)–(Sb4) and (SbRn):*

Sw1: $\mathsf{SWAP}\ (\mathsf{Ct}\ c)\ (\mathsf{CT}\ c)\ z_1\ z_2\ =\ \mathsf{CT}\ c$

Sw2: $\mathsf{SWAP}\ (\mathsf{Var}\ x)\ (\mathsf{VAR}\ x)\ z_1\ z_2\ =\ \mathsf{VAR}\ (x\,[z_1 \wedge z_2])$

Sw3: $\mathsf{SWAP}\ (\mathsf{App}\ X'\ Y')\ (\mathsf{APP}\ X'\ X\ Y'\ Y)\ z_1\ z_2\ =$
$\mathsf{APP}\ (X'\,[z_1 \wedge z_2])\ (\mathsf{SWAP}\ X'\ X\ z_1\ z_2)\ (Y'\,[z_1 \wedge z_2])\ (\mathsf{SWAP}\ Y'\ Y\ z_1\ z_2)$

Sw4: $\mathsf{SWAP}\ (\mathsf{Lm}\ x\ X')\ (\mathsf{LM}\ x\ X'\ X)\ z_1\ z_2 = \mathsf{LM}\ (x\,[z_1 \wedge z_2])\ (X'\,[z_1 \wedge z_2])\ (\mathsf{SWAP}\ X'\ X\ z_1\ z_2)$

SwCg: $\mathsf{FRESH}\ z\ X'\ X$ *and* $\mathsf{FRESH}\ z\ Y'\ Y$ *and* $z \notin \{x, y\}$ *and* $\mathsf{SWAP}\ X'\ X\ z\ x = \mathsf{SWAP}\ Y'\ Y\ z\ y$
implies $\mathsf{LM}\ x\ X'\ X = \mathsf{LM}\ y\ Y'\ Y$

The framework's recursion principles essentially say that terms form the initial FSb and FSw models:[6]

Proposition 38. *Let D be an FSb model (FSw model, respectively). Then there exists a unique function $H : \mathit{term} \to D$ commuting with the constructors, i.e.,*

- $H\ (\mathsf{Var}\ x) = \mathsf{VAR}\ x$
- $H\ (\mathsf{Ct}\ c) = \mathsf{CT}\ c$
- $H\ (\mathsf{App}\ X\ Y) = \mathsf{APP}\ X\ (H\ X)\ Y\ (H\ Y)$
- $H\ (\mathsf{Lm}\ x\ X) = \mathsf{LM}\ x\ X\ (H\ X)$

Additionally, H preserves freshness and commutes with substitution (respectively, swapping):

- $\mathsf{fresh}\ x\ X$ *implies* $\mathsf{FRESH}\ x\ X\ (H\ X)$

[6] The reason why we define our models' operations to act not only on the models' carrier type D but also on *term* is to achieve the higher flexibility of *primitive recursion* compared to *iteration*—see [62, §1.4.2] for a detailed discussion of this distinction.

- $H\ (X[Z\,/\,z]) = \mathsf{SUBST}\ X\ (H\ X)\ Z\ (H\ Z)\ z$
  (respectively, $H\ (X[z_1 \wedge z_2]) = \mathsf{SWAP}\ X\ (H\ X)\ z_1\ z_2$*)*

The principle is much easier to use in practice than its elaborate formulation might suggest: Say one wishes to define a function H from *term* to a type D. Then the functions on D corresponding to the term constructors are determined from the desired recursive clauses for H. Moreover, the functions on D corresponding to freshness and substitution or swapping are determined by the desired behavior of H with respect to these operators, obtained from answering questions such as "How can $H\,(X[Z\ /\ x])$ be expressed in terms of $H\ X$, $H\ Z$ and x?". This methodology is presented in [63] and illustrated in its generality in Chapter 3 of this thesis. Below we show just the definition of a basic recursive function, in order to emphasize the simplicity of the process when using our principle. We will define the *depth* of a term. For development reasons, this function is defined in our framework at a quasi-term level and then it is transported to terms proving that it preserves alpha-equivalence, but it can as well be defined directly on terms and we show this in what follows. We want to define $\mathsf{depth} : X \to \mathit{nat}$ in a way that it respects the following recursive clauses:

- $\mathsf{depth}(\mathsf{Var}\ x) = 1$;
- $\mathsf{depth}\ (\mathsf{Ct}\ c) = 1$
- $\mathsf{depth}\ (\mathsf{App}\ X\ Y) = (\mathsf{max}\ (\mathsf{depth}\ (X))\ (\mathsf{depth}\ (Y))) + 1$
- $\mathsf{depth}\ (\mathsf{Lm}\ x\ X) = (\mathsf{depth}\ (X)) + 1$

We have just seen, that in order to properly instantiate our recursion principle, we have to think of the desired behaviour for freshness and at least one between swapping or substitution. In this case we pick swapping and we ask that the behaviour of our function is trivial (but also natural): every variable will be fresh for the natural number in the image of any term and the depth of a term in which two variable are swapped will be the same of the original term. Formally our model will be the type of natural numbers *nat* endowed with constructors- and operators-like functions as follows:

- $\mathsf{VAR}\ x = 1$
- $\mathsf{CT}\ c = 1$
- $\mathsf{APP}\ X\ n\ Y\ m = (\mathsf{max}\ n\ m) + 1$
- $\mathsf{LM}\ x\ X\ n = n + 1$
- $\mathsf{FRESH}\ x\ X\ n = \mathsf{True}$ (for all n)
- $\mathsf{SWAP}\ X\ n\ x\ y = n$

Note that the freshness clauses (F1)-(F5) are this way trivially satisfied. As for the swapping clauses and the swapping-congruence clause:

- Sw1: $\mathsf{SWAP}\ (\mathsf{Ct}\ c)\ (\mathsf{CT}\ c)\ x\ y\ =\ (\mathsf{CT}\ c)$, after unfolding the definitions of CT and SWAP, what is left to prove is $1 = 1$;
- Sw2: $\mathsf{SWAP}\ (\mathsf{Var}\ x)\ (\mathsf{VAR}\ x)\ z_1\ z_2\ =\ \mathsf{VAR}\ (x[z_1 \wedge z_2])$, after unfolding the definitions of VAR and SWAP, what is left to prove is $1 = 1$;

- Sw3: $\mathsf{SWAP}\ (\mathsf{App}\ X\ Y)\ (\mathsf{APP}\ X\ n\ Y\ m)\ z_1\ z_2\ =$
  $\mathsf{APP}\ (X\,[z_1 \wedge z_2])\ (\mathsf{SWAP}\ X\ n\ z_1\ z_2)\ (Y'\,[z_1 \wedge z_2])\ (\mathsf{SWAP}\ Y\ m\ z_1\ z_2)$, after unfolding the definitions of APP and SWAP, what is left to prove is $(\mathsf{max}\ n\ m)+1 = (\mathsf{max}\ n\ m)+1$;
- Sw4: $\mathsf{SWAP}\ (\mathsf{Lm}\ x\ X)\ (\mathsf{LM}\ x\ X\ n)\ z_1\ z_2 = \mathsf{LM}\ (x\,[z_1 \wedge z_2])\ (X\,[z_1 \wedge z_2])\ (\mathsf{SWAP}\ X\ n\ z_1\ z_2)$, after unfolding the definitions of LM and SWAP, what is left to prove is $n+1 = n+1$;
- SwCg: $\mathsf{FRESH}\ z\ X\ n$ and $\mathsf{FRESH}\ z\ Y\ m$ and $z \notin \{x,y\}$ and $\mathsf{SWAP}\ X\ n\ z\ x = \mathsf{SWAP}\ Y\ m\ z\ y$ implies $\mathsf{LM}\ x\ X\ n = \mathsf{LM}\ y\ Y\ m$, after unfolding the definitions of FRESH, SWAP and LM, what is left to prove is with: $z \notin \{x,y\}$ and $n = m$ implies $n+1 = m+1$.

To summarize: in order to define the depth function by recursion, we only had to (1) define a model "naturally", namely asking that the constructors- and operators-like functions behave according to the recursive clauses we would anyway ask to the function and (2) prove some properties about them, which are automatically handled in Isabelle/HOL (i.e., by `auto`).

Finally note that, in the formalization of this λ-calculus instance of our definition principles, we chose to get the full primitive-recursion version (see Subsection 3.3.2): the definition of many functions throughout the whole development was not easily handled by simple iteration, e.g., the fundamental one for the complete parallel reduction operator, which will be defined in Subsection 4.2.2, Proposition 42.

4.1.2 The Two-Sorted Syntax of λ-Calculus with Values Emphasized

We can split the syntax of λ-calculus in two syntactic categories, by distinguishing the subcategory of values, which consist of variables, constants and Lm-terms. This distinction is quite customary when modeling higher-order programming language semantics, where values are the only programs that have a "static" identity (whereas the non-values must be run/evaluated). Thus, we consider the mutually recursive syntactic categories of values, ranged over V, W and (arbitrary) terms, ranged over by X, Y, Z:

$$\begin{array}{lcl} X & ::= & \mathsf{Val}\ V \mid \mathsf{App}\ X\ Y \\ V & ::= & \mathsf{Var}\ x \mid \mathsf{Ct}\ c \mid \mathsf{Lm}\ x\ X \end{array}$$

where Val is the injection of values into terms.

We capture the above syntax by instantiating our signature to consists of:

- a single variable sort, vs, and two term sorts, vls and ts (for values and terms) and assuming that vs is embedded in vls, which again produces the variable-constructor Var, this time embedding variables into values
- four (non-variable) constructors, Val, Ct c (for each c : *const*), App and Lm, of arities $([\mathsf{vls}], \mathsf{ts})$, $([], \mathsf{vls})$, $([\mathsf{ts}, \mathsf{ts}], \mathsf{ts})$ and $([(\mathsf{vs}, \mathsf{ts})], \mathsf{vls})$, respectively

Applying the same systematic deep-to-shallow transfer process as for the previous one-sorted syntax, we obtain "native" types *val* and *term* for values and terms, the expected constructors (e.g., Val : $val \to term$) and the standard operators, one for either syntactic category (e.g., $\mathsf{fresh}_{val} : var \to val \to bool$ and $\mathsf{fresh}_{term} : var \to term \to bool$). The framework-provided induction and recursion principles now refer to these mutually recursive types: Induction allows us to prove two simultaneous predicates and recursion allows us two define

two simultaneous functions, one on values and one on terms—this will be illustrated in Section 4.3.

4.2 Call-by-Name λ-Calculus

In this section, we show how we have used our framework's infrastructure to formalize two landmark results in the theory of call-by-name (CBN) λ-calculus: the Church-Rosser theorem [8], stating that the order in which call-by-name redexes are reduced is irrelevant "in the long run" and the standardization theorem [60], stating that reducibility is not restricted if we impose a canonical reduction strategy, based on identifying left-most redexes.

All throughout this section, we employ the (single-sorted) syntax of λ-calculus with constants described in Section 4.1.1. We also fix a partial function Ctapp that shows how to apply a constant c_1 to another constant c_2; Ctapp c_1 c_2 can be either None, meaning "no result," or Some X, meaning "the result is X."

4.2.1 Call-by-Name β-Reduction

Evaluation of a λ-calculus term proceeds by reducing *redexes*, which are subterms of one of the following two kinds:

- either β-*redexes*, of the form App (Lm y X) Y, which are reduced to $X\,[Y\,/\,y]$
- or δ-*redexes*, of the form App (Ct c_1) (Ct c_2) such that Ctapp c_1 c_2 has the form Some X, which are reduced to X

The first are general-purpose redexes arising when an abstraction meets an application, whereas the second are custom redexes representing the functionality built in the constants.

In the CBN calculus, there is no restriction on the terms Y located at the right of β-redexes, reflecting the intuition that the argument Y is passed to the function Lm y X "by name," i.e., without first evaluating it. This style of reduction is captured by the following definition:

Definition 39. *The* one-step (CBN) reduction *relation* $\rightarrow$: *term* $\rightarrow$ *term* $\rightarrow$ *bool is defined inductively by the following rules:*

$$\frac{}{\mathsf{App}\ (\mathsf{Lm}\ y\ X)\ Y \rightarrow X\ [Y\ /\ y]}\ (\beta) \qquad \frac{\mathsf{Ctapp}\ c_1\ c_2\ =\ \mathsf{Some}\ X}{\mathsf{App}\ c_1\ c_2 \rightarrow X}\ (\delta)$$

$$\frac{X \rightarrow X'}{\mathsf{App}\ X\ Y \rightarrow \mathsf{App}\ X'\ Y}\ (\mathsf{AppL}) \qquad \frac{Y \rightarrow Y'}{\mathsf{App}\ X\ Y \rightarrow \mathsf{App}\ X\ Y'}\ (\mathsf{AppR})$$

$$\frac{X \rightarrow X'}{\mathsf{Lm}\ y\ X \rightarrow \mathsf{Lm}\ y\ X'}\ (\xi)$$

The reflexive-transitive closure of $\rightarrow$*, denoted by* $\rightarrow^*$*, is called the* multi-step reduction.

Above, the rules (AppL), (AppR) and (ξ) delve into the term to locate a redex, whereas (β) and (δ) perform the redex reduction. Note that $X \rightarrow X'$ means that X' was obtained from X by the reduction of *precisely one* (nondeterministically chosen) redex.

4.2.2 The Church-Rosser Theorem

A binary relation $\succ$ is called *confluent* provided it satisfies the following "diamond" property: For all u, v_1, v_2 such that $u \succ v_1$ and $u \succ v_2$, there exists w such that $v_1 \succ w$ and $v_2 \succ w$. In other words, every span can be joined. The Church-Rosser theorem states that this is the case for multi-step reduction:

Theorem 40. *$\to^*$ is confluent.*

A difficulty when trying to prove this theorem is the need to work with multiple reduction steps. Indeed, $\to$ itself is not confluent, as seen by the following example. Let $X = \mathsf{App}\ (\mathsf{Lm}\ x_1\ (\mathsf{App}\ (\mathsf{Var}\ x_1)\ (\mathsf{Var}\ x_1)))\ X_1$, where $X_1 = \mathsf{App}\ (\mathsf{Lm}\ x\ (\mathsf{Var}\ x))\ (\mathsf{Ct}\ c)$. If we choose to reduce the top redex of X, we obtain $X \to Y_1$, where $Y_1 = (\mathsf{App}\ (\mathsf{Var}\ x_1)\ (\mathsf{Var}\ x_1))\,[X_1\ /\ x_1] = \mathsf{App}\ X_1\ X_1$. On the other hand, if we choose to reduce the inner redex of X (within X_1), we obtain $X \to Y_2$, where $Y_2 = \mathsf{App}\ (\mathsf{Lm}\ x_1\ (\mathsf{App}\ (\mathsf{Var}\ x_1)\ (\mathsf{Var}\ x_1)))\ (\mathsf{Ct}\ c)$. In order to join Y_1 and Y_2, intuitively we must perform the complementary reductions: By reducing the top redex in Y_2, we obtain $Y_2 \to Z$, where $Z = \mathsf{App}\ (\mathsf{Ct}\ c)\ (\mathsf{Ct}\ c)$. However, Y_1 is not just one, but two redexes away from Z, meaning that $Y_1 \to Z$ does not hold (although $Y_1 \to^* Z$ does).

Dealing with multiple steps in the proof is possible, but the reasoning becomes intricate. A more elegant solution, due to William Tait, proceeds along the following lines [8]:

(1) First define a relation $\Rightarrow$ allowing the reduction of multiple (zero or more) redexes in parallel and prove that its transitive closure, $\Rightarrow^*$, is the same as $\to^*$.

(2) Then prove that $\Rightarrow$ is confluent—which should be possible thanks to parallelism. In the above example, we would have $Y_1 \Rightarrow Z$ by the parallel reduction of two Z-redexes.

Then the proof of the Church-Rosser theorem would be immediate: Since $\Rightarrow$ is confluent, than so is $\Rightarrow^*$, i.e., $\to^*$. Next we proceed with tasks (1) and (2).

Definition 41. *The* one-step parallel reduction *relation* $\Rightarrow : \mathit{term} \to \mathit{term} \to \mathit{bool}$ *is defined inductively by the following rules:*

$$\frac{\mathsf{Ctapp}\ c_1\ c_2\ =\ \mathsf{Some}\ X}{\mathsf{App}\ c_1\ c_2 \Rightarrow X}\ (\delta) \qquad \frac{X \Rightarrow X' \qquad Y \Rightarrow Y'}{\mathsf{App}\ (\mathsf{Lm}\ y\ X)\ Y \Rightarrow X'[Y'\ /\ y]}\ (\beta)$$

$$\frac{X \Rightarrow X' \qquad Y \Rightarrow Y'}{\mathsf{App}\ X\ Y \Rightarrow \mathsf{App}\ X'\ Y'}\ (\mathsf{App}) \qquad \frac{}{X \Rightarrow X}\ (\mathsf{Refl})$$

$$\frac{X \Rightarrow X'}{\mathsf{Lm}\ y\ X \Rightarrow \mathsf{Lm}\ y\ X'}\ (\xi)$$

The key technical differences between the definition of $\Rightarrow$ and that of $\to$ are the following. There are no left and right rules for delving into application subterms, but only one rule, (App), allowing the identification of redexes on multiple locations in the term in parallel. Moreover, the (β) rule allows reducing a redex and at the same time continuing to search for redexes further down in the components. Finally, the (Refl) rule allows to abandon the search at any moment, on any reached location.

It is not difficult to prove (by standard rule induction) that $X \to Y$ implies $X \Rightarrow Y$ and that $X \Rightarrow Y$ implies $X \to^* Y$, which ensure that $\Rightarrow^* \ = \ \to^*$. This concludes task (1). Our

formal proof required no special binding-aware type of reasoning, but only standard inductive definitions and rule-induction proofs.

Moving on to task (2), proving that $\Rightarrow$ is confluent, the simplest known approach is due to Takahashi [69]. Let us assume that $X \Rightarrow Y_1$ and $X \Rightarrow Y_2$, which means that both Y_1 and Y_2 have been obtained from X by the parallel reduction of a number of redexes—it is the choice of which redexes have been reduced and which have been ignored (via the (Refl) rule) that constitutes the difference between Y_1 and Y_2. Hence, if Z is the term obtained from X by a *complete* parallel reduction (with no redexes ignored)—which we write as $Z = \mathsf{cpred}\ X$—then Z would be a valid join for Y_1 and Y_2. Indeed, Z would be obtained from both Y_1 and Y_2 by reducing the redexes that had been ignored during the reductions of X to Y_1 and Y_2.

To define the complete parallel reduction operator (sometimes called "complete development" in the literature), $\mathsf{cpred} : \mathit{term} \to \mathit{term}$, intuitively all we need to do is follow the inductive definition of parallel reduction and make that into a structurally recursive function—while restricting the application of the (Refl) rule to variables and constants only, for not skipping the reduction of any redex:

$$\mathsf{cpred}\ (\mathsf{Var}\ x) = \mathsf{Var}\ x \qquad \mathsf{cpred}\ (\mathsf{Ct}\ c) = \mathsf{Ct}\ c \qquad \mathsf{cpred}\ (\mathsf{Lm}\ y\ X) = \mathsf{Lm}\ y\ (\mathsf{cpred}\ X)$$

$$\mathsf{cpred}\ (\mathsf{App}\ X\ Y)\ = \begin{cases} \mathsf{cpred}\ Z, & \text{if } (X,Y) \text{ have the form } (\mathsf{Ct}\ c_1, \mathsf{Ct}\ c_2) \\ & \quad \text{with } \mathsf{Ctapp}\ c_1\ c_2 = \mathsf{Some}\ Z \\ (\mathsf{cpred}\ Z)\ [(\mathsf{cpred}\ Y)/y], & \text{if } X \text{ has the form } \mathsf{Lm}\ y\ Z \\ \mathsf{App}\ (\mathsf{cpred}\ X)\ (\mathsf{cpred}\ Y), & \text{otherwise} \end{cases}$$

However, the problem is that this definition is not *a priori* guaranteed to be correct, given that terms are not a free datatype due to quotienting to alpha-equivalence. One approach would be to redefine cpred on (unquotiented) quasi-terms and prove that it respects alpha-equivalence, but this would be technically quite difficult and would require breaking the term abstraction layer. Our recursion principle provides a better alternative: The above clauses are almost sufficient to construct an FSw model. What we additionally need is a specification of the expected behavior of the to-be-defined cpred with respect to freshness and swapping—which is straightforward, since cpred is expected to preserve freshness and commute with swapping: $\mathsf{fresh}\ y\ X$ implies $\mathsf{fresh}\ y\ (\mathsf{cpred}\ X)$ and $\mathsf{cpred}\ (X[z_1 \wedge z_2]) = (\mathsf{cpred}\ X)[z_1 \wedge z_2]$.

Our recursion principle can now be employed to obtain the following:

Proposition 42. *There exists a unique function* $\mathsf{cpred} : \mathit{term} \to \mathit{term}$ *satisfying all the above clauses (for the term constructors as well as the freshness and swapping operators).*

Indeed, rewriting these clauses to make the required structure on the target type explicit, we see that they simply state the commutation of cpred with the constructors and the operators as described in Prop. 38, where:

- $\mathsf{VAR} = \mathsf{Var}$ and $\mathsf{CT} = \mathsf{Ct}$
- $\mathsf{LM}\ x\ X'\ X\ =\ \mathsf{Lm}\ x\ X$

- $\mathsf{APP}\ X'\ X\ Y'\ Y\ =\ \begin{cases} Z \text{ if } (X', Y') \text{ have the form } (\mathsf{Ct}\ c_1, \mathsf{Ct}\ c_2) \\ \quad \text{with } \mathsf{Ctapp}\ c_1\ c_2 = \mathsf{Some}\ Z \\ Z\,[Y/y] \quad \text{if } X \text{ has the form } \mathsf{Lm}\ y\ Z \text{ and } X' \text{ has the form } \mathsf{Lm}\ y'\ Z' \\ \mathsf{App}\ X\ Y \quad \text{otherwise} \end{cases}$
- $\mathsf{FRESH}\ x\ X'\ X\ =\ \mathsf{fresh}\ x\ X$
- $\mathsf{SWAP}\ X'\ X\ z_1\ z_2\ =\ X[z_1\ \wedge\ z_2]$

Verifying the FSw model clauses for the above is completely routine (follows by Isabelle's "auto" proof method, which applies the natural simplification rules for term constructors and operators). Note that Prop. 38 does not require the target type of the defined function to be a "syntactic" domain such as *term* (or to satisfy any finite-support property)—although this happens to be the case here. With the definition of cpred in place, it remains to prove the following:

Lemma 1. *$X \Rightarrow X'$ implies $X' \Rightarrow \mathsf{cpred}\ X$*

The informal proof of this lemma would go by induction on X, applying the Barendregt convention in the Lm-case, i.e., when X has the form $\mathsf{Lm}\ y\ Y$, to ensure that the bound variable y is fresh for X'. One might expect that the structural fresh induction principle (Prop. 34) is ideal for formalizing this task. However, the problem is that cpred analyzes X more than one-level deep—when testing if X is a β-redex, i.e., has the form $\mathsf{App}\ (\mathsf{Lm}\ x_1\ X_1)\ X_2$. This means that we need to go a bit beyond structural induction. We therefore use induction on the depth of X,[7] and take advantage of the Barendregt convention by means of the fresh case distinction principle (Prop. 35) instead.

4.2.3 The Standardization Theorem

The relation $\rightarrow$ makes a completely nondeterministic choice of the redex it reduces. The standardization theorem [60] refers to enforcing, without loss of expressiveness, a "standard" reduction strategy, which prioritizes leftmost redexes.

Definition 43. *The* one-step left reduction *relation* $\not\rightarrow : \mathit{term} \rightarrow \mathit{term} \rightarrow \mathit{bool}$ *is defined inductively by the following rules:*

$$\frac{\mathsf{Ctapp}\ c_1\ c_2\ =\ \mathsf{Some}\ X}{\mathsf{App}\ c_1\ c_2 \not\rightarrow X}\ (\delta) \qquad \frac{}{\mathsf{App}\ (\mathsf{Lm}\ y\ X)\ Y \not\rightarrow X\ [Y\ /\ y]}\ (\beta)$$

$$\frac{X \not\rightarrow X'}{\mathsf{App}\ X\ Y \not\rightarrow \mathsf{App}\ X'\ Y}\ (\mathsf{AppL}) \qquad \frac{X \text{ has the form } \mathsf{Var}\ x \text{ or } \mathsf{Ct}\ c \qquad Y \not\rightarrow Y'}{\mathsf{App}\ X\ Y \not\rightarrow \mathsf{App}\ X\ Y'}\ (\mathsf{AppR})$$

A first difference between $\not\rightarrow$ and $\rightarrow$ is that the former gives preference to redexes located towards the lefthand side of the term—as shown by the fact that the rule (AppL) has no restriction on Y, whereas (AppR) requires X to be a variable or a constant. In other words, exploring the righthand side of the term in search for redexes is only allowed if exploring the lefthand side is no longer possible. Another difference is that $\not\rightarrow$ does not reduce under Lm—as shown by the absence of a (ξ) rule.

[7] The depth function is built in our framework, but could have been defined by our recursion principle.

Definition 44. *The* standard reduction (s.r.) sequence *predicate* $\mathsf{srs} : \mathit{term\ list} \to \mathit{bool}$ *is defined inductively by the following rules:*

$$\frac{}{\mathsf{srs}\ [\mathsf{Ct}\ c]}\ (\mathsf{Ct}) \qquad \frac{}{\mathsf{srs}\ [\mathsf{Var}\ x]}\ (\mathsf{Var})$$

$$\frac{X \not\to \mathsf{hd}\ Xs \qquad \mathsf{srs}\ Xs}{\mathsf{srs}\ (X \cdot Xs)}\ (\mathsf{Red}) \qquad \frac{\mathsf{srs}\ Xs}{\mathsf{srs}\ (\mathsf{map}\ (\mathsf{Lm}\ x)\ Xs)}\ (\mathsf{Lm})$$

$$\frac{\mathsf{srs}\ Xs \qquad \mathsf{srs}\ Ys}{\mathsf{srs}\ (\mathsf{zipApp}\ Xs\ Ys)}\ (\mathsf{App})$$

Above, for any a, $[a]$ denotes the singleton list containing a and hd, $\cdot$ and map denote the usual head, append and map functions on lists. Moreover, zipApp applied to two lists $[X_1,\ldots,X_n]$ and $[Y_1,\ldots,Y_m]$ yields the list $[(\mathsf{App}\ X_1\ Y_1,\ldots,\mathsf{App}\ X_n\ Y_1,\ldots,\mathsf{App}\ X_n\ Y_m)]$ (obtained from first applying to Y_1 the terms $X_1,\ldots,X_n$, followed by applying X_n to the terms $Y_2,\ldots,Y_m$).

A standard reduction sequence $[X_1,\ldots,X_n]$ represents a systematic way of performing reduction, prioritizing left reduction, but also eventually exploring rightward located redexes. Thus, the rule (App) merges two s.r. sequences under the App construct, scheduling the left one first and the right one second. The standardizaton theorem states that standard reduction sequences cover all possible reductions.

Theorem 45. *$X \to^* X'$ iff there exists a s.r. sequence starting in X and ending in X'.*

The "if" direction, stating that s.r. sequences are subsumed by arbitrary reduction sequences, follows immediately by rule induction on the definition of srs. So let us focus on the "only if" direction. It turns out that it is easier to use the multi-step parallel reduction $\Rightarrow^*$ instead of $\to^*$—which is OK since we know from Section 4.2.2 that they are equal. To have better control over $\Rightarrow$ (and over $\Rightarrow^*$), we need to be able to count the number of redexes that are being reduced in a step $X \Rightarrow Y$. In his informal proof, Plotkin defines this number by a recursive traversal of the derivation tree for $X \Rightarrow Y$. Since we defined the relation $\Rightarrow$ inductively, i.e., as a least fixed point, we do not have direct access to the derivation trees. Instead, we introduce this number in a labeled variation of $\Rightarrow$, defined inductively as follows:

Definition 46. *The* labeled one-step parallel reduction *relation* $\Rightarrow\_ : \mathit{term} \to \mathit{term} \to \mathit{nat} \to \mathit{bool}$ *is defined inductively by the following rules:*

$$\frac{\mathsf{Ctapp}\ c_1\ c_2\ =\ \mathsf{Some}\ X}{\mathsf{App}\ c_1\ c_2 \Rightarrow_1 X}\ (\delta) \qquad \frac{X \Rightarrow_m X' \qquad Y \Rightarrow_n Y'}{\mathsf{App}\ (\mathsf{Lm}\ y\ X)\ Y \Rightarrow_{1+m+n*\mathrm{no}\ X'\ y} X'[Y'\ /\ y]}\ (\beta)$$

$$\frac{X \Rightarrow_m X' \qquad Y \Rightarrow_n Y'}{\mathsf{App}\ X\ Y \Rightarrow_{m+n} \mathsf{App}\ X'\ Y'}\ (\mathsf{App}) \qquad \frac{}{X \Rightarrow_0 X}\ (\mathsf{Refl})$$

$$\frac{X \Rightarrow_m X'}{\mathsf{Lm}\ y\ X \Rightarrow_m \mathsf{Lm}\ y\ X'}\ (\xi)$$

The definitional rules for $\Rightarrow\_$ are identical to those for $\Rightarrow$, except that they also track the number of reduced redexes. This number evolves as expected, e.g., for applications the left and right numbers are added. The most interesting rule is that for β-reduction, where the label of the conclusion is $1+m+n*\mathrm{no}\ X'\ y$. This is obtained by counting:

- 1 for the top redex (which is being explicitly reduced in the rule)

- m for the redexes being reduced in X to obtain X'
- $n * \mathsf{no}\ X'\ y$ for the n redexes being reduced in Y to obtain Y', *one set for each (free) occurrence of y in X'*—because the occurrences of y in X' correspond to the occurrences of Y in $X'[Y/y]$ that will be reduced to Y'

Thus, $\mathsf{no}\ X'\ y$ counts the number of (free) occurrences of the variable y in X'. For defining no we employ our recursion principle, in its freshness-substitution version:

Definition 47. $\mathsf{no} : \mathit{term} \to (\mathit{var} \to \mathit{nat})$ *is the unique function satisfying the following properties:*

$$\mathsf{no}\ (\mathsf{Var}\ y)\ x = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{if } x \neq y \end{cases} \qquad \mathsf{no}\ (\mathsf{Ct}\ c)\ x = 0$$

$$\mathsf{no}\ (\mathsf{App}\ X\ Y)\ x = \mathsf{no}\ X\ x + \mathsf{no}\ Y\ x \qquad \mathsf{no}\ (\mathsf{Lm}\ y\ X)\ x = \begin{cases} 0, & \text{if } x = y \\ \mathsf{no}\ X\ x, & \text{if } x \neq y \end{cases}$$

$$\mathsf{fresh}\ x\ X \text{ implies } \mathsf{no}\ X\ x = 0 \qquad \mathsf{no}\ (X[Y\ /\ y])\ x = \begin{cases} \mathsf{no}\ X\ y * \mathsf{no}\ Y\ y, & \text{if } x = y \\ \mathsf{no}\ X\ x + \mathsf{no}\ X\ y * \mathsf{no}\ Y\ x, & \text{if } x \neq y \end{cases}$$

To justify the above definition, we construct an FSb model obtained from the above clauses, where $D = \mathit{var} \to \mathit{nat}$ and

- $\mathsf{VAR} : \mathit{var} \to D$ is
$$\mathsf{VAR}\ x = \lambda y.\ \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$
- $\mathsf{CT} : \mathit{const} \to D$ is $\mathsf{CT}\ c = \lambda x.\ 0$
- $\mathsf{APP} : \mathit{term} \to D \to \mathit{term} \to D \to D$ and $\mathsf{APP}\ X\ u\ Y\ v = \lambda x.\ (u\ x) + (v\ x)$
- $\mathsf{LM} : \mathit{var} \to \mathit{term} \to D \to D$ is
$$\mathsf{LM}\ x\ X\ u = \lambda y.\ \begin{cases} 0 & \text{if } x = y \\ u\ y & \text{if } x \neq y \end{cases}$$
- $\mathsf{FRESH} : \mathit{var} \to \mathit{term} \to D \to \mathit{bool}$ is $\mathsf{FRESH}\ x\ X\ u = (u\ x = 0)$
- $\mathsf{SUBST} : \mathit{term} \to D \to \mathit{term} \to D \to \mathit{var} \to D$ is
$$\mathsf{SUBST}\ X\ u\ Y\ v\ y = \lambda x.\ \begin{cases} u\ y * v\ y, & \text{if } x = y \\ u\ x + u\ y * v\ x, & \text{if } x \neq y \end{cases}$$

The Isabelle/HOL code at this stage is similar to the above pen on paper description; the operators characterizing the model are collected in a record:

```
definition no_MOD :: "('const, var ⇒ nat) model"
where "no_MOD ==
(|
 VAR = λx z. if x = z then 1 else 0,
 CT = λc z. 0,
 APP = λX1 f X2 g z. f z + g z,
 LM = λy X f z. if y = z then 0 else f z,
 FRESH = λy X f. f y = 0,
 SUBST = λY g X f y z. if y = z then f z * g z else f z + f y * g z,
|)"
```

Now we are ready to instantiate Prop. 38; once instantiated with this particular model such recursion principle will guarantee that there exist a unique function no : $term \to D$ commuting with the constructors, i.e., the first four properties of Definition 47 above. Additionally , no preserves freshness and commutes with substitution, i.e., the last two properties of Definition 47. Namely, thanks to our recursion principle we are able to formally justify the definition given intuitively, with Definition 47, in a way that it satisfy all the desired properties, of course with some proof obligations.

However verifying Prop. 38's conditions is again routine—some simple arithmetics that has been discharged by the "auto" proof method. For example, let us consider (Sb1):

$$\mathsf{SUBST}\ (\mathsf{Var}\ z)\ (\mathsf{VAR}\ z)\ Z\ u\ z = u\ .$$

After unfolding the definitions, it becomes

$$\lambda x.\ \begin{cases} 1 * u\ z, & \text{if } x = z \\ 0 + 1 * u\ x, & \text{if } x \neq z \end{cases} = u$$

and this is indeed something that can be proved automatically by Isabelle's proof methods. Here we can go even further: in Isabelle the obligations (F1)-(F5), (Sb1)-(Sb4) and (SbRn) are collected in a single predicate over models: wlsFSb. In this particular case, such obligations for defining no can be proved in one shot and very few lines, just by unfolding definition and using basic theorems and proof methods. Below we report the Isabelle code for this lemma:

```
lemma wlsFSb_no_MOD: "wlsFSb no_MOD"
 unfolding wlsFSb_defs
apply auto
by(rule ext, auto simp add: add_mult_distrib)+
```

Note again how we included not only the recursive clauses for the constructors, but also those for the interaction with freshness and substitution. On the one hand, the freshness and substitution clauses are needed to establish the correctness of the definition; on the other hand, they are useful theorems that are produced (and proved) at definition time together with the recursive clauses for the constructors.

Back to the proof of the theorem, because $X \Rightarrow Y$ is immediately equivalent with the existence of $n : nat$ such that $X \Rightarrow_n Y$, we are left with proving the following:

Proposition 48. *If $X \Rightarrow_m^* X'$, then there exists a s.r. sequence starting in X and ending in X'.*

The proof idea for the above is to build the desired s.r. sequence by "consuming" $X \Rightarrow_m^* X'$ one step at a time, from left to right, as expressed below:

Proposition 49. *If $X \Rightarrow_m X'$ and Xs is a s.r. sequence starting in X', then there exists a s.r. sequence starting in X and ending in the last term of Xs.*

Prop. 49 easily implies Prop. 48 by rule induction on the definition of the reflexive-transitive closure; in the base case, one uses the fact that $\mathsf{src}\,[X]$ holds for all terms X, which follows immediately by rule induction on the definition of of src.

So it remains to prove Prop. 49. The proof requires a quite elaborate induction, namely lexicographic induction on three measures: the length of Xs, the number (of X-to-X' reduction steps) m and the depth of X. Inside the induction proof, there is a case distinction on the form of X.

The most complex case is when X is an application, since here we have to deal with the redexes. For handling the β-redex subcase, two lemmas are required. The first states that $\Rightarrow\_$ preserves substitution, while keeping the numeric label under a suitable bound:

Lemma 2. *If $X \Rightarrow_m X'$ and $Y \Rightarrow_n Y'$, then there exists k such that $k \leq m + \mathsf{no}\ X'\ y * n$ and $X\,[Y\,/\,y] \Rightarrow_k X'\,[Y'\,/\,y]$.*

It is proved by induction on the depth of X, making essential use of the property that connects no with substitution, which is built in our definition of no (Def. 47). The second expresses commutation between (labeled) parallel reduction and left reduction:

Lemma 3. *If $X \Rightarrow_m Y$ and $Y \leftarrow\!\!\!\!\rightarrow Z$, then there exist Y' and n such that $X \leftarrow\!\!\!\!\rightarrow^* Y'$ and $Y' \Rightarrow_n Z$.*

It is proved by lexicographic induction on m and the depth of X. Back to the proof of Prop. 49, the other cases (different from App) are conceptually quite straightforward. However, the formal treatment of the Lm-case raises a subtle issue, which we describe next.

The informal reasoning in the Lm-case goes as follows: Assume X has the form $\mathsf{Lm}\ y\ Y$. Then, for inferring $\mathsf{Lm}\ y\ Y \Rightarrow_m X'$, the last applied rule must have been either (Refl) or (ξ). In the case of (Refl), we have $X = X'$ so the desired s.r. sequence is Xs. In the case of (ξ), we obtain that $X' = \mathsf{Lm}\ y\ Y'$ for some Y' such that $Y \Rightarrow_m Y'$. Moreover, since Xs is a s.r. sequence starting in $\mathsf{Lm}\ y\ Y'$, there must be a s.r. sequence Ys starting in Y' such that $Xs = \mathsf{map}\ (\mathsf{Lm}\ y)\ Ys$. By the induction hypothesis, we obtain a s.r. sequence Ys' starting in Y and ending in the last term of Ys. Hence we can take $\mathsf{map}\ (\mathsf{Lm}\ y)\ Ys'$ to be the desired s.r. sequence (starting in X).

The above informal argument applies (among other things) a special inversion rule for $\Rightarrow\_$, taking advantage of knowledge about the shape of the lefthand side of the conclusion: a term of the form $\mathsf{Lm}\ y\ Y$. However, as emphasized above, it is implicitly assumed that an application of the (ξ) rule with $\mathsf{Lm}\ y\ Y$ as lefthand side of its conclusion will have the form

$$\frac{Y \Rightarrow_m Y'}{\mathsf{Lm}\ y\ Y \Rightarrow_m \mathsf{Lm}\ y\ Y'}$$

i.e., will "synchronize" with the variable y bound in Y. In other words, we need the following inversion rule:

Lemma 4. *If* $\mathsf{Lm}\ y\ Y \Rightarrow_m X'$*, then one of the following holds:*

- $X' = \mathsf{Lm}\ y\ Y$ *(meaning (Refl) must have been applied)*
- *There exists* Y' *such that* $X' = \mathsf{Lm}\ y\ Y'$ *and* $Y \Rightarrow_m Y'$ *(meaning a* y*-synchronized* (ξ) *must have been applied)*

Proving the above is not straightforward, and relies on some properties of $\Rightarrow_m$ that are global, i.e., depend on the behavior of its rules different from (ξ). All we can get from the standard inversion rule (coming from the inductive definition of $\Rightarrow_m$) is, in the second case, the existence of z, Z and Z' such that $\mathsf{Lm}\ y\ Y = \mathsf{Lm}\ z\ Z$, $X' = \mathsf{Lm}\ z\ Z'$ and $Z \Rightarrow_m Z'$. Using the properties of equality between Lm-terms, we obtain that $Y = Z\,[y \wedge z]$. To complete the proof of Lemma 4, we further need the following:

Lemma 5. $\Rightarrow_{\_}$ *is equivariant, i.e.,* $Z \Rightarrow_m Z'$ *implies* $Z\,[y \wedge z] \Rightarrow_m Z'\,[y \wedge z]$.

Lemma 6. $\Rightarrow_{\_}$ *preserves freshness, i.e.,* $\mathsf{fresh}\ y\ Z$ *and* $Z \Rightarrow_m Z'$ *implies* $\mathsf{fresh}\ y\ Z'$.

Using these lemmas and the basic properties of freshness and swapping, we define Y' to be $Z'\,[y \wedge z]$ and obtain $\mathsf{Lm}\ y\ Y' = \mathsf{Lm}\ z\ Z'$ and $Y \Rightarrow_m Y'$; in particular, $X' = \mathsf{Lm}\ y\ Y'$ and $Y \Rightarrow_m Y'$, as desired. This concludes our outline of the proof of Prop 49 and overall of the standardization theorem.

4.3 Call-By-Value λ-Calculus

The call-By-Value (CBV) λ-calculus differs from the CBN λ-calculus by the insistence that only values are being substituted for variables in terms, i.e., a term is evaluated to a value before being substituted. All the notions pertaining to the CBV calculus are defined as a variation of their CBN counterparts by factoring in the above value restriction. The Ctapp partial function is now assumed to return values instead of arbitrary terms.

Definition 50. *The* one-step CBV reduction *relation* $\rightarrow_v : \mathit{term} \rightarrow \mathit{term} \rightarrow \mathit{bool}$ *is defined inductively by rules similar to those of Def. 39, namely by the rules (AppL) and (AppR) from there (of course, with* $\rightarrow_v$ *replacing* $\rightarrow$*), together with:*

$$\frac{}{\mathsf{App}\ (\mathsf{Val}\ (\mathsf{Lm}\ y\ X))\ (\mathsf{Val}\ W)\ \rightarrow_v\ X\ [W\ /\ y]}\ (\beta)$$

$$\frac{\mathsf{Ctapp}\ c_1\ c_2\ =\ \mathsf{Some}\ V}{\mathsf{App}\ c_1\ c_2\ \rightarrow_v\ \mathsf{Val}\ V}\ (\delta) \qquad \frac{X\ \rightarrow_v\ X'}{\mathsf{Val}\ (\mathsf{Lm}\ y\ X)\ \rightarrow_v\ \mathsf{Val}\ (\mathsf{Lm}\ y\ X')}\ (\xi)$$

Highlighted above are the differences between the one-step CBV reduction and its CBN counterpart. In the (δ) and (ξ) rules the differences are inessential: One employs the value-to-term injection Val to account for the fact that Ctapp returns a value and that Lm-terms are values. The essential difference shows up in the (β) rule, which requires the righthand side of the redex to be a value. Similar differences are highlighted in the next definitions.

Definition 51. *The* one-step parallel CBV reduction *relation* $\Rightarrow_v : term \to term \to bool$ *is defined inductively by rules similar to those of Def. 41, namely by the rules (App) and (Refl) from there (with* $\Rightarrow_v$ *replacing* $\Rightarrow$*), together with:*

$$\frac{\mathsf{Ctapp}\ c_1\ c_2\ =\ \mathsf{Some}\ V}{\mathsf{App}\ c_1\ c_2 \Rightarrow_v \mathsf{Val}\ V}\ (\delta) \qquad \frac{X \Rightarrow X' \qquad Y \Rightarrow \mathsf{Val}\ V'}{\mathsf{App}\ (\mathsf{Val}\ (\mathsf{Lm}\ y\ X))\ Y \Rightarrow_v X'[V'\ /\ y]}\ (\beta)$$

$$\frac{X \Rightarrow_v X'}{\mathsf{Val}\ (\mathsf{Lm}\ y\ X) \Rightarrow_v \mathsf{Val}\ (\mathsf{Lm}\ y\ X')}\ (\xi)$$

Definition 52. *The* one-step left CBV reduction *relation* $\rightarrow\!\!\!\!\!\cdot_v : term \to term \to term$ *is defined inductively by rules similar to those of Def. 43, namely by the rule (AppL) from there (with* $\rightarrow\!\!\!\!\!\cdot_v$ *replacing* $\rightarrow\!\!\!\!\!\cdot$*), together with:*

$$\frac{\mathsf{Ctapp}\ c_1\ c_2\ =\ \mathsf{Some}\ V}{\mathsf{App}\ c_1\ c_2 \rightarrow\!\!\!\!\!\cdot_v \mathsf{Val}\ V}\ (\delta) \qquad \frac{}{\mathsf{App}\ (\mathsf{Val}\ (\mathsf{Lm}\ y\ X))\ (\mathsf{Val}\ W) \rightarrow\!\!\!\!\!\cdot_v X\ [W\ /\ y]}\ (\beta)$$

$$\frac{Y \rightarrow\!\!\!\!\!\cdot_v Y'}{\mathsf{App}\ (\mathsf{Val}\ V)\ Y \rightarrow\!\!\!\!\!\cdot_v \mathsf{App}\ (\mathsf{Val}\ V)\ Y'}(\mathsf{AppR})$$

Except for the above definitions, the CBV concepts are identical to those of the CBN concepts, *mutatis mutandis*, i.e., plugging in the above CBV basic relations instead of the CBN ones. These include the multi-step versions of the relations and the notions of complete parallel reduction operator and standard reduction sequence.

Moreover, the statements and proofs of the Church-Rosser and standardization theorems are essentially identical, *mutatis mutandis*. Like Plotkin has suggested in his informal development [60], the formal proofs could be easy adapted from CBN to CBV, obtaining:

Theorem 53. *Theorem 40 and Theorem 45 hold with the same statements, after replacing the CBN notions with their CBV counterparts.*

While the CBN and CBV formal developments are conceptually very similar, for the latter we employed our framework's infrastructure for a two-sorted syntax. To illustrate how this two-sorted syntax is handled by the framework, we show the definition of the CBV counterpart of cpred.

Definition 54. *The* CBV complete parallel reduction operator *of a term X (written* $\mathsf{cpred}_{term}\ X$*) and of a value V (written* $\mathsf{cpred}_{val}\ V$*) are the unique pair of functions satisfying:*

$$\mathsf{cpred}_{val}\ (\mathsf{Var}\ x) = \mathsf{Var}\ x \qquad \mathsf{cpred}_{val}\ (\mathsf{Ct}\ c) = \mathsf{Ct}\ c$$

$$\mathsf{cpred}_{term}\ (\mathsf{Val}\ V) = \mathsf{Val}\ (\mathsf{cpred}_{val}\ V) \qquad \mathsf{cpred}_{val}\ (\mathsf{Lm}\ y\ X) = \mathsf{Lm}\ y\ (\mathsf{cpred}_{term}\ X)$$

$$\mathsf{cpred}_{term}\ (\mathsf{App}\ X\ Y) = \begin{cases} \mathsf{Val}\,(\mathsf{cpred}_{val}\ V), \\ \quad \textit{if}\ (X,Y)\ \textit{have the form}\ (\mathsf{Val}\,(\mathsf{Ct}\ c_1), \mathsf{Val}\,(\mathsf{Ct}\ c_2)) \\ \quad \textit{with}\ \mathsf{Ctapp}\ c_1\ c_2 = \mathsf{Some}\ V \\ (\mathsf{cpred}_{term}\ Z)\ [(\mathsf{cpred}_{val}\ W)/y], \\ \quad \textit{if}\ (X,Y)\ \textit{have the form}\ (\mathsf{Val}\,(\mathsf{Lm}\ y\ Z), \mathsf{Val}\ W) \\ \mathsf{App}\ (\mathsf{cpred}_{term}\ X)\ (\mathsf{cpred}_{term}\ Y), \quad \textit{otherwise} \end{cases}$$

$$\mathsf{fresh}_{val}\ y\ V \ \textit{ implies }\ \mathsf{fresh}_{val}\ y\ (\mathsf{cpred}_{val}\ V)$$

$$\mathsf{fresh}_{term}\ y\ X \ \textit{ implies }\ \mathsf{fresh}_{term}\ y\ (\mathsf{cpred}_{term}\ X)$$

$$\mathsf{cpred}_{val}\ (V[z_1\ \wedge\ z_2]_{val}) = (\mathsf{cpred}_{val}\ V)[z_1\ \wedge\ z_2]_{val}$$

$$\mathsf{cpred}_{term}\ (X[z_1\ \wedge\ z_2]_{term}) = (\mathsf{cpred}_{term}\ X)[z_1\ \wedge\ z_2]_{term}$$

Similarly to the CBN case, this turns out to be a correct definition thanks to a two-sorted version of Prop. 38, that is, via exhibiting a two-sorted FSw model.

4.4 Overview of the Formalization and Lessons Learned

Our development is based on the general theory of syntax with bindings, which was presented in Chapter 3.The development has two parts.

The first part is the instantiation of the general theory to the two syntaxes, of λ-calculus and of λ-calculus with emphasized values, together with the transfer from a deep to a more shallow embedding (reported in Section 4.1). This is currently a completely routine, but very tedious process—it spans over more than 15000 lines of code (LOC) for each syntax. The reasons for this large size are the sheer number of stated theorems about constructors and substitution (more than 300 facts for the one-sorted syntax and more than 500 for the two-sorted syntax) and the many intermediate facts stated in the process of transferring the recursion theorems. Thanks to using a custom template for the instantiation, the whole process only took us two person-days. However, this is unreasonably long for a process that can be entirely automated.

The second part is the theory of CBN and CBV λ-calculus, culminating with the proofs of the Church-Rosser and Standardization theorems (reported in Sections 4.2 and 4.3). This is where our routine effort from the first part fully paid off. Thanks to our comprehensive collection of facts about substitution and freshness, we were able to focus almost entirely on formalizing the high-level ideas present in the informal proofs—notably in Plotkin's sketches of his elaborate proof development for the standardization theorem. On two occasions—for

defining Takahashi's complete parallel reduction operator and the number of variable occurrences needed in the Standardization proof development—our recursion principle allowed us to quickly get off the ground, in the second case also offering useful freshness and substitution lemmas needed later in the proof. Altogether, the second part consists of 5000 LOC (2000 for CBN and 3000 for CBV) and took us one person-month.

An exception to the above general phenomenon (of being able to focus on the high-level proof ideas) was the need to engage in the low-level task of proving custom constructor-directed inversion rules for our reduction relations—illustrated and motivated in the discussion leading to Lemma 4. This lemma is just one example of the several similar inversion rules we proved, corresponding to the inductive rules involving λ-abstraction in the reduction relations' definitions. These rules are essentially the binding-aware version of what Isabelle/HOL offers via the "inductive cases" command [78]. They seems to be generally useful in proof developments that involve inductively defined reductions but require structural induction over terms. The literature on formal reasoning seems to have overlooked the general usefulness of these rules; and state of the art definitional packages such as Nominal Isabelle [71, 74] do not attempt to infer them automatically.

Finally, our case study illustrates another interesting and apparently not uncommon phenomenon: that fresh structural induction on terms may be too weak in proofs, whereas depth-based induction in conjunction with fresh cases may do the job *while still enabling the use of Barendregt's convention*—as illustrated in our proof of Lemma 1.

In conclusion, we believe that our approach did a very good job handling some formalization task from the "real world". There are indeed some technical initialization issues, but we believe they can be resolved once and for all, at the "developers" level, and completely hidden from the users—especially in our second formalization, described starting from the next chapter. On the other hand, we believe the features of our framework (notably a rich built-in theory of substitution and operator-aware recursion) have enabled us to produce a fully formal, yet high-level presentation of these fundamental results.

Chapter 5

Intermezzo: More Bindings to be Captured

Although the POPLmark Challenge and its recursively defined binding structures from parts 1B and 2B have been openly among our motivations, our first framework does not easily capture its syntax. In that framework binding structures are modelled via abstractions (Section 3.1.1, Chapter 3) that bind *one* variable in *one* term at a time. We could have tried an encoding of the binding patterns from the challenge, by iterating the abstraction binder from our theory. Instead we decided to remain faithful to our generality purpose and changed some design features of the framework.

The inability of dealing gracefully with complex binders is not the only limitation of this first framework. Lack of modularity is also an issue: we have modelled the generic syntax by means of a binding signature, which fixes constructors and their arities, and precludes from any nesting or modularity features.

We have indeed captured infinitely branching terms and, in particular, these terms contain infinitely many free variables. On the other hand, we have limited ourselves to well-founded syntaxes.

In this chapter we change our design decisions for the framework, extending the theory to overcome the just mentioned limitations. In Section 5.1 we analyse and criticise what we have achieved up to now. Then in section 5.2 we build step by step our new formalization, based on *functors*. This way we obtain our second framework, which will be described in the next Chapter 6.

5.1 Critique of the First Framework

In Chapter 3 we have presented our first framework aimed at capturing the essence of syntax involving binding mechanisms and formalizing this in a general theory. Then with Chapter 4 we provided an instance of the theory, a formalized case study that could help understanding what we have achieved, what is good and what is missing in our development.

We believe that one of the major strengths of our theory is the exhaustive infrastructure it gives for its instances. The development is conducted at a general level, definitions are given and lemmas are proven independently of the particular syntax. We define alpha equivalence and the basic operators, the fundamental ones being freshness (or equivalently the set of

free variables of a term), swapping and substitution. We give results about these objects up to reasoning and definition principles suitable for dealing with (alpha-equated) terms with bindings. Then the theory can be instantiated to a particular syntax and all the infrastructure is made available and customized for it.

In the the framework from Chapter 3 we achieve this by defining an arbitrary syntax, via an instantiable binding signature—in the instantiation process constructors, binders and arities are specified.

We quotient the general terms to alpha equivalence so that, when deploying the formalization in concrete developments, there is no need to deal with it: as we have already pointed out, one of our aims is the usability of the formalized theory and we believe that this is better reached if two alpha-equivalent terms are treated as equal and still the variables, also when bound, are explicitly named.

In order to avoid complications arising from reasoning about alpha-equivalence classes, we need a collection of lemmas and properties that deal directly with these classes and do not involve their particular representatives; these are indeed provided in the formalization and we have dedicated the whole Subsection 3.2.4 of Chapter 3 to their description. These lemmas allow us to seal the abstraction barrier and the user to deal only with high level concepts and ignore alpha-equivalence. Moreover some of these properties are subtle and, when dealing with a particular syntax, the time spent to formulate and prove them would be wasted, since they can be proved once and for all, at a general level.

We have endowed our theory with useful reasoning and definition principles suitable for dealing with (alpha-equated) terms with bindings, described in Sections 3.4 and 3.3 from Chapter 3: Nominal's fresh induction [58, 76, 73], which allows the formal use of Barendregt's variable convention, Micheal Norrish's swapping-sensitive recursion principle [52], our own substitution-sensitive one and a combination of the two—overall a fair variety of them for the user to pick from.

Our theory can capture many-sorted binding syntaxes (Section 3.5); so among its possible instances we have, e.g., λ-calculus (also distinguishing the separate syntactic category of values, see Setion 4.1.2 from Chapter 4), FOL, process calculi, and System F.

Finally we have captured infinitely branching syntaxes, like CCS [51]: these allow for an infinite number of free variables in their terms; thus we were forced to go beyond finite support and exploit cardinality theory.

However this framework, also when confronted with what we declared as our aims (Chapter 1, Section 1.4), has some major drawbacks. For example, it deals with an infinite number of variables, but only if they are organized in recursively defined, infinitely-branching terms; namely, non-well-founded syntaxes with terms that have possibly-infinite depth, are not yet covered.

Our theory also has an overly rigid binding mechanism: our standalone abstractions (see Chapter 3, Sections 3.1 and 3.2) allow to bind in a term just *one* variable at a time. If it is true that this framework can capture many syntaxes with bindings, we have an important left-out: the motivational example of System $F_{<:}$ from the influential POPLmark challenge [7].

The encoding in our first framework of complex bindings, such as the recursively specified binding patterns of System $F_{<:}$, is not immediate, if possible at all.

Finally, it is important to remark that our theory is developed in the style of universal algebra, namely, it is based on binding signatures, see Subsection 3.5.1. With the signature, we fix our syntactic objects (organized in sorts, operation symbols, variables etc.) once and for all, not allowing for any kind of modularity. When formalizing mathematics in a proof assistant, any object, once defined, is exactly that one: it is not easy to transfer the reasoning to isomorphic structures, as instead is commonly done in pen-on-paper algebraic practice. For this reason, it is useful to define flexible structures from the beginning, so that they can be reused in a wider context, without the need of defining isomorphic ones. In our case, as already stated in Subsection 1.4, the correct property to go after is *modularity*. A syntax should be able to be arbitrarily nested in other syntaxes, just like it happens with free (i.e. with no bindings involved) datatypes: it can be the case that we have a list of trees and also a tree with lists at its nodes.

This kind of modularity and also infintely-deep structures are features that have been captured by the (co)datatype package of Isabelle/HOL [18], based on bounded natural functors (BNFs, see Subsections 2.2 and 2.3 from Chapter 2), which follows a compositional design [70] and provides flexible ways to nest types [16] and mix recursion with corecursion [13, 17]. However this work does not cover bindings.

Motivated by its modularity features, we start from the (co)datatype package development and model syntax with bindings with an approach based on functors—in particular on a refined version of BNFs. This approach has a main advantage: it is based on an already existing mechanism from the logical foundation, namely the dependency of type constructors on type variables. This will give more flexibility to our structures, which will have generic parameters as inputs, instead of variables picked from a fixed universe, thus allowing for the modular nesting property discussed above.

5.2 Towards an Abstract Notion of Binder

Motivated by the analysis from Section 5.1, we start here shaping up a theory of syntax, based on functors, that captures the essence of binding mechanisms. We start with the following simple observation: The literature so far has focused on binding notions relying on syntactic formats [64, 68, 75], as well as we have done with our previous framework (Chapter 3). In contrast, here we ask a semantic question: Can we provide an abstract, syntax-free axiomatization of binders?

5.2.1 Examples of Binders

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

remains free. Thus, in a λ-binder we distinguish two main components: the binding variable and the body (the term where this variable is to be bound).

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

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

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

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

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

5.2.2 Abstract Binder Types

Binders distinguish between binding variables and the other entities, typically terms, which could be either inside or outside the scope of the binder. Thus, we can think of a binder type as a type constructor $(\overline{\alpha},\overline{\tau})\,F$, with $m = \mathsf{len}\,(\overline{\alpha})$ and $n = \mathsf{len}\,(\overline{\tau})$, that takes as inputs

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

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

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

Section 5.2.1's examples are modeled as follows (writing α and τ instead of α_1 and τ_1 if $m = 1$):

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

For the "let" binder, the type constructor, $(\alpha,\tau_1,\tau_2)\,F$, needs to distinguish between the type of potential terms in the binder's scope, τ_2, and that of potential terms outside its scope, τ_1. This is necessary to describe the binder's structure accurately; but the actual terms corresponding to τ_1 and τ_2 will be allowed to be the same, as in $(\alpha,\tau,\tau)\,F$. One may wonder why the binder should care about potential terms that fall outside the scope of its binding variables. The answer is that this could lead to severe lack of precision, as argued by [64]. In the "parallel let" construct let $a_1 = t_1$ and $\ldots$ and $a_n = t_n$ in t, the terms t_i are outside the scope of the variables a_i, but they must be considered as inputs for "let" to ensure that the number of terms t_i matches the number of variables a_i.

It could be argued that our proposal constitutes yet another restrictive format. However, leaving F unspecified gives considerable flexibility compared with the syntactic approach. F can incorporate arbitrarily complex binders, including the datatype $\alpha\,pat$ needed for the POPLmark "pattern let." It can also accommodate unforeseen situations. Capturing the "parallel let" construct above rests on the observation that the structure of binding variables can be intertwined with that of the out-of-scope potential terms, which a syntactic format would need to anticipate explicitly. By contrast, with our modular semantic approach, it suffices to choose a suitable type constructor: $(\alpha,\tau_1,\tau_2)\,F = (\alpha \times \tau_1)\,list \times \tau_2$, with $\theta = \{(1,2)\}$. As another example, the type schemes in Hindley–Milner type inference [49] are assumed to have all the schematic type variables bound at the top level, but not in a particular order. A permutative type such as that of finite sets can be used: $(\alpha,\tau)\,F = \alpha\,fset \times \tau$, with $\theta = \{(1,1)\}$. In summary, this is our first proposal:

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

As it stands, the proposal is not particularly impressive. For all its generality, it tells us nothing about how to construct actual terms with bindings or how to reason about them. Let us look closer at our proposal and try to improve it. By modeling "binder types" not as mere types but as type constructors, we can distinguish between the binder's structure and the variables and potential terms that populate it—that is, between shape and content. This

follows our intuition of BNFs (Section 2.2). And indeed, all the type constructors we used in examples of binders seem to be BNFs. So we can be more specific:

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

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

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

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

To address this issue, we refine the notion of BNF by restricting, on selected inputs, all conditions involving the map function, including the functoriality, to injective functions only. For reasons of symmetry, we take the more drastic measure of restricting to *bijective* functions only, which additionally have the same domain and codomain—we call these endobijections, but we could have called them permutations, as is common practice in Group Theory, and they are a proper generalization of the Nominal swapping (see Section 2.4, Chapter 2, and Section 7.1 from Chapter 7). In other words, all the conditions of the BNF definition (Definition 4) remain the same, except that on some of the inputs (which are marked as "restricted") they are further conditioned by endobijectiveness assumptions about the corresponding functions. For our type constructor $(\overline{\alpha},\overline{\tau})\,F$, the restricted inputs will be $\overline{\alpha}$, which means that F will behave like a functor with respect to endobijections $\overline{f}:\overline{\alpha}\to\overline{\alpha}$ and arbitrary functions $\overline{g}:\overline{\tau}\to\overline{\tau'}$. All our examples involving multiple variable bindings, including $(\alpha,\tau)\,F=\{(a,b):\alpha\times\alpha\mid a\neq b\}\times\tau$, belong to this category. We call this refined notion *map-restricted BNF* (*MRBNF*, or $\overline{\alpha}$-*MRBNF*).

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

MRBNFs remain general while offering a sound mechanism for renaming bound variables. To validate this proposal, we ask two questions, which will be answered in Sections 6.1 and 6.2 of the next Chapter: (1) How can nonrepetitive MRBNFs be constructed from possibly repetitive ones? (2) How can MRBNFs be used to define and reason about actual terms with bindings and their fundamental operators?

Starting from the answers to (1) and (2), our theory will evolve and lead to a new framework for syntax with bindings. The next Chapter 6 is entirely dedicated to this development, with an overview of the main and novel features of the framework.

Chapter 6

Bindings are Functors

In what follows we present our functor-based framework for specifying and reasoning about syntax with bindings. Abstract binder types are modeled using a universe of functors on sets (the just introduced MRBNFs, Section 5.2 of the previous Chapter), subject to a number of operations that can be used to construct complex binding patterns and binding-aware datatypes, including non-well-founded and infinitely branching types, in a modular fashion. Despite not committing to any syntactic format, the framework is "concrete" enough to provide definitions of the fundamental operators on terms (free variables, alpha-equivalence, and capture-avoiding substitution) and reasoning and definition principles. This work is compatible with classical higher-order logic and has been formalized in the proof assistant Isabelle/HOL.

The goal of this framework is to further systematize and simplify the task of constructing and reasoning about variable binding and variable substitution, namely the operations of binding variables into terms and of replacing them with other variables or terms in a well-scoped fashion.

To improve on our previous work and also on the state of art, we believe that the missing ingredient is a semantics-based understanding of the binding and substitution mechanisms, as opposed to ever more general syntactic formats. Here we aim at identifying the fundamental laws that guide binding and substitution and expressing them in a syntax-free manner. Bindings can be abstractly understood in a world of *shapes* and of *content* that fills the shapes. A binder puts together one or more variables in a suitable shape that is connected with the body through common content. Variable renaming and replacement, which give rise to alpha-equivalence and capture-free substitution, amount to replacing content while leaving the shape unchanged.

We here perfect the definition of MRBNFs, the class of functors on sets that we have introduced in Section 5.2 of Chapter 5, so that they can express the action of arbitrarily complex binders while supporting the construction of the key operations involving syntax with bindings—such as free variables, alpha-equivalence, and substitution—and the proof of their fundamental properties. This class of functors will subsume a large number of results for a variety of syntactic formats. Another gain will be *modularity*: complex binding patterns can be developed separately and placed in larger contexts in a manner that guarantees correct scoping and produces correct definitions of alpha-equivalence and substitution. The abstract perspective also will clarify the acquisition of fresh variables, allowing us to go

beyond finitary syntax. Our theory gracefully extends the scope of techniques for reasoning about bindings to infinitely branching and non-well-founded terms. Finally we will equip our binding-aware (co)datatypes with a powerful arsenal of reasoning and definitional principles that follow the (co)datatypes' structure.

As said this work targets the Isabelle/HOL proof assistant, a popular implementation of higher-order logic (HOL), and extends Isabelle's framework of BNFs (see Chapter 2), turning them into binding-aware entities (see Chapter 5).

In this chapter we reprise the questions left open at the end of the previous Chapter 5. We first analyze how complex binder types can be constructed in a uniform way (Section 6.1) and then we try to apply these ideas to define terms with bindings (Section 6.2): Is our abstraction "concrete" enough to support all the constructions and properties typically associated with syntax with bindings, including binding-aware (co)datatypes? And can the constructions be performed in a modular fashion, allowing previous constructions to be reused for new ones? After undergoing a few more refinements, our binders pass the test of properly handling not only bound variables, but also free variables. This suggests that we have identified a "sweet spot" between the assumptions and the guarantees involved in the construction of (co)datatypes. We synthesize the process we have been through, by giving a formal definition of the class of functors we have identified (Section 6.3).

We equip our binding-aware (co)datatypes with a powerful arsenal of reasoning and definitional principles that follow the (co)datatypes' structure. We generalize fresh induction in the style of nominal logic and propose new coinduction principles (Section 6.4). Moreover, we provide (co)recursion principles that improve on the state of the art even in the case of simple binders, and we validate the constructions by characterizing them up to isomorphism (Section 6.5). We expand on this by giving different versions of the definition principles—which can be more or less useful according to the application where they are invoked—(Section 6.6) and by confronting ours with other recursion principles from literature, in the simple case of λ-calculus so that we can focus on the features of the recursors themselves and not on the complexity of the binding syntax (Section 6.7).

A slightly restricted version of the definitions and theorems from this chapter have been mechanically checked in Isabelle/HOL (Section 6.8). Our formal development is publicly available at `https://sites.google.com/view/lorgheri/research`.

6.1 Constructing Nonrepetitive Map-Restricted BNFs

For MRBNFs' predecessors, BNFs (Sections 2.2 and 2.3, Chapter 2, and Section A.1 from the Appendix for a further insight), the question of constructibility has a most satisfactory answer: We can start with the basic BNFs and repeatedly apply composition, least fixpoint (datatype), and greatest fixpoint (codatatype). Any BNF also constitutes a map-restricted BNF, and it is in principle possible to lift the map-restricted arguments through fixpoints on the nonrestricted type arguments. However, nonrepetitiveness—the defining feature for MRBNFs, see Subsection 5.2.2 of Chapter 5—is not closed under fixpoints. Thus, if $(\alpha,\tau)\,F$ is a nonrepetitive α-MRBNF, the (least or greatest) fixpoint $\alpha\,T$ specified as $(\alpha,\alpha\,T)\,F \simeq \alpha\,T$

will be an α-MRBNF, but not necessarily a nonrepetitive one. For example, $(\alpha,\tau)\,F = \mathit{unit} + \alpha \times \tau$ is a nonrepetitive α-MRBNF (because α atoms cannot occur multiple times in members of $(\alpha,\tau)\,F$), but its least and greatest fixpoints are α *list* and α *llist*, the usual types of list and lazy lists—which are repetitive α-MRBNFs because lists, whether lazy or otherwise, may contain repeated elements.

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

Can we express nonrepetitiveness generally for any BNF? A first idea is to rely on the cardinality of sets of atoms. For the $\alpha\,\mathit{list}$ BNF, the nonrepetitive items are those lists $as = [a_1,\ldots,a_n]$ containing precisely n distinct elements $a_1,\ldots,a_n$—or, equivalently, having a maximal cardinality of atoms, $|\mathsf{set}_{\mathit{list}}\ x|$, among the lists of a given length. This idea can be generalized to arbitrary BNFs $\alpha\,F$ by observing that the length of a list is essentially its "shape." So we can define the fact that two members x, x' of $\alpha\,F$ have the same shape, written $\mathsf{sameShape}_F\ x\ x'$, to mean that $\mathsf{rel}_F\ \top\ x\ x'$ holds, where $\top : \alpha \to \alpha \to \mathit{bool}$ is the vacuously true relation that ignores the content. Indeed, recall from Section 2.2 that the main intuition behind a BNF relator rel_F is that $\mathsf{rel}_F\ R\ x\ x'$ holds if and only if (1) x and x' have the same shape and (2) their atoms are positionwise related by R. Condition (2) is trivially satisfied for $R := \top$. For lists and lazy lists, $\mathsf{sameShape}$ means "same length," and for various kinds of trees it means that the two trees become identical if we erase their content (e.g., the labels on their nodes or their branches).

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

In general, for $x : \alpha\,F$, we define $\mathsf{nonrep}_F\ x$ so that for all x' such that $\mathsf{sameShape}_F\ x\ x'$, there exists a function f that maps x to x', so that $x' = \mathsf{map}_F\ f\ x$. This gives the correct result for lists, lazy lists, trees, and in general for any combination of (co)datatypes where each atom has a fixed position in the shape—i.e., for strong BNFs (Section 2.3, Chapter 2). We

can define the corresponding nonrepetitive MRBNF:

Theorem 55. If αF is a strong BNF and nonrep_F is nonempty, then $\alpha G = \{x : \alpha F \mid \mathsf{nonrep}_F\, x\}$, in conjunction with the corresponding restrictions of map_F, set_F, rel_F, and bd_F, forms an MRBNF.

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

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

For BNFs that are not strong such as finite sets and multisets, the nonrepetitiveness construction tends to give counterintuitive results, e.g., no finite set but the empty one is nonrepetitive. However, these structures do not require any nonrepetitiveness revision, being useful as they are. For example, Hindley–Milner type schemes bind finite sets of variables.

6.2 Defining Terms with Bindings via Map-Restricted BNFs

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

Before we can define actual terms, we must prepare for a dual phenomenon to the binding of variables: The terms must be allowed to have *free* variables in the first place, before these can be bound. Thus, in addition to binding mechanisms, we need mechanisms to inject free variables into potential terms. This can be achieved by upgrading F: Instead of $(\overline{\alpha}, \overline{\tau})\, F$, we work with an $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F$, where we consider an additional vector of inputs $\overline{\beta}$ representing the types of (injected) free variables. It is natural to consider the same types of variables as possibly free and possibly bound. Hence, we will assume $\mathsf{len}\,(\overline{\beta}) = \mathsf{len}\,(\overline{\alpha}) = m$ and use $(\overline{\alpha}, \overline{\alpha}, \overline{\tau})\, F$ when defining actual terms. Nevertheless, it is important to allow F to distinguish between the two kinds of inputs.

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

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

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

Example 56. Consider the syntax of the λ-calculus, where the collection $\alpha\,T$ of terms t with variables in α are defined by the grammar $t ::= \mathsf{Var}\,a \mid \lambda a.t \mid t\,t$. A term is either a variable, an abstraction, or an application. This is supported in our abstract scheme by taking $m = 1$, $n = 2$, $\theta = \{(1,1)\}$, and $(\beta,\alpha,\tau_1,\tau_2)\,F = \beta + \alpha \times \tau_1 + \tau_2 \times \tau_2$. The resulting $\alpha\,T$ satisfies the recursive equation $\alpha\,T \simeq \alpha + \alpha \times \alpha\,T + \alpha\,T \times \alpha\,T$. Not visible in this equation is how F distinguishes

- between the free-variable type β and the binding-variable type α—a distinction that ensures that the occurrence of α as the first summand stands for an injection of free variables, whereas the first occurrence of α in the second summand stands for binding variables; and
- between two different types of potential terms, τ_1 and τ_2—a distinction that ensures, via θ, that α's binding powers extend to the occurrence of $\alpha\,T$ in the second summand but not to the two occurrences in the third summand.

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

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

Example 57. Consider the extension of the λ-calculus syntax with "parallel let" binders:

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

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

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

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

We have assumed F to be a full functor on $\overline{\tau}$, and to be a functor on $\overline{\alpha}$ with respect to endobijections. But how should it behave on $\overline{\beta}$? A natural answer would be to require full functoriality, because the nonrepetitiveness condition that compelled us to restrict F's behavior on binding variable inputs seems unnecessary here: There is no apparent need to

avoid repeated occurrences of free variables. In fact, the central operation of substitution introduces repetitions, e.g., by substituting a for a' while a was already free in the term. So for now, we will assume full functoriality on $\overline{\beta}$.

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

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

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

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

Example 58. Consider a variation of the λ-calculus syntax where abstractions simultaneously bind two variables in two terms, given by the grammar $t ::= \mathsf{Var}\ a \mid \lambda(a,b).(t_1,t_2)$, with the usual requirement that the variables a and b are distinct. We can take $\theta = \{(1,1)\}$ and $(\beta,\alpha,\tau)F = \beta + ((\alpha \times \alpha)\ @\ \alpha) \times \tau \times \tau$. We write Inl and Inr for the left and right injections of the components into sums types: $\mathsf{Inl} : \beta \to (\beta,\alpha,\tau)F$ and $\mathsf{Inr} : ((\alpha \times \alpha)\ @\ \alpha) \times \tau \times \tau \to (\beta,\alpha,\tau)F$.

6.2.1 Free Variables

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

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

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

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

following rules:

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

In the context of our running Example 58, let us assume from now on that $a, b, c, d, \ldots$ are mutually distinct variables. First, consider the term $t = \mathsf{Var}\ c$. It can be written as $\mathsf{ctor}\ x$, where $x = \mathsf{Inl}\ c$. Therefore, $\mathsf{topFree}\ x = \{c\}$, $\mathsf{topBind}\ x = \varnothing$, and $\mathsf{rec}\ x = \varnothing$. (We omit the indices since $m = n = 1$.) Thus, t has c as its single top-free variable, has no top-binding variables, and has no recursive components. Moreover, t has c as its single free variable: Applying the first rule in the definition of FVars, we infer $c \in \mathsf{FVars}\,(\mathsf{ctor}\ x)$ from $c \in \mathsf{topFree}\, x$.

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

6.2.2 Alpha-Equivalence

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

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

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

Intuitively, two terms t_1 and t_2 should be alpha-equivalent if they are the same up to a renaming of their bound variables. More precisely, the situation is as follows (Fig. 6.1):

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

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

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

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

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

Returning to the context of Example 58, we first note that $\mathsf{Var}\ a \equiv \mathsf{Var}\ a$ for every a. This is shown by applying the definitional clause of $\equiv$ with the identity for f. The first hypothesis can be immediately verified: Since $\mathsf{Var}\ a$ has no top-binding variables or recursive components, only the condition concerning the top-free variables needs to be checked: $a = a$.

Now consider the terms $t_1 = \lambda(a,b).\,(\mathsf{Var}\ a, \mathsf{Var}\ c)$ and $t_2 = \lambda(b,a).\,(\mathsf{Var}\ b, \mathsf{Var}\ c)$, which can be written as $\mathsf{ctor}\ x_1$ and $\mathsf{ctor}\ x_2$, where $x_1 = \mathsf{Inr}\,((a,b), (\mathsf{Var}\ a, \mathsf{Var}\ c))$ and $x_2 = \mathsf{Inr}\,((b, a), (\mathsf{Var}\ b, \mathsf{Var}\ c))$. We can prove these to be alpha-equivalent by taking f to swap a and b (i.e., send a to b and b to a) and leave all the other variables, including c, unchanged. Verifying the first hypothesis of $\equiv$'s definitional clause amounts to the following: Concerning positionwise equality of the top-free variables, there is nothing to check (since t_1 and t_2 have none); concerning the top-binding variables, we must check that $f\ a = b$ and $f\ b = a$; concerning the recursive components, we must check that $\mathsf{map}_T\, f\,(\mathsf{Var}\ a) \equiv \mathsf{Var}\ b$ and $\mathsf{map}_T\, f\,(\mathsf{Var}\ c) \equiv \mathsf{Var}\ c$. Applying the definition of map_T, the last equivalences become $\mathsf{Var}\ (f\ a) \equiv \mathsf{Var}\ b$—i.e., $\mathsf{Var}\ b \equiv \mathsf{Var}\ b$ and $\mathsf{Var}\ c \equiv \mathsf{Var}\ c$. Finally, verifying the second hypothesis, $\mathsf{cond}\,(f)$, amounts to checking that f is bijective and that f is the identity on all variables in the set $(\bigcup_{t_1 \in \{\mathsf{Var}\ a, \mathsf{Var}\ c\}} \mathsf{FVars}\ t_1) \setminus \{a,b\} = \{a,c\} \setminus \{a,b\} = \{c\}$—i.e., f sends c to c.

The approach above is one of the several possible choices to define alpha-equivalence. We could pose stricter conditions on the f_i, allowing them to change *only* the top-binding variables, whereas we also allow it to change some other (nonfree) variables occurring in the components. In the context of the running example, if the left term is $\lambda(a,b).\,(\mathsf{Var}\ c, \lambda(c,d).\,(\mathsf{Var}\ c, \mathsf{Var}\ d))$, then both solutions allow f to change a and b, and do not allow it to change c. Our definition additionally allows f to change d. Another alternative would consist in a symmetric formulation: Rather than renaming variables of the left term only, we could rename the variables of both terms to a third term, whose binding variables are all distinct from those of the first two. All these variants of introducing alpha-equivalence have different virtues in terms of the ease or elegance of proving various basic properties, but in the end produce the same concept.

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

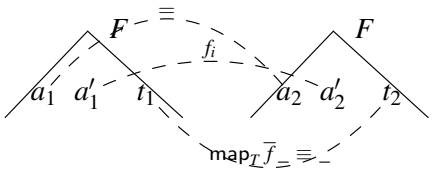

FIGURE 6.1: Alpha-equivalence

Theorem 59. Alpha-equivalence is an equivalence and is compatible with

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

6.2.3 Alpha-Quotiented Terms

Exploiting Theorem 59, we can define the quotient $\overline{\alpha}\,\boldsymbol{T} = (\overline{\alpha}\,T)/\equiv$, and lift the relevant functions to $\overline{\alpha}\,\boldsymbol{T}$. Using overloaded notation, we obtain the constructor $\mathsf{ctor} : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\,\boldsymbol{T}]^n)\,F \to \overline{\alpha}\,\boldsymbol{T}$ and the operators $\mathsf{FVars} : \overline{\alpha}\,\boldsymbol{T} \to \alpha\ \mathit{set}$ and $\mathsf{map}_{\boldsymbol{T}} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\,\boldsymbol{T} \to \overline{\alpha}\,\boldsymbol{T}$.

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

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

Proposition 60. Given $x_1, x_2 : (\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\,\boldsymbol{T}]^n)\,F$, the following are equivalent:

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

Thus, $\overline{\alpha}\,\boldsymbol{T}$ was defined by fixpoint construction framed by F followed by a quotienting construction to a notion of alpha-equivalence determined by the binding dispatcher θ, which for $(\overline{\beta}, \overline{\alpha}, \overline{\tau})\,F$ states what $\overline{\alpha}$ binds in $\overline{\tau}$. We will use the notation

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

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

The presence of the operators ctor, FVars, and $\mathsf{map}_{\boldsymbol{T}}$ on quotiented terms offers them a large degree of independence from the underlying terms. Indeed, we are here being faithful

to one of our main design goals, that was already present in the development we presented in Chapter 3, namely, developing an abstraction layer for reasoning about the type $\overline{\alpha}\,\boldsymbol{T}$ that allows us to forget about $\overline{\alpha}\,T$.

Terminology From now on, we will adhere to the following convention: We will call the members of $\overline{\alpha}\,\boldsymbol{T}$ *terms* and the members of the underlying type $\overline{\alpha}\,T$ *raw terms*[1].

6.2.4 Infinitely Branching Terms

Also in this second functor-based formalization of our theory, our constructions do not commit to the finite branching of terms, thus capturing situations required by some calculi [51] and logics [40, 35] (see also Section 3.1.5 from Chapter 3).

Example 61. A simplified version of the syntax for processes in the calculus of communicating systems (CCS) [51] is the following, where c ranges over a fixed type C of channels, e over a type $\alpha\,E$ of arithmetic expressions with variables in α, and J over subsets of a fixed, possibly infinite type I of indices:

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

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

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

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

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

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

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

6.2.5 Substitution

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

[1] Here we use again again, as in Chapter 3 the word "raw" to indicate those items not yet quotiented by alpha-equivalence.

case of simultaneous substitution, defined as $t[a/b] = \mathsf{sub}\ f_{a,b}\ t$, where $f_{a,b}$ sends b to a and all other variables to themselves.

A candidate for sub that suggests itself is $\mathsf{map}_{\boldsymbol{T}}$, which $\overline{\alpha}\,\boldsymbol{T}$ inherits from $\overline{\alpha}\,T$. However, this operator is not suitable, since it only works with bijections f_i. The fundamental desired property of sub concerns its recursive behavior on terms of the form ctor x:

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

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

The first of these two conditions, $\mathsf{topBind}_i\, x \cap \mathsf{FVars}_i\,(\mathsf{ctor}\ x) = \varnothing$, which we will abbreviate as $\mathsf{noClash}_i\, x$, follows a general hygiene principle that is independent of the notion of substitution (and which will be observed by all our proof and definition principles): We never consider variables that appear both bound and free in the same term. The condition is vacuously true for syntaxes in which no constructor simultaneously binds variables and introduces free variables.

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

6.2.6 Acquiring Enough Fresh Variables

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

Here is an illustration of the above phenomenon, in a case that goes beyond finite support. Using the notations of Example 61, let $J = I = \mathit{nat}$ and let t be the process term $c\,(a_0).\sum_{i \in \mathit{nat}} \overline{c}\,a_i.0$, where the a_i's are all distinct variables. Then what is the term sub $f\ t$, where f is the function that sends a_0 to a_0 and any a_{i+1} to a_i? Clearly, sub $f\ t$ should start with $c\,(a)$ for some variable a. However, a cannot be an a_i, since a_i must be free in sub $f\ t$ (due to a_{i+1} being free in t). But what if the a_i's exhaust all the available variables? To avoid this scenario, we ask that the support of f be smaller than the set of available variables, which is true for example (1) if the support is finite and there are infinitely many variables,

or (2) if the support is countable and there are uncountably many variables. In the first case, f is not deemed suitable for substitution. In the second case, there exists a fresh variable x_π with the help of which we can express $\mathsf{tsub}\ f\ t$ as $c\,(a_\pi).\sum_{i\in nat}\overline{c}\,a'_i.0$, where $a'_0 = a_\pi$ and $a'_{i+1} = a_i$. By keeping the type $\alpha\,\boldsymbol{T}$ of terms polymorphic in the type α of variables, we can avoid committing to a specific scenario. This contributes to modularity: When using $\alpha\,\boldsymbol{T}$ as part of a larger (co)datatype (perhaps defined as a $\boldsymbol{T}$-nested fixed point), $\boldsymbol{T}$ will be able to export collections of variables of any required size. (See also Section 6.2.9.)

The above solution seems to require needlessly many variables when F is finitary—i.e., $\mathsf{bd}_F = \aleph_0$ (the countable cardinal). This is the case for all finitely branching datatypes. With our approach, we would need α to be uncountable, even though countably infinitely many variables would suffice. It could be argued that variable countability is unimportant. Indeed, some textbooks assume only "an infinite supply of variables," without mentioning countability. But countability becomes important when we consider practical aspects such as executability. Therefore, it is worth salvaging countability if we can. It turns out that we can do that with a little insight from the theory of cardinals. We note that the crucial property that $|\mathsf{FVars}_i\ t| < |\alpha_i|$ for all $i \in [m]$ and $t : \overline{\alpha}\,\boldsymbol{T}$ can be achieved using the nonstrict inequality $\mathsf{bd}_F \leq |\alpha_i|$ if $|\alpha_i|$ is a regular cardinal—again regularity proves itself to be the right property to exploit in this context, see 3.2.3.

Theorem 62. There exists a (polymorphic) function $\mathsf{sub} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\,\boldsymbol{T} \to \overline{\alpha}\,\boldsymbol{T}$ satisfying $(*)$ for all α_i and f_i if $|\alpha_i|$ is regular, $\mathsf{bd}_F \leq |\alpha_i|$, and $|\mathsf{supp}\ f_i| < |\alpha_i|$.

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

6.2.7 Term-for-Variable Substitution

So far, we have only discussed variable-for-variable substitutions. Often it is necessary to perform a term-for-variable substitution, in a capture-avoiding fashion. In the λ-calculus, we could substitute $\lambda c.\mathsf{Var}\ a$ for b in $\lambda a.\,(\mathsf{Var}\ a)\ (\mathsf{Var}\ b)$, yielding (after a renaming which does not affect alpha-equivalence) $\lambda a'.\,(\mathsf{Var}\ a')\,(\lambda c.\mathsf{Var}\ a)$. However, not all syntaxes with bindings allow substituting terms for variables. Process terms in the π-calculus [50] contain channel variables (names), which can be substituted by other channel variables but not by processes.

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

We can express such situations abstractly in our framework, by requiring that the framing MRBNF $(\overline{\beta},\overline{\alpha},\overline{\tau})\,F$ accommodate such embeddings. We fix $I \subseteq [m]$ and assume injective natural transformations $\eta_i : \beta_i \to (\overline{\beta},\overline{\alpha},\overline{\tau})\,F$ for $i \in I$ such that $\mathsf{set}_i^F\ (\eta_i\ a) = \{a\}$. Moreover, we assume that η_i is the only source of variables in F, by requiring that $\mathsf{set}_i^F\ x = \varnothing$ for every x that is not in the image of η_i. The injections of variables into terms, $\mathsf{Var}_i : \alpha_i \to \overline{\alpha}\,\boldsymbol{T}$, are

defined as $\mathsf{Var}_i = \mathsf{ctor} \circ \eta_i$. For the syntax of our running Example 58, where $(\beta,\alpha,\tau)\,F = \beta + ((\alpha \times \alpha)\ @\ \alpha) \times \tau \times \tau$, we have that $\eta : \beta \to (\beta,\alpha,\tau)\,F$ is the injection of the leftmost summand.

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

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

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

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

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

Theorem 63. There exists a (polymorphic) function $\mathsf{tsub} : (\prod_{i\in I}(\alpha_i \to \overline{\alpha}\,\boldsymbol{T})) \to \overline{\alpha}\,\boldsymbol{T}$ satisfying $(**)$ for all α_i and f_i in case $|\alpha_i|$ is regular, $\mathsf{bd}_F \leq |\alpha_i|$ and $|\mathsf{supp}\ f_i| < |\alpha_i|$.

6.2.8 Non-Well-Founded Terms

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

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

To this end, raw terms are defined as a greatest fixpoint: $\overline{\alpha}\,T \simeq^{\infty} (\overline{\alpha},\overline{\alpha},[\overline{\alpha}\,T]^n)\,F$. Then alpha-equivalence $\equiv : \overline{\alpha}\,T \to \overline{\alpha}\,T \to \mathit{bool}$ is defined by the same rules as in Section 6.2.2, but employing a coinductive (greatest fixpoint) interpretation. On the other hand, the free-variable operator is still defined inductively, by the same rules as in Section 6.2.1.

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

This inductive–coinductive asymmetry seems to stand in the way of a duality principle, which would allow us to reuse, or at least copy, the proofs above to cover non-well-founded terms. Fortunately, there is a way to restore the symmetry. On well-founded terms, $\equiv$ could have been equivalently defined coinductively. This is because the fixpoint operator $\mathsf{Op}_{\equiv} : (\overline{\alpha}\,T \to \overline{\alpha}\,T \to \mathit{bool}) \to (\overline{\alpha}\,T \to \overline{\alpha}\,T \to \mathit{bool})$ underlying the definition of $\equiv$ has a unique fixpoint: $(\equiv) = \mathsf{lfp}\ \mathsf{Op}_{\equiv} = \mathsf{gfp}\ \mathsf{Op}_{\equiv}$. In addition, the recursive definition of map_T on well-founded terms in Section 6.2.2 has an identical formulation for non-well-founded terms,

although it has different, corecursive justification.

As a result, many properties concerning the constructor, alpha-equivalence, free variables, the map function, and their combination on raw terms, including Theorem 59, which justifies the construction of $\overline{\alpha}\,\boldsymbol{T}$, can be proved in exactly the same way. All the theorems shown in Sections 6.2.1 to 6.2.7 hold for non-well-founded terms as well, with identical formulations. In particular, our solution to allow infinite support also applies to non-well-founded terms, which is crucial given that infinite terms rarely have finite support.

6.2.9 Modularity Considerations

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

Theorem 64. $\overline{\alpha}\,\boldsymbol{T}$ is an $\overline{\alpha}$-MRBNF with map function $\mathsf{map}_{\boldsymbol{T}}$ and set functions FVars_i.

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

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

Example 65. Consider the syntax of System F types, $\sigma ::= \mathsf{TyVar}\;a \mid \forall a.\,\sigma \mid \sigma \to \sigma$, assumed to be quotiented by the alpha-equivalence standardly induced by the $\forall$ binders. In our framework, this is modeled as $\alpha\,\boldsymbol{T} \simeq_\theta (\alpha,\alpha,\alpha\,\boldsymbol{T},\alpha\,\boldsymbol{T})\,F$, where $\theta = \{(1,1)\}$ and $(\beta,\alpha,\tau_1,\tau_2)$ $F = \beta + \alpha \times \tau_1 + \tau_2 \times \tau_2$. Now consider the syntax of System F terms, writing a' for term variables:

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

This should be expressed as $\overline{\alpha}\,\boldsymbol{T}' \simeq_\theta (\overline{\alpha},\overline{\alpha},[\alpha\,\boldsymbol{T}']^3)\,F'$, where $\theta = \{(1,1),(2,2)\}$ and $(\overline{\beta},\overline{\alpha},\overline{\tau})\,F'$ $= \beta_2 + \alpha_1 \times \tau_1 + \alpha_2 \times \beta_1\,\boldsymbol{T} \times \tau_2 + \tau_3 \times \beta_1\,\boldsymbol{T} + \tau_3 \times \tau_3$ with $\mathsf{len}\,(\overline{\beta}) = \mathsf{len}\,(\overline{\alpha}) = 2$ and $\mathsf{len}\,(\overline{\tau}) = 3$. Indeed, this would give the overall fixpoint equation

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

In this scheme, α_1 stores the System F type variables, and α_2 stores the System F term variables. As usual, this isomorphism is considered up to the alpha-equivalence induced by θ, which tells us that in the second summand α_1 binds in its neighboring $\overline{\alpha}\,\boldsymbol{T}'$, and in the third

summand α_2 binds in its neighboring $\overline{\alpha}\,\boldsymbol{T}'$. Note that System F type variables (represented by α_1) appear as binding in the second summand and as free (as part of $\alpha_1\,\boldsymbol{T}$) in the third summand. However, the definition of $\overline{\alpha}\,\boldsymbol{T}'$ is not possible; due to the presence of $\beta_1\,\boldsymbol{T}$ as a component, $(\overline{\beta},\overline{\alpha},\overline{\tau})\,F'$ is not an $\overline{\alpha}$-restricted MRBNF, but is additionally map-restricted on β_1.

The above problem would vanish if $\boldsymbol{T}$ were a full functor (with respect to arbitrary functions). However, the map function's $\mathsf{map}_{\boldsymbol{T}}$ restriction to endobijections is quite fundamental: Its definition is based on the low-level map_T on raw terms, which preserves alpha-equivalence only if applied to endoinjections or endobijections. This phenomenon is well known in nominal logic, and is a main reason for this logic's focus on the swapping operator (as we pointed out, considering swapping for finitary objects is essentially the same as considering endobijections): Swapping a and b respects alpha-equivalence (e.g., starting with $\lambda a.\,a\,b \equiv \lambda c.\,c\,b$ we obtain $\lambda b.\,b\,a \equiv \lambda c.\,c\,a$), whereas substituting a for b (in a capturing fashion) does not (e.g., starting with the same terms as above we obtain $\lambda a.\,a\,a \not\equiv \lambda c.\,c\,a$).

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

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

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

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

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

Thus, we will require that $(\overline{\beta},\overline{\alpha},\overline{\tau})\,F$ be a functor with respect to small-support endofunctions on $\overline{\beta}$, with respect to small-support endobijections on $\overline{\alpha}$, and with respect to arbitrary functions on $\overline{\tau}$. Full functoriality on $\overline{\tau}$ is necessary to solve the fixpoint equations that define the (co)datatypes. To clearly indicate this refined classification of its inputs, we will call $(\overline{\beta},\overline{\alpha},\overline{\tau})\,F$ a $\overline{\beta}$-free $\overline{\alpha}$-binding MRBNF, where we omit "$\overline{\beta}$-free" or "$\overline{\alpha}$-binding" if the corresponding vector $\overline{\beta}$ or $\overline{\alpha}$ is empty.

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

Theorem 66. $\overline{\alpha}\,\boldsymbol{T}$ is an $\overline{\alpha}$-free MRBNF with map function sub and set functions FVars_i.

Now we can give the complete definition of our just developed map-restricted bounded natural functors, to which the next section is dedicated. In the subsequent two sections, we complete our stated goal of offering the terms with bindings an abstraction layer, consisting of reasoning and definitional principles, that insulates them completely from the low-level details of raw terms.

6.3 Full Definition of Map-Restricted Bounded Natural Functors

Section 5.2, from Chapter 5, and 6.1 and 6.2, from this chapter, develop the notion of MRBNF through a sequence of refinements. While it can be inferred from the refinements, it may also be useful to list the end product as a single definition.

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

- F is an m_1+m_2+n-ary type constructor;
- bd_F is an infinite cardinal number;
- $\mathsf{bd}_F \leq |\beta_i|$ and $|\beta_i|$ is a regular cardinal for all $i \in [m_1]$;
- $\mathsf{bd}_F \leq |\alpha_i|$ and $|\alpha_i|$ is a regular cardinal for all $i \in [m_2]$;
- $\mathsf{map}_F : (\beta_1 \to \beta_1) \to \cdots \to (\beta_{m_1} \to \beta_{m_1}) \to (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_{m_2} \to \alpha_{m_2}) \to (\tau_1 \to \tau'_1) \to \cdots \to (\tau_n \to \tau'_n) \to (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to (\overline{\beta}, \overline{\alpha}, \overline{\tau'})\, F$;
- $\mathsf{set}^i_F : (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to \beta_i\ \mathit{set}$ for $i \in [m_1]$;
- $\mathsf{set}^{m_1+i}_F : (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to \alpha_i\ \mathit{set}$ for $i \in [m_2]$;
- $\mathsf{set}^{m_1+m_2+i}_F : (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to \tau_i\ \mathit{set}$ for $i \in [n]$;

F's action on relations $\mathsf{rel}_F : (\beta_1 \to \beta_1) \to \cdots \to (\beta_{m_1} \to \beta_{m_1}) \to (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_{m_2} \to \alpha_{m_2}) \to (\tau_1 \to \tau'_1 \to \mathit{bool}) \to \cdots \to (\tau_n \to \tau'_n \to \mathit{bool}) \to (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to (\overline{\beta}, \overline{\alpha}, \overline{\tau})\, F \to \mathit{bool}$ is defined by

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

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

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

$$\begin{aligned}
&\mathsf{map}_F\ [\mathsf{id}]^{m_1}\ [\mathsf{id}]^{m_2}\ [\mathsf{id}]^{n} = \mathsf{id}\\
&(\forall i \in [m_1].\ |\mathsf{supp}\ u_i| < |\beta_i| \wedge |\mathsf{supp}\ u'_i| < |\beta_i|) \wedge\\
&(\forall i \in [m_2].\ |\mathsf{supp}\ v_i| < |\alpha_i| \wedge \mathsf{bij}\ v_i \wedge |\mathsf{supp}\ v'_i| < |\alpha_i| \wedge \mathsf{bij}\ v'_i) \longrightarrow\\
&\quad \mathsf{map}_F\ (u_1 \circ u'_1) \cdots (u_{m_1} \circ u'_{m_1})\ (v_1 \circ v'_1) \cdots (v_{m_2} \circ v'_{m_2})\ (g_1 \circ f_1) \cdots (g_n \circ f_n) =\\
&\quad \mathsf{map}_F\ \overline{u}\ \overline{v}\ \overline{g} \circ \mathsf{map}_F\ \overline{u'}\ \overline{v'}\ \overline{f};
\end{aligned}$$

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

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

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

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

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

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

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

$$\begin{aligned}
&\mathsf{rel}_F\ [\mathsf{id}]^{m_1}\ [\mathsf{id}]^{m_2}\ [(=)]^{n} = (=)\\
&(\forall i \in [m_1].\ |\mathsf{supp}\ u_i| < |\beta_i| \wedge |\mathsf{supp}\ u'_i| < |\beta_i|) \wedge\\
&(\forall i \in [m_2].\ |\mathsf{supp}\ v_i| < |\alpha_i| \wedge \mathsf{bij}\ v_i \wedge |\mathsf{supp}\ v'_i| < |\alpha_i| \wedge \mathsf{bij}\ v'_i) \longrightarrow\\
&\quad \mathsf{rel}_F\ (u_1 \circ u'_1) \cdots (u_{m_1} \circ u'_{m_1})\ (v_1 \circ v'_1) \cdots (v_{m_2} \circ v'_{m_2})\ (R_1 \circledcirc S_1) \cdots (R_n \circledcirc S_n) =\\
&\quad \mathsf{rel}_F\ \overline{u}\ \overline{v}\ \overline{R} \circledcirc \mathsf{rel}_F\ \overline{u'}\ \overline{v'}\ \overline{S}.
\end{aligned}$$

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

6.4 Binding-Aware (Co)induction Proof Principles

This section is dedicated to reasoning about terms, in their well-founded and non-well-founded incarnations, taking their binding structure into consideration (see Chapter 3, Section 3.4). In what follows, we will implicitly assume that α_i is such that $|\alpha_i|$ is regular and $\mathsf{bd}_F \leq |\alpha_i|$.

6.4.1 Induction

Next, we take $\overline{\alpha}\,\boldsymbol{T}$ to be type of (well-founded) terms (as in Section 6.2.3). Let us fix a polymorphic type $\overline{\alpha}\,P$, of entities we will call "parameters." For proving a property such as $\forall t : \overline{\alpha}\,T.\, \forall p : \overline{\alpha}\,P.\, \varphi\ t\ p$, we have at our disposal the standard structural induction principle inherited by the quotient $\overline{\alpha}\,\boldsymbol{T}$ from the (free) datatype $\overline{\alpha}\,T$ of raw terms: It suffices to prove that, for each term $\mathsf{ctor}\ x$, the predicate $\lambda t.\, \forall p : \overline{\alpha}\,P.\, \varphi\ t\ p$ holds for it provided that holds for all its recursive components. However, we can be more ambitious. The following *fresh structural induction* (*FSI*) is a binding-aware improvement inspired by the nominal logic principle of [76]:

Theorem 67 (FSI)**.** Let $\mathsf{PFVars}_i : \overline{\alpha}\,P \to \alpha_i\ \mathit{set}$ with $\forall p : \overline{\alpha}\,P.\, |\mathsf{PFVars}_i\ p| < |\alpha_i|$.
Given a predicate $\varphi : \overline{\alpha}\,\boldsymbol{T} \to \overline{\alpha}\,P \to \mathit{bool}$, if the condition

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

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

Above, we highlighted the two differences from standard structural induction: We assume that parameters p come with sets of variables $\mathsf{PFVars}_i\ p$ which are smaller than α_i. This weakens what we must prove in the induction step for a given term $\mathsf{ctor}\ x$, by allowing us to further assume that x is no-clashing and that its top-binding variables are fresh for the parameter's variables.

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

Proposition 68. We have $\mathsf{sub}\ (g_1 \circ f_1) \cdots (g_m \circ f_m) = \mathsf{sub}\ \overline{g} \circ \mathsf{sub}\ \overline{f}$ for all $f_i, g_i : \alpha_i \to \alpha_i$ such that $|\mathsf{supp}\ f_i| < |\alpha|$ and $|\mathsf{supp}\ g_i| < |\alpha|$ for all $i \in [m]$.

This property would be difficult to prove by standard induction, since the support of the functions $\overline{f}$ and $\overline{g}$ may capture bound variables. With (FSI) the problem is avoided: Taking the parameters to be tuples $(\overline{f}, \overline{g})$ of endofunctions of small support, $\mathsf{PFVars}_i\,(\overline{f}, \overline{g})$ to be $\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i$, and $\varphi\ t\ (\overline{f}, \overline{g})$ to be $\mathsf{sub}\ (g \circ f)\, t = (\mathsf{sub}\ \overline{g} \circ \mathsf{sub}\ \overline{f})\, t$. In the inductive case, we must prove

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

assuming that the fact holds for all recursive components of x and that the binding variables of x do not appear free in $\mathsf{ctor}\ x$ and also are not in $\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i$ for each $i \in [m]$. Thanks

to this second assumption, we are able push the substitution inside the components of ctor x according to the recursive law $(*)$ for sub, thus reducing what we need to prove to

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

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

6.4.2 Coinduction

Next, we take $\overline{\alpha}\,\boldsymbol{T}$ to be the type of non-well-founded terms (as in Section 6.2.8). Concerning binding-aware proof principles for $\overline{\alpha}\,\boldsymbol{T}$, we encounter a discrepancy from the inductive case. The standard structural coinduction principle imported from raw terms would allow us to prove that a binary relation on $\overline{\alpha}\,\boldsymbol{T}$ is included in the equality if it is an F-bismulation (i.e., if it is preserved by F's relator). Using the ideas discussed in Section 6.4.1, we can prove a parameter-based fresh variation of this principle, where we again emphasize the binding-specific enhancements:

Theorem 69 (FSC)**.** Let $\overline{\alpha}\,P$ and PFVars_i be like in Theorem 67. Given a binary relation $\varphi : \overline{\alpha}\,\boldsymbol{T} \to \overline{\alpha}\,\boldsymbol{T} \to \overline{\alpha}\,P \to \mathit{bool}$, if the condition

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

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

However, this proof principle turns out to be not as useful as its inductive counterpart. Consider the task of proving Proposition 68 for non-well-founded terms. Let us attempt to prove it using (FSC). We again take the parameters to be tuples $(\overline{f}, \overline{g})$ of endofunctions of small support and $\mathsf{PFVars}_i\ (\overline{f}, \overline{g})$ to be $\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i$. Define $\varphi\ t_1\ t_2\ (\overline{f}, \overline{g})$ as $\exists t.\ t_1 = \mathsf{sub}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m)\ t \wedge t_2 = (\mathsf{sub}\ \overline{g} \circ \mathsf{sub}\ \overline{f})\ t$. Then, it suffices to verify (FSC)'s hypothesis. So we assume that, for all endofunctions of small support f_i and g_i, (1) $\mathsf{ctor}\ x_1 = \mathsf{sub}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m)\ t$ and $\mathsf{ctor}\ x_2 = (\mathsf{sub}\ \overline{g} \circ \mathsf{sub}\ \overline{f})\ t$, and (2) $(\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i) \cap (\mathsf{topBind}_i\ x_1 \cup \mathsf{topBind}_i\ x_2) = \varnothing$ for all $i \in [m]$. We must prove (3) $\mathsf{rel}_F\ [(=)]^m\ [(=)]^m\ [\lambda t_1\ t_2.\ \forall (\overline{f}, \overline{g}).\ \varphi\ t_1\ t_2\ (\overline{f}, \overline{g})]^m\ x_1\ x_2$. To this end, assume t has the form $\mathsf{ctor}\ x$, where thanks to the availability of enough fresh variables we can assume $(\mathsf{supp}\ f_i \cup \mathsf{supp}\ g_i) \cap \mathsf{topBind}_i\ x = \varnothing$ for all $i \in [m]$. This allows us to push sub under the constructor in the equalities (1), obtaining (4) $\mathsf{ctor}\ x_1 = \mathsf{ctor}\ x_1'$ and $\mathsf{ctor}\ x_2 = \mathsf{ctor}\ x_2'$ where

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

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

but not both.

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

Theorem 70 (ESC)**.** Given a binary relation $\varphi : \overline{\alpha}\,\boldsymbol{T} \to \overline{\alpha}\,\boldsymbol{T} \to \mathit{bool}$, if the condition

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

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

(ESC) is more general than, and in fact can easily prove, (FSC). It resolves the problem in our concrete example with substitution (because it allows us to dynamically shift from x_1 and x_2 to x_1' and x_2') and similar problems when proving equational theorems on non-well-founded terms.

6.5 Binding-Aware (Co)recursive Definition Principles

Two aspects have not been formally addressed so far. The first concerns Theorems 62 and 63, stating the existence of substitution operators. While we have shown how the need for sufficiently many fresh variables can be fulfilled, we have not accounted for the possibility to define the substitutions as well-behaved functions on (quotiented) terms. The second aspect concerns a standard litmus test for abstract data types: the unique characterization of a construction up to isomorphism. In contrast to raw terms, which are known to form initial objects in the category of BNF algebras [70], the status of terms is currently less abstract, since they rely on alpha-equivalence. Can we also characterize the term algebras as initial objects?

The resolution of both aspects rests on the availability of suitable (co)recursion definitional principles, *(co)recursors* for short, for the types of terms— the older brothers of these principles have been presented for the previous framework in Chapter 3, Section 3.3. The main technical difficulty in developing such a recursor is that terms do not form a free datatype, which means that defining functions on terms is no longer possible by simply listing some constructor-based recursive clauses. Instead, the recursor must be aware of the nonfreeness introduced by bindings. And a similar (dual) problem holds for the corecursor. The key to address these problem is to identify suitable abstract algebraic structures that satisfy term-like properties, to be used as (co)domains for (co)recursive definitions.

We will present a simple version of the (co)recursors, which are more commonly called (co)iterators. The supplementary material covers the straightforward extension to full-fledged (co)recursors.

Below, we implicitly assume that $|\alpha_i|$ is regular and $\mathsf{bd}_F \leq |\alpha_i|$. In addition, unless otherwise stated, f_i and g_i range over (endo)bijections of type $\alpha_i \to \alpha_i$ that have small support: $|\mathsf{supp}\ f_i| < |\alpha_i|$ and $|\mathsf{supp}\ g_i| < |\alpha_i|$.

Definition 71. A *term-like structure* is a triple $\mathcal{D} = (\overline{\alpha}\,D, \overline{\mathsf{DFVars}}, \mathsf{Dmap})$, where

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

are such that the following properties hold:

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

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

6.5.1 Binding-Aware Recursor

Next, we take $\overline{\alpha}\,\boldsymbol{T}$ to be type of (well-founded) terms. Recall from Section 6.4.1 that fresh induction relies on parameters, assumed to be equipped with small-cardinality free-variable-like operators. To discuss recursion, we need parameters to have map functions as well:

Definition 72. A *parameter structure* is a term-like structure $\mathcal{P} = (\overline{\alpha}\,P, \overline{\mathsf{FVars}}, \mathsf{Pmap})$ such that $\forall p : \overline{\alpha}\,P.\ |\mathsf{PFVars}_i\ p| < |\alpha_i|$.

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

Definition 73. Given a parameter structure $\mathcal{P}$, a $\mathcal{P}$*-model* is a quadruple $\mathcal{U} = (\overline{\alpha}\,U, \overline{\mathsf{UFVars}}, \mathsf{Umap}, \mathsf{Uctor})$, where

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

such that the following properties hold:

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

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

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

Theorem 74. Given a parameter structure $\mathcal{P}$ and a $\mathcal{P}$-model $\mathcal{U}$, there exists a unique function $H : \overline{\alpha}\,\boldsymbol{T} \to \overline{\alpha}\,P \to \overline{\alpha}\,U$ that preserves the constructor, mapping, and free-variable operators:

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

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

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

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

This function depends on both terms and parameters. Intuitively, H recurses over terms while the binding variables are assumed to avoid the parameters' variables. Indeed, the theorem's clause (C) specifies the behavior of H on terms of the form ctor x not for an arbitrary x, but for a (no-clashing) x whose top-binding variables do not overlap with those of a given parameter p. This is the recursive-definition incarnation of Barendregt's convention, just as the parameter twist of fresh induction (Theorem 67) is its inductive-proof incarnation.

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

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

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

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

(VC) is the model version of this last property, mutatis mutandis, e.g., replacing ctor and FVars_i with Uctor and UFVars_i, together with the additional weakening brought by its first premise: the top-binding variables in Uctor y p are fresh for the parameters. Given that weaker model axiomatizations lead to more expressive recursors, our recursor improves on the state of the art (see next Chapter 7).

To define the variable-for-variable substitution sub, we define $\mathcal{P}$ by taking $\overline{\alpha}\,P$ to consist of all tuples of small-support endofunctions $\overline{f}$, $\mathsf{PFVars}_i\ \overline{f} = \mathsf{supp}\ f_i$ and $\mathsf{Pmap}\ \overline{g}\ \overline{f} = \overline{g} \circ \overline{f} \circ \overline{g}^{-1}$. We define the $\mathcal{P}$-model $\mathcal{U}$ by taking $\overline{\alpha}\,U = \overline{\alpha}\,\boldsymbol{T}$, $\mathsf{UFVars}_i = \mathsf{FVars}_i$, $\mathsf{Umap} = \mathsf{map}_{\boldsymbol{T}}$ and $\mathsf{Uctor}\ y\ \overline{f} = \mathsf{ctor}\ (\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\lambda pu.\ pu\,\overline{f}]^n\ y)$. To apply Theorem 74, we must check its hypotheses, which amount to standard identities on terms. We obtain a function $\mathsf{sub} : \overline{\alpha}\,\boldsymbol{T} \to \overline{\alpha}\,P \to \overline{\alpha}\,\boldsymbol{T}$ satisfying three clauses, among which

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

By (restricted) functoriality, $\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\lambda pu.\ pu\,\overline{f}]^n\ (\mathsf{map}_F\ [\mathsf{id}]^{2*m}\ [\mathsf{sub}]^n\ x) = \mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [\lambda t.\ \mathsf{sub}\ t\,\overline{f}]^n\ x$, making (C) equivalent to Section 6.2.5's clause $(*)$, hence proving the desired behavior for substitution (Theorem 62)—that is, after flipping the arguments of sub, to turn it into a function of type $\overline{\alpha}\,P \to \overline{\alpha}\,\boldsymbol{T} \to \overline{\alpha}\,\boldsymbol{T}$.

Term-for-variable substitution (Section 6.2.7) can be defined similarly.

To characterize terms as an abstract data type, let $\bot$ be the parameter structure where the carrier type is a singleton, the map function is trivial, and PFVars_i returns $\varnothing$ for all $i \in [m]$. We obtain the following, as an immediate consequence of Theorem 74:

Corollary 75. $(\overline{\alpha}\,\boldsymbol{T}, \overline{\mathsf{FVars}}, \mathsf{map}_{\boldsymbol{T}}, \mathsf{ctor})$ is the initial $\bot$-model (where a model morphism is a function that preserves all the operators).

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

6.5.2 Binding-Aware Corecursor

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

Yet something akin to a coalgebaic structure can still be obtained if we leave some room for nondeterminism. Namely, we define a nondeterministic destructor $\mathsf{dtor} : \overline{\alpha}\,\boldsymbol{T} \to ((\overline{\alpha}, \overline{\alpha}, [\overline{\alpha}\,\boldsymbol{T}]^n)\,F)\,set$ as $\mathsf{dtor}\ t = \{x \mid t = \mathsf{ctor}\ x\}$. Crucially, this destructor is still *deterministic up to a renaming* of the top-binding variables. Indeed, Prop. 60 ensures that, for any $x, x' \in \mathsf{dtor}\ t$, we have $x' = \mathsf{map}_F\ [\mathsf{id}]^m\ \overline{f}\ [\mathsf{map}_{\boldsymbol{T}}\,\overline{f}]^n\ x$ for some small-support endobijections $\overline{f}$ subject to some suitable conditions. This suggests the following axiomatization of corecursive models:

Definition 76. A *comodel* is a quadruple $\mathcal{U} = (\overline{\alpha}\,U, \overline{\mathsf{UFVars}}, \mathsf{Umap}, \mathsf{Udtor})$, where

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

- $\mathsf{Udtor} : \overline{\alpha}\,U \to ((\overline{\alpha},\overline{\alpha},[\overline{\alpha}\,U]^n)\,F)\,set$

such that the following properties hold:

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

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

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

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

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

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

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

Theorem 77. Given a comodel $\mathcal{U}$, there exists a unique function $H : \overline{\alpha}\,U \to \overline{\alpha}\,\boldsymbol{T}$ that preserves the destructor, mapping and free-variable operators, in the following sense:

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

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

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

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

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

which shows the corecursive behavior of H in a more operational fashion: To build a (possibly infinite) term starting with the input d, H can choose any $y \in \mathsf{Udtor}\ d$ and then delve into y after "producing" a ctor. Thanks to the comodel axioms (notably, (DRen)), the choice of y does not matter.

Corollary 78. $(\overline{\alpha}\,\boldsymbol{T}, \overline{\mathsf{FVars}}, \mathsf{map}_{\boldsymbol{T}}, \mathsf{dtor})$ is the final comodel.

Unlike models, our comodels do not have parameters. This is because, in the corecursive case, any freshness assumptions can be easily incorporated in the choice of the destructor-like operator. (This mirrors the situation of binding-aware coinduction, which also departs from binding-aware induction on the very topic of explicit parameters.) The corecursive definition of substitution is a good illustration of this phenomenon. We define the comodel $\mathcal{U}$ by taking $\overline{\alpha}\,U$ to consists of all pairs $(t, \overline{f})$ with t term and $\overline{f}$ tuple of small-support endofunctions, $\mathsf{UVar}_i\,(t,\overline{f}) = \mathsf{FVars}_i\,t \cup \mathsf{supp}\ f_i$, $\mathsf{Umap}\,\overline{g}\,(t,\overline{f}) = (\mathsf{map}_{\boldsymbol{T}}\,\overline{g}\,t, \overline{g} \circ \overline{f} \circ \overline{g}^{-1})$, and $\mathsf{Udtor}\ (t,\overline{f}) = \{\mathsf{map}_F\ \overline{f}\ [\mathsf{id}]^m\ [(\lambda t'.\,(t',\overline{f}))]^n\ x \mid x \in \mathsf{dtor}\ t \wedge \mathsf{noClash}_i\,x \wedge \mathsf{topBind}_i\,x \cap \mathsf{supp}\ f_i = \varnothing\}$. We have highlighted how we insulate, among all possible ways to choose $x \in \mathsf{dtor}\ t$, those x's that avoid capture, as required by the desired clause $(*)$ for substitution—which is the same as for well-founded terms. This is a general trick for replacing the explicit use of parameters. After checking that this is indeed a comodel, Theorem 77 offers the function $\mathsf{sub} : \overline{\alpha}\,U \to \overline{\alpha}\,\boldsymbol{T}$ that satisfies three clauses, among which

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

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

6.6 Useful Variations of the (Co)recursion Principles

6.6.1 A Fixed-Parameter Restriction

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

Definition 79. Given a tuple $\mathcal{A}$ of small sets, an $\mathcal{A}$-model is a quadruple $\mathcal{U} = (\overline{\alpha}\,U, \overline{\mathsf{UFVars}}, \mathsf{Umap}, \mathsf{Uctor})$, where:

- $(\overline{\alpha}\,U, \overline{\mathsf{UFVars}}, \mathsf{Umap})$ is a term-like structure
- $\mathsf{Umap} : (\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_m \to \alpha_m) \to \overline{\alpha}\,U \to \overline{\alpha}\,U$

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

such that the following hold:

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

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

Then Theorem 74 instantiates to:

Theorem 80. Given a tuple of small sets $\mathcal{A}$ and an $\mathcal{A}$-model $\mathcal{U}$, there exists a unique function $H : \overline{\alpha}\,\boldsymbol{T} \to \overline{\alpha}\,U$ such that:

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

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

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

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

6.6.2 The Full-Fledged Primitive (Co)recursor

In Section 6.5 we have presented a restricted form of (co)recursors that are usually known as (co)iterators. Here we formulate the full-fledged (co)recursors, which constitute a theoretically straightforward but practically useful extension of the (co)iterators.

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

Definition 81. An *extended term-like structure* is a triple $\mathcal{D} = (\overline{\alpha}\,D, \overline{\mathsf{DFVars}}, \mathsf{Dmap})$, where

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

are such that the following hold:

- $\mathsf{Dmap}\ [\mathsf{id}]^m\ t\ = \mathsf{id}$
- $\mathsf{Dmap}\ (g_1 \circ f_1)\ \cdots\ (g_m \circ f_m)\ t\ = \mathsf{Dmap}\ \overline{g}\ t \circ \mathsf{Dmap}\ \overline{f}\ t$

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

Definition 82. Given a parameter structure $\mathcal{P}$, an extended $\mathcal{P}$-model is a quadruple $\mathcal{U} = (\overline{\alpha}\, U, \overline{\mathsf{UFVars}}, \mathsf{Umap}, \mathsf{Uctor})$, where:

- $(\overline{\alpha}\, U, \overline{\mathsf{UFVars}}, \mathsf{Umap})$ is an extended term-like structure
- $\mathsf{Uctor} : (\overline{\alpha}, \overline{\alpha}, [\ \overline{\alpha}\, \boldsymbol{T} \times (\overline{\alpha}\, P \to \overline{\alpha}\, U)]^n)\, F \to \overline{\alpha}\, P \to \overline{\alpha}\, U$

such that the following hold:

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

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

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

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

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

Full-fledged recursion extension of Theorem 74: Given a parameter structure $\mathcal{P}$ and a $\mathcal{P}$-model $\mathcal{U}$, there exists a unique function $H : \overline{\alpha}\, \boldsymbol{T} \to \overline{\alpha}\, P \to \overline{\alpha}\, U$ such that:

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

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

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

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

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

6.6.3 A Constructor-Based Variation

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

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

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

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

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

Theorem 83. Any weak extended model is also an extended model, hence (the extended version of) Theorem 74 also holds for weak extended $\mathcal{P}$-models.

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

6.7 Formal Comparison with Recursors from the Literature

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

Now, $\alpha\ \boldsymbol{T}$ denotes the type of λ-terms with variables in α. To avoid confusion with the meta-level, we will write $\mathsf{App} : \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$ and $\mathsf{Lam} : \alpha \to \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$ for the application and λ-abstraction constructors on terms. Note that our abstract constructor $\mathsf{ctor} : (\alpha, \alpha, \alpha\ \boldsymbol{T}, \alpha\ \boldsymbol{T})\ F = \alpha + \alpha\ \boldsymbol{T} \times \alpha\ \boldsymbol{T} + \alpha \times \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$ is the joining of the "concrete" Var, App and Lam constructors. We will prefer to present the λ-calculus instance of our theorem

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

in terms of the concrete constructors. This will also apply to models, where the abstract constructor-like operator Uctor will be correspondingly split into three operators UVar, UApp and ULam. The abstract (single-constructor) and the concrete (multi-constructor) views are the same, modulo the trivial transformations of sum splitting/joining and (un)currying of functions.

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

- $\alpha\ D$ is a polymorphic type
- $\mathsf{DFVars} : \alpha\ D \to \alpha\ \boldsymbol{T} \to \alpha\ set$
- $\mathsf{Dmap} : (\alpha \to \alpha) \to \alpha\ \boldsymbol{T} \to \alpha\ D \to \alpha\ D$

are such that the following hold:

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

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

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

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

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

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

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

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

(CC)
- $\mathsf{UFVars}\ (pu_1\ p) \subseteq \mathsf{FVars}\, t_1 \cup \mathsf{PFVars}\ p \wedge \mathsf{UFVars}\ (pu_2\ p) \subseteq \mathsf{FVars}\, t_2 \cup \mathsf{PFVars}\ p \wedge$
  $\mathsf{UFVars}\ (pu'_1\ p) \subseteq \mathsf{FVars}\, t'_1 \cup \mathsf{PFVars}\ p \wedge \mathsf{UFVars}\ (pu'_2\ p) \subseteq \mathsf{FVars}\, t'_2 \cup \mathsf{PFVars}\ p$
  $\mathsf{supp}\ f \cap (\mathsf{FVars}\,(\mathsf{App}\ t_1\ t_2) \cup \mathsf{PFVars}\ p) = \varnothing \wedge$
  $\mathsf{supp}\ f' \cap (\mathsf{FVars}\,(\mathsf{App}\ t'_1\ t'_2) \cup \mathsf{PFVars}\ p) = \varnothing \wedge$
  $\mathsf{map}_{\boldsymbol{T}}\ f\ t_1 = \mathsf{map}_{\boldsymbol{T}}\ f'\ t'_1 \wedge \mathsf{Umap}\ f\ t_1\ pu_1 = \mathsf{Umap}\ f'\ t'_1\ pu'_1 \wedge$
  $\mathsf{map}_{\boldsymbol{T}}\ f\ t_2 = \mathsf{map}_{\boldsymbol{T}}\ f'\ t'_2 \wedge \mathsf{Umap}\ f\ t_2\ pu_2 = \mathsf{Umap}\ f'\ t'_2\ pu'_2$
  $\longrightarrow \mathsf{UApp}\ t_1\ pu_1\ t_2\ pu_2 = \mathsf{UApp}\ t'_1\ pu'_1\ t'_2\ pu'_2$
- $\{a,a'\} \cap \mathsf{PFVars}\ p = \varnothing \wedge$
  $\mathsf{UFVars}\ (pu\ p) \subseteq \mathsf{FVars}\, t \cup \mathsf{PFVars}\ p \wedge \mathsf{UFVars}\ (pu'\ p) \subseteq \mathsf{FVars}\, t' \cup \mathsf{PFVars}\ p \wedge$
  $\mathsf{supp}\ f \cap (\mathsf{FVars}\,(\mathsf{Lam}\ a\ t) \cup \mathsf{PFVars}\ p) = \varnothing \wedge f\ a \neq a \wedge$
  $\mathsf{supp}\ f' \cap (\mathsf{FVars}\,(\mathsf{Lam}\ a'\ t') \cup \mathsf{PFVars}\ p) = \varnothing \wedge f'\ a' \neq a' \wedge$
  $f\ a = f'\ a' \wedge \mathsf{map}_{\boldsymbol{T}}\ f\ t = \mathsf{map}_{\boldsymbol{T}}\ f'\ t' \wedge \mathsf{Umap}\ f\ t\ pu = \mathsf{Umap}\ f'\ t'\ pu'$
  $\longrightarrow \mathsf{ULam}\ a\ t\ pu = \mathsf{ULam}\ a'\ t'\ pu'$

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

Instantiating (the extended version of) Theorem 74 and Theorem 83 for this syntax gives us the following (again, after performing the splitting according to concrete constructors):

Corollary 84. Given a parameter structure $\mathcal{P}$ and a (weak) extended $\mathcal{P}$-model $\mathcal{U}$, there exists a unique function $H : \alpha\ \boldsymbol{T} \to \alpha\ P \to \alpha\ U$ such that

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

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

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

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

If above we remove the highlighted text, we obtain the more primitive, iterative form of the recursor. It is also worth spelling out the fixed-parameter instance of the iterator (described in general in Section 6.6.1). Fixing $\mathcal{A}$ to consist of a single finite[5] set A, the notion of $\mathcal{A}$-model (for the λ-calculus syntax) becomes a quadruple $\mathcal{U} = (\alpha\ U, \mathsf{UFVars}, \mathsf{Umap}, \mathsf{Uctor})$, where:

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

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

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

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

Then Theorem 80 instantiates to:

Corollary 85. Given an $\mathcal{A}$-model $\mathcal{U}$, there exists a unique function $H : \alpha\ \boldsymbol{T} \to \alpha\ U$ such that:

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

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

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

Comparison with the nominal recursor

Recall that, for this particular finitary syntax:

- Our type variable α, assumed countable, corresponds to a Nominal set of atoms
- Our functions $f : \alpha \to \alpha$ of small support correspond to permutations of finite support
- Our map function $\mathsf{map}_{\boldsymbol{T}}$ corresponds to the swapping action on terms (see also the discussion in Section 7.1 from Chapter 7)

As we have already pointed out (Chapter 2, Section 2.4), a classic result of group theory is that $Sym_f(\alpha)$ (namely the group of all finitely supported endo-bijections on α) is a group generated by transpositions, i.e., every finitely supported bijection is obtainable from the composition of a finite number of transposition. We can make one more claim and state the next result.

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

Lemma 7. *The usual swapping operation, with its properties, induces the nominal action.*

Formally, we can obtain this with the following steps (where the notation $\_ \circ \_$ indicates the usual function composition):

1 We define the nominal action not for all permutations with finite support, but only for swapping (transpositions).

2 We extend it to any generic finitely supported permutation f, picking $f.t$ to be $(x_n \leftrightarrow x_{n-1})$ $.(\ldots.((x_1 \leftrightarrow x_0).t)\ldots)$, where $f = (x_n \leftrightarrow x_{n-1}) \circ \cdots \circ (x_1 \leftrightarrow x_0)$.

3 We observe that for every two representation in terms of transpositions of y, $f = (x_n \leftrightarrow x_{n-1}) \circ \cdots \circ (x_1 \leftrightarrow x_0) = (y_m \leftrightarrow y_{m-1}) \circ \cdots \circ (y_1 \leftrightarrow y_0)$, we obtain that $(x_n \leftrightarrow x_{n-1}).(\ldots .((x_1 \leftrightarrow x_0).t)\ldots) = (y_m \leftrightarrow y_{m-1}).(\ldots.((y_1 \leftrightarrow y_0).t)\ldots)$ (namely, that what we are defining is indeed a function)[6].

4 We do routine checks for the two properties defining an action.

5 Proving that the two actions are equal is again routine.

Thus, we can formulate (the λ-calculus instance of) the nominal recursor [58] just relying on swapping and using the terminology of this chapter:

Definition 86. The notion *nominal $\mathcal{A}$-model* is obtained from that of $\mathcal{A}$-model by removing the last two (out of four) term-like structure axioms and the (VC) axiom and adding instead the following axioms:

(FfromM) $\mathsf{UFVars}\ u = \{a \mid \{a' \mid \mathsf{Umap}\ (a \leftrightarrow a')\ u \neq u\}\ \text{finite}\}$

(FCB) $\exists u.\ \forall a.\ a \notin A \wedge a \notin \mathsf{UFVars}\ u$

In clause (FfromM), $(a \leftrightarrow a')$ denotes the swapping function[7] in $\alpha \to \alpha$, sending a to a', a' to a and everything else to itself. Note that we use a free-variable-like operator $\mathsf{UFVars} : U \to \alpha\ set$, whereas the nominal logic literature considers a freshness operator $\mathsf{Ufresh} : \alpha\ U \to \alpha \to bool$ (and usually writes $a\#u$ instead of $\mathsf{Ufresh}\ u\ a$). These are of course inter-definable via negation, as follows:

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

With this translation, we see that (FfromM) states that freshness is definable from mapping (i.e., from the nominal permutation action)[8] and (FCB) is the so-called *freshness condition for binders*, both familiar from nominal logic:

(FfromM) $\mathsf{Ufresh}\ a\ u \longleftrightarrow \{a' \mid \mathsf{Umap}\ (a \leftrightarrow a')\ u \neq u\}$ infinite

[6] We have proved this in Isabelle [**actions-scripts**], the proof goes by induction on the overall number of transpositions, plus some lemmas on the properties of swapping.

[7] Here for swapping we use a notation close to the one we used for group actions in Chapter 2, Subsection 2.4.

[8] See in [76], Def. 3. [58] gives a slightly more complex definition (as the minimal set that supports and entity)—these two definitions are known to be equivalent in the presence of finite support, which is a pervasive assumption in nominal logic.

(FCB) $\exists u.\, \forall a.\, a \notin A \wedge a \notin \mathsf{Ufresh}\ a\ u$

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

Proposition 87. Any nominal $\mathcal{A}$-model is an $\mathcal{A}$-model.

As a consequence, we obtain the following:

Corollary 88. We have that Corollary 85 stays true if we replace $\mathcal{A}$-models with nominal $\mathcal{A}$-models.

This last corollary is essentially the sort-directed alpha-structural recursion theorem (Theorem 5.1) of [58]—henceforth abbreviated ASRT—instantiated to the λ-calculus syntax (thus, taking the signature Σ to consist of the λ-calculus constructors $\mathsf{Var}, \mathsf{Lam}, \mathsf{App}$).

Indeed, concerning the input to the two recursion theorems: Our model carrier $\alpha\ U$ corresponds to the the ASRT target domain X, and our constructor-like operators $\mathsf{UVar}, \mathsf{ULam}, \mathsf{UApp}$ correspond to the ASRT functions f_k considered on the target domain. Our set A corresponds to the ASRT set A. Our first two term-like structure axioms (the only ones kept in the notion of $\mathcal{A}$-nominal model) correspond the ASRT requirement that the permutation action satisfies the first nominal-set axioms concerning the identity and composition of permutation actions. Our axioms (MC) correspond to the fact that the functions f_k are supported by A.[9] Our condition (FCB) corresponds to the identically named ASRT condition—noting that for the λ-calculus syntax this condition is only meaningful for Lam, the only constructor that actually binds variables. Finally, our nominal $\mathcal{A}$-models have Ufresh as part of their structure, whereas the ASRT target domain does not have any freshness operator as a primitive. However, the ASRT target domain is required to be a nominal set, hence it has a notion of freshness definable from the permutation action—which is exactly what our (FfromM) axiom imposes. In short, there is a precise (bijective) correspondence between our nominal $\mathcal{A}$-models and the structure considered on the ASRT target domains.

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

Comparison with the Norrish and the Gheri-Popescu recursors These recursors operate with swapping actions $(a \leftrightarrow b) : \alpha \to \alpha$, also known as transpositions, rather than arbitrary permutations. The possibility to restrict the focus in this way is based on the fact that all

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

finite-support permutations are generated from transpositions via composition . (This does not scale, however, to the infinitary case, as already pointed out in Section 2.4, 2.)

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

Definition 89. A swapping-based term-like structure is a triple $\mathcal{D} = (\alpha\ D, \mathsf{DFVars}, \mathsf{Dswap})$, where

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

are such that the following hold:[10]

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

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

$$\mathsf{Dmap}\ a_1\ a'_1\ \circ\ \ldots\ \circ\ \mathsf{Dmap}\ a_n\ a_n \qquad (\$)$$

where

$$f = (a_1, a'_1) \circ \ldots \circ (a_n, a'_n) \qquad (\$\$)$$

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

Proposition 90. The above correspondence (extended to morphisms in the expected way) is an equivalence between the categories of swapping-based term-like structures and that of term-like structures.

Based on this correspondence, it is straightforward to adapt our permutation based recursor to a swapping-based recursor. It employs swapping-based models, which feature an operator $\mathsf{Uswap} : \alpha \times \alpha \to \alpha\ U \to \alpha\ U$ instead of $\mathsf{Umap} : (\alpha \to \alpha) \to \alpha\ U \to \alpha\ U$, etc. Its final theorem is a swapping-based variant of Corollary 85 that replaces clause (M) with the following:

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

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

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

- our recursion theorem is a slightly stronger form of the recursor in [53]
- our constructor-based variation is a parameter-based improvement of the recursor described in Chapter 3

Our Isabelle formalization [14] contains formal proofs of this facts (for the syntax of λ-calculus).

6.8 Isabelle Formalization and Implementation

All our results have been formalized in Isabelle/HOL [14], in a slightly less general case than presented in this Chapter. Namely, while the formalization is abstract in that it works with arbitrary type constructors and constants (such as F and map_F), it is concrete in that it fixes certain arities for the type constructors. Thus, for the fixpoint constructions, instead of $\theta \subseteq [m] \times [n]$ and $(\overline{\beta},\overline{\alpha},\overline{\tau})\,F$ where $\mathsf{len}\,(\overline{\beta}) = \mathsf{len}\,(\overline{\alpha}) = m$ and $\mathsf{len}\,(\overline{\tau}) = n$, we work with an F of a fixed arity, $(\beta_1,\alpha_1,\tau_1,\tau_2)\,F$ taking $m = n = 1$ and $\theta = \{(1,2)\}$.

This is the best we can do while working in the Isabelle/HOL user space, given that in the HOL logic we cannot consider type constructors depending on varying numbers of type variables. On the positive side, having the fixed-arity case fully worked out gives us strong confidence that our results are correct. Moreover, by doing a lot of copy-pasting we can easily adapt our formalization to any given m,n and θ. On the negative side, our framework is not yet usable in the way a definitional package such as Isabelle Nominal [76] and the BNF-based (co)datatype package [18] is. Besides the above issue with arities, another missing component is a mechanism for hiding away the category theory under some user-friendly notation, e.g., splitting, for sum types, the single abstract constructor into multiple user-named constructors.

As a first step in the usability direction, our coauthors—for the submitted paper [15] currently under review— have implemented in Standard ML a tool that automates the process of instantiating an Isabelle formalization parameterized by some type constructors and polymorphic constants with any desired user specified instances—e.g., to switch from an arbitrary functor F to the particular one required by the syntax of the λ-calculus. This tool has already been very helpful with instantiating the arbitrary (co)models used by our (co)recursion principles into the specific ones needed to define substitution.

The distance between our current work and a fully usable definitional package is a good opportunity to reflect on the benefits and limitations of our bindings as functors approach. The main benefit is the semantic treatment of binders, which allows us to "plug and play" arbitrarily complex binders into (co)datatypes—an improvement over the syntactic approaches which essentially inline the whole complexity of binders into the (co)datatypes. On the other hand, we should stress that the functorial approach is no substitute for the usual combinatorial

complexity stemming from many-sortedness: multiple types of variables bound in multiple types of terms. To cope with these, we are forced to consider multiple arguments for our functors (and also mutually recursive (co)datatypes). In conclusion, unlike the syntactic approaches we can hide the binding complexity, but like the syntactic approaches we must face the many-sortedness complexity.

Chapter 7

Conclusion and Related Work

7.1 Literature Review

Following our desiderata, we have developed a theory of syntax with bindings, exploiting along the way different concepts and testing it against different theoretical and formalization challenges. In its final shape, it relies on map-restricted bounded natural functors, MRBNFs in short, an appropriate class of functors we have identified. These are an extension of BNFs (Sections 2.2 and 2.3, Chapter 2) that can accommodate arbitrarily complex statically scoped bindings without committing to any syntactic format. The universe of MRBNFs is closed under the operations of turning an input nonrepetitive (useful for constructing binders) and of taking binding-aware least and greatest fixpoints ((co)datatypes). These constructions come equipped with powerful reasoning and definitional principles.

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

7.1.1 Major Frameworks for Bindings

Nominal Logic There is considerable overlap between our framework and nominal logic [59, 58], which is itself a syntax-free axiomatization of term-like entities that can contain variables, called atoms. Our theory takes the nominal approach, in which it uses explicit names for bound variables, has a similar treatment of alpha-equivalence and applies nominal techniques for recovering Barendregt's variable convention (e.g. fresh induction [76]). Furthermore, we can draw a precise correspondence between a specific fragment of our framework and nominal logic: our MRBNF restriction to small-support functions, as well as our overall approach, are inspired by nominal logic [58]. Let $\overline{\alpha}F$ be an $\overline{\alpha}$-binding MRBNF (whose inputs are all binding inputs) that is finitary (i.e., $\mathsf{bd}_F = \aleph_0$) and fix $\overline{\alpha}$ to some countable types. Then $\overline{\alpha}F$ is a (multi-atom) nominal set having $\overline{\alpha}$ as sets of atoms. Moreover, our endobijections $f_i : \alpha_i \to \alpha_i$ of small support coincide with what nominal logic calls permutations of finite support, and the map function $\mathsf{map}_{\boldsymbol{T}}$ is the same as the nominal permutation action. In what follows we give some more detail for the unary case.

Recalling definition 1 from Subsection 2.4, Chapter 2, an action of finitely-supported permutations of atoms (variables) on a set S is a function:

$$\Phi \ : \ Sym_f(A) \to Sym(S)$$

where A is the set of atoms, satisfying the following:

1. $\Phi\ \mathsf{id} = \mathsf{id}$;
2. $\Phi\ (h \circ g) = (\Phi\ h) \circ (\Phi\ g)$, for all f, g finitely supported permutations.

Also, a MRBNF is first of all a functor (property **(Fun)** from Section 6.3, Chapter 6). Namely, considering the unary case and fixing, we have a (unary) type constructor F with a mapper $\mathsf{map}_F : (\alpha \to \alpha) \to (\alpha\ F \to \alpha\ F)$ satisfying the following two properties:

1. $\mathsf{map}_F\ \mathsf{id} = \mathsf{id}$;
2. $\mathsf{map}_F\ (h \circ g) = (\mathsf{map}_F\ h) \circ (\mathsf{map}_F\ g)$, for all functions f and g.

If we take the set of atoms A to be the set of elements of type α and we restrict the functorial action of map_F to finitely-supported permutations (*endobijections*) of α, the two formulations of these properties express exactly the same concept, the first time in the terminology of group theory, the second in that of category theory.

Nevertheless, there are some important differences between our theory and the nominal development, due to the choice of functors for modelling the presence of variables, instead of atom-enriched sets.

First, we exploit a mechanism that is already present in the logical foundation: the dependence of type constructors on type variables. Moreover, unlike nominal sets, which are assigned fixed collections of atoms, the inputs to our functors are parameters that can be instantiated in various ways. We exploit this flexibility to remove the finite support restriction and to accept terms that are infinitely branching, that have infinite depth, or both. To accommodate such larger entities, all we need to do is instantiate type variables with suitably large types: we have generalized finite support to "small" support (property **(Bound)** from Section 6.3, Chapter 6), in order to capture infinitary syntaxes. Because of this, swapping has lost his central role in our development, replaced by small-support *renaming* endobijections. However this is no surprise: from group theory it is well-known that every finitely-supported permutation is a composition of transpositions (2.4), but at the same time this result cannot be extended to infinitely-supported permutations.

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

Higher-order abstract syntax A well-established alternative to nominal logic is higher-order abstract syntax (HOAS) [34, 56], which reduces the bindings of the object syntax to those of the metalogic. HOAS can often simplify reasoning [27, 57], but its reliance on the metalogic's binder makes it difficult (if possible at all) to encode complex binding patterns. In [34] this fact is openly recognized and so useful techniques that improved on this issue have been proposed—immediately in that paper and in many subsequent works. However,

the vast majority of these improvements consists of a substitution of the metalogic binding mechanism for a more general and powerful one [56], thus showing that this metalogic binder represents indeed a limitation, a "supremum" for what binders can be encoded.

Nameless and Locally Nameless Another approach is to treat (bound) variables as nameless objects. In this context many different frameworks have been developed in different proof assistants, such as presheaf-based abstract syntax [29, 38, 3], binding signatures based on modules over monads [1] (building on [36, 37]; results checked in the UniMath library in Coq), bindings embedded in nested datatypes [11], bindings embedded in dependent types [2] (formalized in Agda), the locally nameless representation [6, 21], Autosubst [66], (the last two formalized in Coq). As it happens with HOAS, this approach, although good for implementations, involves some encoding for the native binding constructors. This brings to difficulties when reasoning about the syntax, see Section 7.1.3 below.

Less Related Work Scope graphs [4] are a recent language-independent framework for specifying bindings. This research is not concerned with definitional or reasoning principles, but with the integration of bindings with programming language parsers, compilers and static analyzers.

7.1.2 Complex Bindings

The literature on specification mechanisms for syntax with bindings offers a wide range of syntactic formats of various levels of sophistication, including those underlying CαML [64], Ott [68], Unbound [77], and Isabelle Nominal2 [75]. By contrast, we axiomatize binders through their fundamental properties and show that any binder satisfying the axiomatization can participate in binding-aware (co)datatypes. For example, the Isabelle Nominal2 package [75] is an impressive piece of engineering that caters for complex binders specified as recursive datatypes, but suffers from the lack of flexibility specific to syntactic formats. It cannot be combined with datatypes specified outside the framework, in particular, its nominal datatypes cannot nest standard (co)datatypes, as discussed in Section 1.5, Chapter 1. Our approach owes its generality and modularity to the use of category theory (and functors in particular). To our knowledge, our approach is the first in which category theory is used not only for the construction of (co)datatypes but also for capturing complex binders. Other category-theoretic frameworks [1, 29, 38, 46] focus on identifying the right category that has the generic syntax as the initial object. Thanks to this the syntax is characterized up to isomorphism and provided with an iterative structure. Our work is instead all about capturing the fundamental behaviour of (possibly complex) binding structures from literature, independently of their particularities. To this aim we first used signatures, but later realized that category theory would have provided us with better tools, in particular functors and their properties. Nonetheless we give as well a characterization of the syntax up to isomorphism, identifying it as an initial object in an appropriate category (Section 6.5, Chapter 6).

7.1.3 Reasoning and Definitional Infrastructure

Besides specification expressiveness, another criterion for assessing a formal framework is the amount of infrastructure built around the specification language, including reasoning and

definitional mechanisms. For syntactic approaches, the difficulty of providing such an infrastructure increases with the complexity of the supported binders. For example, Nominal Isabelle includes simple binders supported by induction [76] and recursion [72], whereas Isabelle Nominal2 provides complex binders but only induction. By contrast, our induction and recursion principles operate generically for arbitrary MRBNFs, regardless of the binders' complexity. For finitary syntax, our (co)induction and (co)recursion principles are as expressive as those of nominal logic, via the correspondence between MRBNFs and nominal sets described above; any predicate that is provable or function that is definable using one approach is also provable or definable with the other approach.

Binding-aware recursion is technically more complex than induction, given the requirement that one produces a function that is well-defined on alpha-quotiented terms. Here, the state of the art on high expressiveness is the nominal recursor [58] (implemented in Isabelle [72] and in Coq [5]), and the essential variation due to [53] (implemented in HOL4). Our recursor improves on the expressiveness of these recursors, by combining their respective strengths: It uses a flexible notion of (dynamic) parameter as in [53] with our improved Horn-style axiomatization (Chapter 3 Section 3.3)and circumvents the nominal recursor's limitation that freshness must be definable from the permutation action. In Section 6.7 of Chapter 6, we have formally proved these claims for the particular case of the λ-calculus.

The last paragraph only covers nominal-style recursors, where the recursive clauses refer to the native "first-order" binding operators, such as $\lambda : \alpha \times \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$ for the λ-calculus. This has the important advantage of manipulating terms in a natural way, reflecting the informal practice in describing logics and programming languages. Other recursors in the literature emphasize different constructors, such as $\lambda_{\text{de Bruijn}} : \alpha\ \boldsymbol{T} \to \alpha\ \boldsymbol{T}$ and $\lambda_{\text{weak HOAS}} : (\alpha \to \alpha\ \boldsymbol{T}) \to \alpha\ \boldsymbol{T}$. These circumvent the difficulties arising from quotienting, at the cost of having to filter out unwanted terms and introduce an encoding overhead. In the end, these different approaches target the same platonic concept of term, and our binding specification framework could in principle offer these alternative "views" of the term datatype by defining the alternative constructors and proving their associated recursors.

As exemplified above for the lambda constructor, what de Brujin, HOAS and other hybrid approaches (e.g. locally nameless) have in common is that they rely on some encoding of binding, as opposed to using native constructors—this is instead done by theories, like ours, taking the nominal approach. De Brujin indexes are a very good solution for implementation, but at the same time they are counterintuitive and heavy for human readability and hence for reasoning. The nameless approach does not offer support for operators that explicitly bind free variables, as in $\lambda x.t$. Instead, it uses name-free de Bruijn style operators, where in $\lambda_{\text{de Bruijn}}\, t$ the structure of t itself contains the slot for the binding. The explicit-variable operators can of course be defined from the de Bruijn ones, but the induction and recursion principles provided by the nameless datatypes lack a strong "awareness" about these operators, and therefore do not capture informal definitional and reasoning idioms such as Barendregt's variable convention. For a more detailed discussion, we refer to [9].

HOAS offers a smart alternative and it results more intuitive then the (locally) nameless approach, with the representation of variables as metavariables; however this process relies

heavily, as discussed above, on the binding construct of the metatheory and moreover needs adequacy proofs. These can be regarded as a formal connection between the two styles of constructors: the native first-order ones and the higher-order ones; unfortunately this is an heavy task to be mechanized and often is left to pen-on-paper practice.

For non-well-founded syntax with bindings such as infinite-depth λ-calculus terms (also known as Böhm trees [8]), coinduction seems to be largely unexplored territory—where we study and criticize parameter-based fresh coinduction, and produce an improved version based on dynamically changing binding variables in terms. By contrast, corecursion has been studied for Böhm trees [45] and more generally for nominal codatatypes [46]. However, these works impose the finite-support restriction even for infinite objects. With the help of regular cardinals, we are able to lift this artificial restriction. The idea of using cardinality was already present in [22], but in this work the aim was different: the author builds models that allow for infinite (but "small") support; these models form a sound and complete semantics for nominal logic and Herbrand theorems can be proved. A different approach to go beyond finite support has been taken by [30], as an infinitary extension of his previous work on nonstandard set-theoretic foundations of nominal logic.

7.2 Conclusion

The major ambition of our PhD project has been developing a general mathematical theory of syntax with bindings.

Syntaxes with binding mechanisms have been singularly employed and studied for decades in mathematical logic and to model computation. The arguably most famous example is Barendregt's book on λ-calculus [8]. In this work the calculus is carefully studied from its syntax with many results, to its operational and denotational semantics.

On the other hand, in regard to the formal treatment of concepts about binding independently of the particular syntax, the state of the art is unsatisfactory. If we think of a generic syntax with bindings as an algebraic structure, for example as we think of a generic group, what we have addressed is developing a general theory, where syntaxes with bindings are studied as groups in group theory. Each framework in literature (Section 7.1), to our knowledge, lack some important feature or is not general enough (e.g., does not capture some complex binding structure) to be up to the task.

Not only generality was among our motivations: we wanted our theory to be expressive. While capturing many instances, the theory should have been able to represent the essential mechanisms of the binding structures involved, thus giving the possibility to reason about these—in this regard the features we considered paramount for such a theory have been presented in Section 1.4 of Chapter 1 and then discussed throughout this document. Moreover we wanted that the end result was a tool that eased this reasoning. That is why we formalized it in a proof assistant: we checked every result in Isabelle/HOL and we are working to the development of a package that implements the theory and is easily instantiable.

Our work started with the study and consequent extension of an older framework (see Chapter 3) with which we identified the main features that should be asked to our theory

and obtained some important original results: a treatment of infinitely branching terms (with infinitely many free variables) based on cardinality theory (Sections 3.1.5 and 3.2.3) and a substitution-based recursion principle (Section 3.3), thanks to which we were able to define generally, independently of the syntax, an interpretation function in semantics domains (Section 3.3.4). For this first framework we decided to adopt a "universal algebra approach": a syntax is specified via a binding signature (Section 3.5), which also organizes statically terms in their respective sorts.

With the second framework for our theory (Chapters 5 and 6) we decided to abandon signatures. Now syntaxes are captured by mean of a special class of functors (Sections 6.1 and 6.2). This led to a significant improvement in terms of generality—on our previous framework, but also on the state of art:

- We achieve full modularity for our syntaxes (Section 6.2.9), being able to specify new syntaxes with bindings from previously defined ones.
- We are able to handle a wide variety of binding structures, of arbitrary complexity in a uniform way (Section 5.2).
- We extend our treatment of infinitary syntaxes also to non-well-founded ones (Section 6.2.8).
- We give an adaptation of fresh induction [76] to our framework and, symmetrically, a binding-aware coinduction principle (Section 6.4).
- We provide our theory with renaming-based (co)recursion principles, showing that the terms form initial (final) objects in an appropriate category (Section 6.5). Our recursion principle generalizes the nominal-style principles due to Pitts, Urban and Tasson, and Norrish, as well as the swapping-based recursor developed within our first framework (Section 6.7).

7.3 Future Work

The next step to be taken for our theory, is to get it into a fully functional package for Isabelle/HOL, with a fully automated interface for the user. This means that the user will be able to specify quickly their syntax, by just giving its constructors, and in exchange they will get a rich collection of lemmas about terms and operators on them (see Section 3.2.4 from Chapter 3), as well as bindings-aware reasoning and definition principles, all customized for the particular instance. The whole functorial setting will be hidden from the user and the properties of it will be phrased naturally in terms of the constructors of the syntax. This will include all the modularity properties, as well as mutually-inductively specified objects to get many sorted syntaxes. This process is theoretically straightforward, but technically demanding, as witnessed by its predecessor: the BNFs-based (co)datatype package for Isabelle/HOL [18, 70, 16, 13, 17].

On a more theoretical level, there are some additional features that we plan to add soon to our framework. In particular, we have shown how we implement and generalize to our functorial setting a swapping-based (co)recursion—as discussed, by now we should more

properly call it "*renaming-based* (co)recursion". However our second functorial framework still lacks a substitution-based (co)recursion, that generalizes our principle presented in Section 3.3, Chapter 3. After we formalize this result, we will also be able to port to the new framework a generalization of our previous interpretation of syntaxes in semantic domains (Section 3.3.4, Chapter 3)—the interpretation function will be defined independently of the particular syntax and a good behaviour with respect to substitution (the "substitution lemma") will be guaranteed once and for all.

As later developments, we plan to go beyond the study of datatypes and formalize binding-aware non-structural induction principles, such as rule induction—which is very well supported by Isabelle's nominal package [73]. For a dual rule coinduction principle, our study (Section 6.4.2, Chapter 6) suggests that fresh coinduction may not be the best notion to consider. Similar considerations hold for general recursion in the presence of bindings.

Appendix A

Appendix

A.1 More Details About BNFs and BNF-based (Co)datatypes

The properties of BNFs and their employment in the construction of modular (co)datatypes are described in [70] and [18]. Here we recall the reasoning principles emerging from these constructions, referring to the notations in Section 2.3.

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

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

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

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

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

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

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

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

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

$$(\alpha, \alpha\,T)\,F \overset{\mathsf{ctor}}{\to} \alpha\,T$$

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

(A) Applying the (co)datatype constructor

(B) Elements of the datatypes as piles of F-shapes

FIGURE A.1: Visualizing a datatype

themselves, i.e., applying set_2^F to them yields $\varnothing$. Fig. A.1b pictures a recursive component path of such a tree.

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

If instead of a datatype we consider the codatatype αT defined as $\alpha T \simeq^{\infty} (\alpha, \alpha T)F$, the pictures in Fig. A.1 remain relevant. The difference is that members of αT can now be unfolded into possibly *non-well-founded* trees—i.e., trees that are allowed to have infinite recursive-component paths, corresponding to an infinite number of applications of the constructor. As a result, induction is no longer a valid proof principle. However, we can take advantage of the fact that the tree obtained by fully unfolding a member t of the codatatype determines t uniquely—along the principle "to be is to do," where "to be" refers to t's identity and "to do" refers to t's unfolding behavior.[1] Thus, s and t will be equal whenever they are bisimilar as F-trees—that is, if there exists an F-bisimilarity relation $\varphi : \alpha T \to \alpha T \to \mathit{bool}$ such that $\varphi\ s\ t$ holds. The notion of φ being an F-bisimilarity means that, whenever φ relates two F-trees $\mathsf{ctor}\ x$ and $\mathsf{ctor}\ y$, their top F-layers have the same shapes, positionwise equal α-atoms, and positionwise φ-related components. As seen in Section 2.2, such positionwise relations can be expressed using the relator of F. The structural coinduction principle (SC) embodies the above reasoning pattern.

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

The above definitional modularity is matched by modularity with respect to proof principles: The (co)induction principle associated with a (co)datatype respects the abstraction barrier of the (co)datatypes nested in it, in that it does not refer to their definition or their constructors; instead, it only uses their BNF interfaces, consisting of map functions, set functions and relators. For example, here is the structural coinduction principle for finitely branching

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

possibly non-well-founded rose trees, defined as a codatatype by $\alpha\, tree_\infty \simeq \alpha \times (\alpha\, tree_\infty)\, list$, where for its constructor we write Node instead of ctor:

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

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

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

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

A.2 More Details on the (Co)recursive Definition of Substitution

We show the main proof obligations that must be discharged when defining the variable-for-variable substitution on $\overline{\alpha}\ \boldsymbol{T}$, i.e., the conditions from Definitions 73 and 76 instantiated with the corresponding substitution-specific definitions introduced in Sections 6.5.1 and 6.5.2. We omit the easy-to-prove obligations that the instantiations yield term-like structures or parameter structures.

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

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

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

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

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

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

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

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

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

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

Bibliography

[1] Benedikt Ahrens et al. “High-level signatures and initial semantics”. In: *CoRR* abs/1805.03740 (2018). arXiv: 1805.03740. URL: http://arxiv.org/abs/1805.03740.

[2] Guillaume Allais et al. “Type-and-scope safe programs and their proofs”. In: *CPP*. 2017, pp. 195–207.

[3] S. J. Ambler, Roy L. Crole, and Alberto Momigliano. “A definitional approach to primitive recursion over higher order abstract syntax”. In: *MERLIN*. 2003.

[4] Hendrik van Antwerpen et al. “A constraint language for static semantic analysis based on scope graphs”. In: *PEPM*. 2016, pp. 49–60.

[5] Brian E. Aydemir, Aaron Bohannon, and Stephanie Weirich. “Nominal Reasoning Techniques in Coq: Extended Abstract”. In: *Electr. Notes Theor. Comput. Sci.* 174.5 (2007), pp. 69–77.

[6] Brian E. Aydemir et al. “Engineering formal metatheory”. In: *POPL*. 2008, pp. 3–15.

[7] Brian E. Aydemir et al. “Mechanized Metatheory for the Masses: The PoplMark Challenge”. In: *TPHOLs*. 2005, pp. 50–65.

[8] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1984.

[9] Stefan Berghofer and Christian Urban. “A Head-to-Head Comparison of de Bruijn Indices and Names”. In: *Electronic Notes in Theoretical Computer Science* 174.5 (2007). LFMTP 2006, pp. 53 –67.

[10] Stefan Berghofer and Markus Wenzel. “Inductive Datatypes in HOL—Lessons Learned in Formal-Logic Engineering”. In: *TPHOLs ’99*. Vol. 1690. LNCS. 1999, pp. 19–36.

[11] Richard S. Bird and Ross Paterson. “De Bruijn Notation as a Nested Datatype”. In: *J. Funct. Program.* 9.1 (1999), pp. 77–91.

[12] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. “Cardinals in Isabelle/HOL”. In: *ITP*. 2014, pp. 111–127.

[13] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. “Foundational Extensible Corecursion”. In: *ICFP 2015*. ACM, 2015, pp. 192–204.

[14] Jasmin Christian Blanchette et al. *Binders as Bounded Natural Functors (Formal Scripts associtated to the homonymous paper.)* Available at: https://sites.google.com/view/lorgheri/research.

[15] Jasmin Christian Blanchette et al. “Bindings As Bounded Natural Functors”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 22:1–22:34. ISSN: 2475-1421. DOI: 10.1145/3290335. URL: http://doi.acm.org/10.1145/3290335.

[16] Jasmin Christian Blanchette et al. “Foundational nonuniform (co)datatypes for higher-order logic”. In: *LICS*. IEEE, 2017.

[17] Jasmin Christian Blanchette et al. “Friends with benefits: Implementing corecursion in foundational proof assistants”. In: *ESOP 2017*. LNCS. To appear. Available at http://andreipopescu.uk/pdf/amico.pdf. Springer, 2017.

[18] Jasmin Christian Blanchette et al. “Truly Modular (Co)datatypes for Isabelle/HOL”. In: *ITP*. 2014, pp. 93–110.

[19] N. de Bruijn. “Λ-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem”. In: *Indag. Math* 34.5 (1972), pp. 381–392.

[20] Lukas Bulwahn, Alexander Krauss, and Tobias Nipkow. “Finding Lexicographic Orders for Termination Proofs in Isabelle/HOL”. In: *Theorem Proving in Higher Order Logics*. Ed. by Klaus Schneider and Jens Brandt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 38–53. ISBN: 978-3-540-74591-4.

[21] Arthur Charguéraud. “The Locally Nameless Representation”. In: *J. Autom. Reasoning* 49.3 (2012), pp. 363–408.

[22] James Cheney. “Completeness and Herbrand theorems for nominal logic”. In: *J. Symbolic Logic* 71.1 (Mar. 2006), pp. 299–320. DOI: 10.2178/jsl/1140641176. URL: https://doi.org/10.2178/jsl/1140641176.

[23] Adam Chlipala. “Parametric higher-order abstract syntax for mechanized semantics”. In: *ICFP*. 2008, pp. 143–156.

[24] Alonzo Church. “A Formulation of the Simple Theory of Types”. In: *J. Symb. Logic* 5.2 (1940), pp. 56–68.

[25] Ernesto Copello et al. “Alpha-Structural Induction and Recursion for the Lambda Calculus in Constructive Type Theory”. In: *Electr. Notes Theor. Comput. Sci.* 323 (2016), pp. 109–124.

[26] Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. “Higher-Order Abstract Syntax in Coq”. In: *TLCA*. 1995, pp. 124–138.

[27] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. “The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations - Part 2 - A Survey”. In: *J. Autom. Reasoning* 55.4 (2015), pp. 307–372.

[28] Amy P. Felty and Brigitte Pientka. “Reasoning with Higher-Order Abstract Syntax and Contexts: A Comparison”. In: *ITP*. 2010, pp. 227–242.

[29] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. “Abstract Syntax and Variable Binding (Extended Abstract)”. In: *LICS*. 1999, pp. 193–202.

[30] Murdoch Gabbay. “A general mathematics of names”. In: *Inf. Comput.* 205.7 (2007), pp. 982–1011.

[31] Lorenzo Gheri and Andrei Popescu. "A Case Study in Reasoning about Syntax with Bindings: The Church-Rosser and Standardization Theorems". Draft, submitted. Available at https://sites.google.com/view/lorgheri/research.

[32] Lorenzo Gheri and Andrei Popescu. *A Case Study in Reasoning about Syntax with Bindings: The Church-Rosser and Standardization Theorems (website)*. http://andreipopescu.uk/papers/Lambda_CR_Std.html.

[33] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[34] Robert Harper, Furio Honsell, and Gordon Plotkin. "A Framework for Defining Logics". In: *LICS*. 1987, pp. 194–204.

[35] Matthew Hennessy and Robin Milner. "On Observing Nondeterminism and Concurrency". In: *ICALP*. 1980, pp. 299–309.

[36] André Hirschowitz and Marco Maggesi. "Modules over Monads and Linearity". In: *Logic, Language, Information and Computation*. Ed. by Daniel Leivant and Ruy de Queiroz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 218–237. ISBN: 978-3-540-73445-1.

[37] André Hirschowitz and Marco Maggesi. "Nested Abstract Syntax in Coq". In: *Journal of Automated Reasoning* 49.3 (2012), pp. 409–426.

[38] Martin Hofmann. "Semantical Analysis of Higher-Order Abstract Syntax". In: *LICS*. 1999, pp. 204–213.

[39] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. "Locales—A Sectioning Concept for Isabelle". In: *TPHOLs*. 1999, pp. 149–166.

[40] H. J. Keisler. *Model Theory for Infinitary Logic*. North-Holland, 1971.

[41] Alexander Krauss. "Certified Size-Change Termination". In: *Automated Deduction – CADE-21*. Ed. by Frank Pfenning. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 460–475. ISBN: 978-3-540-73595-3.

[42] Alexander Krauss. "Partial Recursive Functions in Higher-Order Logic". In: *IJCAR*. 2006, pp. 589–603.

[43] Ramana Kumar et al. "CakeML: A Verified Implementation of ML". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, 2014, pp. 179–191. ISBN: 978-1-4503-2544-8.

[44] Ramana Kumar et al. "Self-Formalisation of Higher-Order Logic". In: *Journal of Automated Reasoning* 56.3 (2016), pp. 221–259. ISSN: 1573-0670.

[45] Alexander Kurz et al. "An Alpha-Corecursion Principle for the Infinitary Lambda Calculus". In: *CMCS*. 2012, pp. 130–149.

[46] Alexander Kurz et al. "Nominal Coalgebraic Data Types with Applications to Lambda Calculus". In: *Logical Methods in Computer Science* 9.4 (2013).

[47] James McKinna and Robert Pollack. “Pure Type Systems Formalized”. In: *TLCA*. 1993.

[48] Dale Miller and Alwen Tiu. “A proof theory for generic judgments”. In: *ACM Transactions on Computational Logic* 6.4 (2005), pp. 749–783.

[49] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375.

[50] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge, 2001.

[51] Robin Milner. *Communication and concurrency*. Prentice Hall, 1989.

[52] Michael Norrish. “Mechanising lambda-calculus using a classical first order theory of terms with permutations”. In: *Higher-Order and Symbolic Computation* 19.2-3 (2006), pp. 169–195.

[53] Michael Norrish. “Recursive Function Definition for Types with Binders”. In: *TPHOLs*. 2004, pp. 241–256.

[54] Michael Norrish and René Vestergaard. “Proof Pearl: De Bruijn Terms Really Do Work”. In: *TPHOLs*.

[55] Lawrence C. Paulson. “A Mechanised Proof of Gödel’s Incompleteness Theorems Using Nominal Isabelle”. In: *Journal of Automated Reasoning* 55.1 (2015), pp. 1–37. ISSN: 1573-0670.

[56] Frank Pfenning and Conal Elliott. “Higher-Order Abstract Syntax”. In: *PLDI*. Ed. by Richard L. Wexelblat. 1988, pp. 199–208.

[57] Brigitte Pientka. “POPLMark reloaded: mechanizing logical relations proofs (invited talk)”. In: *CPP*. 2018, p. 1.

[58] Andrew M. Pitts. “Alpha-structural recursion and induction”. In: *J. ACM* 53.3 (2006).

[59] Andrew M. Pitts. “Nominal logic, a first order theory of names and binding”. In: *Inf. Comput.* 186.2 (2003), pp. 165–193.

[60] Gordon D. Plotkin. “Call-by-Name, Call-by-Value and the lambda-Calculus”. In: *Theor. Comput. Sci.* 1.2 (1975), pp. 125–159.

[61] Randy Pollack, Masahiko Sato, and Wilmer Ricciotti. “A Canonical Locally Named Representation of Binding”. In: *J. Autom. Reasoning* 49.2 (2012), pp. 185–207.

[62] Andrei Popescu. “Contributions to the theory of syntax with bindings and to process algebra”. PhD thesis, Univ. of Illinois, 2010. Available at `andreipopescu.uk/thesis.pdf`.

[63] Andrei Popescu and Elsa L. Gunter. “Recursion principles for syntax with bindings and substitution”. In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 2011, pp. 346–358.

[64] François Pottier. “An Overview of CαML”. In: *Electron. Notes Theor. Comput. Sci.* 148.2 (2006), pp. 27–52. ISSN: 1571-0661.

[65] J. J. M. M. Rutten. “Relators and Metric Bisimulations”. In: *Electr. Notes Theor. Comput. Sci.* 11 (1998), pp. 252–258.

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

[67] Andreas Schropp and Andrei Popescu. “Nonfree Datatypes in Isabelle/HOL - Animating a Many-Sorted Metatheory”. In: *CPP*. 2013, pp. 114–130.

[68] Peter Sewell et al. “Ott: Effective tool support for the working semanticist”. In: *J. Funct. Program.* 20.1 (2010), pp. 71–122.

[69] Masako Takahashi. “Parallel Reductions in lambda-Calculus”. In: *Inf. Comput.* 118.1 (1995), pp. 120–127.

[70] 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, pp. 596–605.

[71] Christian Urban. “Nominal Techniques in Isabelle/HOL”. In: *J. Autom. Reasoning* 40.4 (2008), pp. 327–356.

[72] Christian Urban and Stefan Berghofer. “A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL”. In: *IJCAR*. 2006, pp. 498–512.

[73] Christian Urban, Stefan Berghofer, and Michael Norrish. “Barendregt’s Variable Convention in Rule Inductions”. In: *CADE*. 2007, pp. 35–50.

[74] Christian Urban and Cezary Kaliszyk. “General Bindings and Alpha-Equivalence in Nominal Isabelle”. In: *ESOP*. 2011, pp. 480–500.

[75] Christian Urban and Cezary Kaliszyk. “General Bindings and Alpha-Equivalence in Nominal Isabelle”. In: *Logical Methods in Computer Science* 8.2 (2012).

[76] Christian Urban and Christine Tasson. “Nominal Techniques in Isabelle/HOL”. In: *CADE*. 2005, pp. 38–53.

[77] Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. “Binders unbound”. In: *ICFP*. 2011, pp. 333–345.

[78] Makarius Wenzel. “The Isabelle/Isar Reference Manual”. Available at `http://isabelle.in.tum.de/doc/isar-ref.pdf`. 2018.