
Article:
Gheri, L. and Popescu, A. orcid.org/0000-0001-8747-0619 (2020) A formalized general theory of syntax with bindings: extended version. Journal of Automated Reasoning, 64 (4). pp. 641-675. ISSN 0168-7433

https://doi.org/10.1007/s10817-019-09522-2

Noname manuscript No.
(will be inserted by the editor)

A Formalized General Theory of Syntax with Bindings

Extended Version

Lorenzo Gheri · **Andrei Popescu**

Received: date / Accepted: date

Abstract We present the formalization of a theory of syntax with bindings that has been developed and refined over the last decade to support several large formalization efforts. 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.

Keywords Syntax with bindings · Recursion and induction principles · Isabelle/HOL

1 Introduction

Syntax with bindings is an essential ingredient in the formal specification and implementation of logics and programming languages. However, 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 and is reflected in manifestos and benchmarks such as the influential POPLmark challenge [1].

In the past decade, in a framework developed intermittently starting with the second author's PhD [51] and moving into the first author's ongoing PhD, a series of results in logic and λ-calculus have been formalized in Isabelle/HOL [42, 44]. These include classic results (e.g., FOL completeness and soundness of Skolemization [15, 17, 19], λ-calculus standardization and Church-Rosser theorems [51], System F strong normalization [54]), as well as the meta-theory of Isabelle's Sledgehammer tool [10, 15].

Lorenzo Gheri
Department of Computer Science
Middlesex University London
E-mail: lg571@live.mdx.ac.uk

Andrei Popescu
Department of Computer Science
Middlesex University London
E-mail: a.popescu@mdx.ac.uk

In this paper, we present the Isabelle/HOL formalization of the framework itself, which is publicly available [27]. While concrete system syntaxes differ in their details, there are some fundamental phenomena concerning bindings that follow the same generic principles. It is these fundamental phenomena that our framework aims to capture, by mechanizing a form of universal algebra for bindings. The framework has evolved over the years through feedback from concrete application challenges: Each time a tedious, seemingly routine construction was encountered, a question arose as to whether this could be performed once and for all in a syntax-agnostic fashion.

The paper is structured as follows. We start with an example-driven overview of our design decisions (Section 2). Then we present the general theory: terms as alpha-equivalence classes of "quasiterms," standard operators on terms and their basic properties (Section 3), custom induction (Section 5) and recursion schemes (Section 4), including support for the semantic interpretation of syntax, and the sorting of terms according to a signature (Section 6). Finally, we briefly survey the various applications of the framework (Section 7). pointing out the usage of its various features. Within the large body of formalizations in the area (Section 8), distinguishing features of our work are the general setting (many-sorted signature, possibly infinitely branching syntax), a rich theory of the standard operators, and operator-aware recursion.

This paper is a substantially extended version of our conference paper presented at ITP 2017 [28]. The newly added material mostly expands the presentation of our main novel contributions: a rich theory of the operators and operator-aware recursion principles.

Operator properties: We give a quasi-exhaustive presentation of the properties we have proved about the syntactic operators and their interaction (in Subsection 3.4).

Recursion:

- We present two more recursion schemes, which factor in the swapping operator instead of, or in addition to, the substitution operator (in Subsection 4.1).
- We discuss primitive recursion, which is an extension of the iteration schemes (in a newly added Subsection 4.2).
- We introduce the skeleton operator, defined as an instance of one of our iteration schemes (in a newly added Subsection 4.3).
- We show the end-product scheme of many-sorted recursion (in a newly added Subsection 6.4).

Applications: We survey the applications of our theory (in a newly added Section 7).

2 Design Decisions

In this section, we use some examples to motivate our design choices for the theory. We also introduce conventions and notations that will be relevant throughout the paper.

The paradigmatic example of syntax with bindings is that of the λ-calculus [7]. We assume an infinite supply of variables, $x \in \mathbf{var}$. The λ-terms, $X, Y \in \mathbf{term}_\lambda$, are defined by the following BNF grammar:

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

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.

2.1 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 $\mathsf{Lam}\ (\mathsf{Abs}\ x\ X)$.

2.2 Freshness and substitution

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

- the freshness predicate, $\mathsf{fresh} : \mathbf{var} \to \mathbf{term}_\lambda \to \mathbf{bool}$, 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, $\_[\_/\_] : \mathbf{term}_\lambda \to \mathbf{term}_\lambda \to \mathbf{var} \to \mathbf{term}_\lambda$, 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 $\_[\_\wedge\_] : \mathbf{term}_\lambda \to \mathbf{var} \to \mathbf{var} \to \mathbf{term}_\lambda$, 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. Freshness, substitution and swapping are pervasive in the meta-theory of λ-calculus, as well as in most logical systems and formal semantics of programming languages. The basic properties of these operators lay at the core of important meta-theoretic results in these fields—our formalized theory aims at the exhaustive coverage of these basic properties.

2.3 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 $\mathbf{qterm}_\lambda$ and $\mathbf{qabs}_\lambda$ be the types of quasiterms and quasiabstractions in λ-calculus. Here, the quasi-abstraction constructor, $\mathsf{qAbs} : \mathbf{var} \to \mathbf{qterm}_\lambda \to \mathbf{qabs}_\lambda$, is a free constructor, of the kind produced by standard datatype specifications [9, 13]. The types $\mathbf{term}_\lambda$ and $\mathbf{abs}_\lambda$ are $\mathbf{qterm}_\lambda$ and $\mathbf{qabs}_\lambda$ quotiented to alpha. We prove compatibility of qAbs with alpha and then define $\mathsf{Abs} : \mathbf{var} \to \mathbf{term}_\lambda \to \mathbf{abs}_\lambda$ 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 com-

positional) and (2) Barendregt's variable convention [7] (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. To address these obligations, we formalize state-of-the-art techniques from the literature [48, 53, 68].

2.4 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) [31] 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 [41] we have channel names but no process variables, etc.

Sortedness is important, but formally quite heavy. In our formalization, we postpone dealing with it 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."

2.5 Possibly infinite branching

Nominal Logic's [47, 68] notion of finite support has become central in state-of-the-art techniques for reasoning about bindings. Occasionally, however, important developments step outside finite support. For example, (a simplified) CCS [40] has the following syntactic categories of data expressions $E \in \mathbf{exp}$ and processes $P \in \mathbf{proc}$:

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

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 modeling 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 [36] and Hennessey-Milner logic [32]. In our theory, we cover such infinitely branching syntaxes.

3 General Terms with Bindings

We start the presentation of our formalized theory, in its journey from quasiterms (3.1) to terms via alpha-equivalence (3.2). The journey is fueled by the availability of fresh variables, ensured by cardinality assumptions on constructor branching and variables (3.3). It culminates with a systematic study of the standard term operators (3.4).

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

Fig. 1 Alpha-Equivalence

3.1 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**:

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

Thus, any quasiabstraction has the form $\mathsf{qAbs}\ xs\ x\ X$, 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 $\mathsf{qVar}\ xs\ x$, of a variable x of varsort xs, or has the form $\mathsf{qOp}\ \delta\ inp\ binp$, i.e., consists of an operation symbol applied to some inputs that can be either free, *inp*, or bound, *binp*.

We use $(\alpha,\beta)\,\mathbf{input}$ as a type synonym for $\alpha \to \beta\ \mathbf{option}$, the type of partial functions from α to β; such a function returns either None (representing "undefined") or $\mathsf{Some}\ b$ for $b : \beta$. This type models inputs to the quasiterm constructors of varying number of arguments. An operation symbol δ : **opsym** can be applied, via qOp, to: (1) a varying number of free inputs, i.e., families of quasiterms modeled as members of $(\mathbf{index}, \mathbf{qterm})\,\mathbf{input}$ and (2) a varying number of bound inputs, i.e., families of quasiabstractions modeled as members of $(\mathbf{index}, \mathbf{qabs})\,\mathbf{input}$. For example, taking **index** to be **nat** we capture n-ary operations for any n (passing to $\mathsf{qOp}\ \delta$ inputs defined only on $\{0,\ldots,n-1\}$), as well as as countably-infinitary operations (passing to $\mathsf{qOp}\ \delta$ inputs defined on the whole **nat**).

Note that, so far, we consider sorts of variables but not sorts of terms. The latter will come much later, in Section 6, when we introduce signatures. Then, we will gain control (1) on which varsorts should be embedded in which term sorts and (2) on 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.

On quasiterms, we define freshness, 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, alpha : **qterm** $\to$ **qterm** $\to$ **bool**—and corresponding operators on quasiabstractions: $\mathsf{qFreshAbs}$, $\mathsf{alphaAbs}$, 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 ρ : **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.

3.2 Alpha-equivalence

We define the predicates alpha (on quasiterms) and alphaAbs (on quasiabstractions) mutually recursively, as shown in Fig. 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 $P : \beta^2 \to \mathbf{bool}$, we write $\uparrow P$ for its lifting to $(\alpha,\beta)\,\mathbf{input}^2 \to \mathbf{bool}$, defined as

$\uparrow P\ inp\ inp' \iff$
$\forall i.$ case $(inp\ i, inp'\ i)$ 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 1. Throughout this paper, 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. 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 [48].

3.3 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:

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

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 $|\mathbf{var}|$ 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 $|\mathbf{var}|$, so to conclude we would need the following: *A union of sets smaller than* $|\mathbf{var}|$ *indexed by a set smaller than* $|\mathbf{var}|$ *stays smaller than* $|\mathbf{var}|$. It turns out that this is a well-studied property of cardinals, called *regularity*—with guaranteed that for every cardinal there exists a regular cardinal bigger than that and with $|\mathbf{nat}|$ being the smallest regular cardinal. Thus, the desirable generalization of countability is regularity (which is available from Isabelle's cardinal library [16]). Henceforth, we will assume:

Assumption 2. $|\mathbf{var}|$ is a regular cardinal.

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

Prop 3. $\mathsf{qGood}\ X \Longrightarrow \forall xs.\ |\{x.\ \mathsf{qFresh}\ xs\ x\ X\}| = |\mathbf{var}|$

Table 1 Constructors and Operators on Terms and Abstractions

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**

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 [6, 39, 46].)

Prop 4. The following hold:
(1) alpha and alphaAbs are equivalences on good quasiterms and quasiabstractions
(2) The predicates defined by replacing, in Fig. 1's definition, the abstraction case with

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

coincide with alpha and alphaAbs.

3.4 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. Table 1 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 Section 2.1, 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. 4(2), respectively:

Prop 5. Assume good X and good X'. Then the following are equivalent:
(1) $\mathsf{Abs}\ xs\ x\ X = \mathsf{Abs}\ xs'\ x'\ X'$
(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:

Prop 6. Assume good X and good X'. Then the following hold:
(1) $y \notin \{x, x'\} \wedge \mathsf{fresh}\ xs\ y\ X \wedge \mathsf{fresh}\ xs\ y\ X' \wedge X\,[(\mathsf{Var}\ xs\ y)\,/\,x]_{xs} = X'\,[(\mathsf{Var}\ xs\ y)\,/\,x']_{xs} \Longrightarrow \mathsf{Abs}\ xs\ x\ X = \mathsf{Abs}\ xs\ x'\ X'$
(2) $\mathsf{fresh}\ xs\ y\ X \implies \mathsf{Abs}\ xs\ x\ X = \mathsf{Abs}\ xs\ y\ (X\,[(\mathsf{Var}\ xs\ y)\,/\,x]_{xs})$

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):

Prop 7. Assume good X, $\uparrow\mathsf{good}\ inp$, $\uparrow\mathsf{good}\ binp$, $|\mathsf{dom}\ inp| < |\mathbf{var}|$ and $|\mathsf{dom}\ binp| < |\mathbf{var}|$. The following hold:
(1) $(ys, y) \neq (xs, x) \implies \mathsf{fresh}\ ys\ y\ (\mathsf{Var}\ xs\ x)$
(2) $\uparrow(\mathsf{fresh}\ ys\ y)\ inp \wedge \uparrow(\mathsf{freshAbs}\ ys\ y)\ binp \implies \mathsf{fresh}\ ys\ y\ (\mathsf{Op}\ \delta\ inp\ binp)$
(3) $(ys, y) = (xs, x) \vee \mathsf{fresh}\ ys\ y\ X \implies \mathsf{freshAbs}\ ys\ y\ (\mathsf{Abs}\ xs\ x\ X)$

Here and elsewhere, when dealing with Op, we make cardinality assumptions on the domains of the inputs to make sure the terms $\mathsf{Op}\ \delta\ inp\ binp$ are good.

We can further improve on Prop. 7, obtaining "iff" facts that resemble a primitively recursive definition of freshness (as a function):

Prop 8. Prop. 7 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.

Prop 9. Assume good X, $\uparrow\mathsf{good}\ inp$, $\uparrow\mathsf{good}\ binp$, $|\mathsf{dom}\ inp| < |\mathbf{var}|$ and $|\mathsf{dom}\ binp| < |\mathbf{var}|$. 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

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

Prop 10. Assume good X, good Y, $\uparrow\mathsf{good}\ inp$, $\uparrow\mathsf{good}\ binp$, $|\mathsf{dom}\ inp| < |\mathbf{var}|$ and $|\mathsf{dom}\ binp| < |\mathbf{var}|$. 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) $(xs, x) \neq (ys, y) \wedge \mathsf{fresh}\ xs\ x\ Y \implies (\mathsf{Abs}\ xs\ x\ X)\,[Y/y]_{ys} = \mathsf{Abs}\ xs\ x\ (X\,[Y/y]_{ys})$

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

Prop 11. Assume good X, $\uparrow\mathsf{good}\ inp$, $\uparrow\mathsf{good}\ binp$, $\uparrow\mathsf{good}\ \rho$, $|\mathsf{dom}\ inp| < |\mathbf{var}|$, $|\mathsf{dom}\ binp| < |\mathbf{var}|$ and $|\mathsf{dom}\ \rho| < |\mathbf{var}|$. 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) $\uparrow(\mathsf{fresh}\ xs\ x)\ \rho \implies (\mathsf{Abs}\ xs\ x\ X)\,[\rho] = \mathsf{Abs}\ xs\ x\ (X\,[\rho])$

Above, the notation $\uparrow(\mathsf{fresh}\ xs\ x)\ \rho$ follows the spirit of Convention 1, in that it lifts the freshness predicate from terms to environments for parallel substitution, i.e., partial variable-to-term assignments ρ : **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.

Prop 12 (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:

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

Prop 13 (Swapping versus Freshness)**.** Assume $\mathsf{good}\ X$. The following hold:
(1) Swapping preserve 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 they should not be bothered with having to work with the much heavier parallel substitution properties.

Prop 14 (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** $\rightarrow$ **var** $\rightarrow$ **term option**. Point (2) follows easily from point (1).

Prop 15 (Parallel Substitution versus Freshness)**.** Assume good X and $\uparrow$good ρ. The following hold:

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

In the unary case we obtain three separate properties:

Prop 16 (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 preserve 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:

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

Prop 17 (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.

Prop 18 (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 $\rho[y \leftarrow Y[\rho]]$ 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.

4 Operator-Sensitive Recursion

In this section we present several definition principles for functions having terms and abstractions as their domain. The principles we formalize are generalizations to an arbitrary syntax of results that have been previously described for the particular syntax of λ-calculus [45, 53]. 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.

We start with the simpler-structured *iteration* principles (4.1) followed by their extension to primitive recursion (4.2). 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 (4.3). The second employs freshness-substitution-based iteration to produce a whole class of instances: the interpretation of syntax in semantic domains (4.4).

4.1 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, $\mathbf{T}$ and $\mathbf{A}$
- operations corresponding to the constructors: $\mathsf{VAR} : \mathbf{varsort} \to \mathbf{var} \to \mathbf{T}$, $\mathsf{OP} : \mathbf{opsym} \to (\mathbf{index}, \mathbf{T})\ \mathbf{input} \to (\mathbf{bindex}, \mathbf{A})\ \mathbf{input} \to \mathbf{T}$, $\mathsf{ABS} : \mathbf{varsort} \to \mathbf{var} \to \mathbf{T} \to \mathbf{A}$
- operations corresponding to freshness and substitution: $\mathsf{FRESH} : \mathbf{varsort} \to \mathbf{var} \to \mathbf{T} \to \mathbf{bool}$, $\mathsf{FRESHABS} : \mathbf{varsort} \to \mathbf{var} \to \mathbf{A} \to \mathbf{bool}$, $\mathsf{SUBST} : \mathbf{T} \to \mathbf{T} \to \mathbf{var} \to \mathbf{varsort} \to \mathbf{T}$ and $\mathsf{SUBSTABS} : \mathbf{A} \to \mathbf{T} \to \mathbf{var} \to \mathbf{varsort} \to \mathbf{A}$

and it is required to satisfy the following freshness clauses F1–F5 (analogous to the implicational simplification rules for freshness in Prop. 7), substitution clauses Sb1–Sb4 (analogous to the simplification rules for substitution in Prop. 10), and substitution-renaming clause SbRn (analogous to the substitution-based abstraction equality rule in Prop. 6(2)):

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: $\mathsf{FRESH}\ ys\ y\ X \implies \mathsf{FRESHABS}\ ys\ y\ (\mathsf{ABS}\ xs\ x\ X)$
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: $\mathsf{FRESH}\ xs\ y\ X \implies \mathsf{ABS}\ xs\ x\ X = \mathsf{ABS}\ xs\ y\ (\mathsf{SUBST}\ (\mathsf{VAR}\ xs\ y)\ x\ xs\ X)$

Theorem 19. The good terms and abstractions form the *initial FSb model*. Namely, for any FSb model as above, there exist the functions $f : \mathbf{term} \to \mathbf{T}$ and $fAbs : \mathbf{abs} \to \mathbf{A}$ that

commute, on good terms, with the constructors and with substitution and preserve freshness:

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

In addition, the two functions are uniquely determined on good terms and abstractions, in that, for all other functions $g : \mathbf{term} \to \mathbf{T}$ and $gAbs : \mathbf{abs} \to \mathbf{A}$ 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 $\mathsf{Nil} : \mathbf{G\ list}$ and $\mathsf{Cons} : \mathbf{G} \to \mathbf{G\ list} \to \mathbf{G\ list}$. To define, by iteration, a function from lists, say, $\mathsf{length} : \mathbf{G\ list} \to \mathbf{nat}$, 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 $\mathsf{NIL} : \mathbf{nat}$ to be 0 and $\mathsf{CONS} : \mathbf{G} \to \mathbf{nat} \to \mathbf{nat}$ 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 it maps these concepts to corresponding concepts on the target domain.

Michael Norrish has developed a similar principle that employs swapping instead of substitution [45]. We have also formalized this in our framework—in a slightly restricted form, namely without factoring in fixed variables and parameters.

A *fresh-swapping model* is a structure similar to our fresh-substitution models, just that instead of the substitution-like operators, SUBST and SUBSTABS, it features swapping-like operators, $\mathsf{SWAP} : \mathbf{T} \to \mathbf{var} \to \mathbf{var} \to \mathbf{varsort} \to \mathbf{T}$ and $\mathsf{SWAPABS} : \mathbf{A} \to \mathbf{var} \to \mathbf{var} \to \mathbf{varsort} \to \mathbf{A}$, assumed to satisfy rules corresponding to those for simplifying term swapping:

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: $\mathsf{SWAPABS}\ (\mathsf{ABS}\ xs\ x\ X)\ z_1\ z_2\ zs\ =\ \mathsf{ABS}\ xs\,(x\,[z_1 \wedge z_2]_{xs,zs})\,(\mathsf{SWAP}\ X\ z_1\ z_2\ zs)$

and, instead of the substitution-based variable-renaming property, the following swapping-based congruence rule for abstractions:

SwCong: $\mathsf{FRESH}\ xs\ y\ X\ \wedge\ \mathsf{FRESH}\ xs\ y\ X'\ \wedge\ \mathsf{SWAP}\ X\ y\ x\ ys = \mathsf{SWAP}\ X'\ y\ x'\ ys\ \Longrightarrow\ \mathsf{ABS}\ xs\ x\ X = \mathsf{ABS}\ xs\ x'\ X'$

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

Theorem 20. The good terms and abstractions form the initial FSw model. Namely, there exists a pair of functions $f : \mathbf{term} \to \mathbf{T}$ and $fAbs : \mathbf{abs} \to \mathbf{A}$ that commute, on good terms, with the constructors and preserve freshness—similarly to how it is described in Theorem 19, 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 19).

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 21. The good terms and abstractions form the *initial* FSbSw model. Namely, there exists a pair of functions $f : \mathbf{term} \to \mathbf{T}$ and $fAbs : \mathbf{abs} \to \mathbf{A}$ that commute, on good terms, with the constructors, substitution, swapping and preserve freshness, as described in Theorems 19 and 20. In addition, the two functions are uniquely determined on good terms and abstractions.

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.

4.2 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 $\mathsf{pred} : \mathbf{nat} \to \mathbf{nat}$ 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 $\mathsf{pred}' : \mathbf{nat} \to \mathbf{nat} \times \mathbf{nat}$ by iteration, $\mathsf{pred}'\ 0 = (0,0)$ and $\mathsf{pred}'\,(\mathsf{Suc}\ n) =$ 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 [51, §1.4.2] for a description of this phenomenon for the general case of initial models in Horn theories. 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 19, where we highlight the additions compared to iteration. (The other two primitive recursion theorems are obtained similarly from Theorems 20 and 21.)

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

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

– 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: FRESH : **varsort** $\to$ **var** $\to$ **term** $\to$ **T** $\to$ **bool** and 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 22. 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 19, *mutatis mutandis*. For example:

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

4.3 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.

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

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 skel : **term** $\to$ (**index**, **bindex**) **tree** and skelAbs : **abs** $\to$ (**index**, **bindex**) **tree** by the following mutually recursive clauses:

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

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

- FRESH : **varsort** $\to$ **var** $\to$ **tree** $\to$ **bool**, FRESHABS : **varsort** $\to$ **var** $\to$ **atree** $\to$ **bool**
- SWAP : **tree** $\to$ **var** $\to$ **var** $\to$ **varsort** $\to$ **tree**, SWAPABS : **atree** $\to$ **var** $\to$ **var** $\to$ **varsort** $\to$ **atree** and/or
- SUBST : **tree** $\to$ **tree** $\to$ **var** $\to$ **varsort** $\to$ **tree**, SUBSTABS : **atree** $\to$ **tree** $\to$ **var** $\to$ **varsort** $\to$ **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:

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

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

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

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 20, 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 $\delta =$ 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 20 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 5.

4.4 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 23. 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:

- $\mathsf{sem}\,(\mathsf{Var}\ xs\ x)\,\rho = \rho\ xs\ x$
- $\mathsf{sem}\,(\mathsf{Op}\ \delta\ inp\ binp)\,\rho = \mathsf{op}\ \delta\ (\uparrow(\mathsf{sem}\ \_\ \rho)\ inp)\ (\uparrow(\mathsf{semAbs}\ \_\ \rho)\ binp)$
- $\mathsf{semAbs}\,(\mathsf{Abs}\ xs\ x\ X)\,\rho = \mathsf{abs}\ xs\ (\lambda d.\ \mathsf{sem}\ X\ (\rho[(xs,x) \leftarrow d]))$,
  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 23 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 19: 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 23 (which we set out to prove). For example, FRESH was defined so that the freshness property listed in Theorem 23 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 19, what we are left to check in order to prove Theorem 23 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.

5 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. 16(a), we use (mutual) structural induction over terms and abstractions, proving the statement together with the corresponding statement for abstractions,

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

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. 10(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. 8(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. 8(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 [48, 66, 68], we also know how to apply this morally correct convention fully rigorously. To capture it in our formalization, we model parameters p : **param** as anything that allows for a notion of freshness, or, alternatively, provides a set of (free) variables for each varsort, varsOf : **param** $\rightarrow$ **varsort** $\rightarrow$ **var set**. With this, a "fresh induction" principle can be formulated, if all parameters have fewer variables than $|\mathbf{var}|$ (in particular, if they have only finitely many).

Theorem 24. Let φ : **term** $\rightarrow$ **param** $\rightarrow$ **bool** and φAbs : **abs** $\rightarrow$ **param** $\rightarrow$ **bool**. Assume:
(1) $\forall xs, p.\ |\mathsf{varsOf}\ xs\ p| < |\mathbf{var}|$
(2) $\forall xs, x, p.\ \varphi\ (\mathsf{Var}\ xs\ x)\ p$
(3) $\forall \delta, inp, binp, p.\ |\mathsf{dom}\ inp| < |\mathbf{var}| \wedge |\mathsf{dom}\ binp| < |\mathbf{var}| \wedge \uparrow(\lambda X.\ \mathsf{good}\ X \wedge (\forall q.\ \varphi\ X\ q))$

$inp \wedge \uparrow(\lambda A.\ \mathsf{goodAbs}\ A \wedge (\forall q.\ \varphi Abs\ A\ q))\ binp \Longrightarrow \varphi\ (\mathsf{Op}\ \delta\ inp\ binp)\ p$

(4) $\forall xs, x, X, p.\ \mathsf{good}\ X \wedge \varphi\ X\ p \wedge x \notin \mathsf{varsOf}\ xs\ p \Longrightarrow \varphi Abs\ (\mathsf{Abs}\ xs\ x\ X)\ p$

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. 16(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 $\mathbf{param} = (\mathbf{varsort} \times \mathbf{var})^2 \times \mathbf{term}$ and $\mathsf{varsOf}\ xs\ ((ys, y), (zs, z), Y) = \{y \mid xs = ys\} \cup \{z \mid xs = zs\} \cup \{x \mid \neg\, \mathsf{fresh}\ xs\ x\ Y\}$.

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.

6 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 (6.1), we classify terms according to sorts (6.2) and prove that well-sorted terms are good (6.3)—this gives us sorted versions of all theorems (6.5).

6.1 Binding signatures

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

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

Thus, a signature prescribes which varsorts correspond to which sorts (as discussed in Section 2.4) 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 in Section 2, we will write $(i_1 \mapsto a_1, \ldots, i_n \mapsto a_n)$ for the partial function that sends each i_k to $\mathsf{Some}\ a_k$ and everything else to None. In particular, $()$ denotes the totally undefined function.

For the λ-calculus syntax, we take $\mathbf{index} = \mathbf{bindex} = \mathbf{nat}$, $\mathbf{varsort} = \mathbf{sort} = \{\mathsf{lam}\}$ (a singleton datatype), $\mathbf{opsym} = \{\mathsf{App}, \mathsf{Lam}\}$, 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: $\mathsf{arOf}\ \mathsf{App} = (0 \mapsto \mathsf{lam}, 1 \mapsto \mathsf{lam})$, $\mathsf{barOf}\ \mathsf{App} = ()$. 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: $\mathsf{arOf}\ \mathsf{Lam} = ()$, $\mathsf{barOf}\ \mathsf{Lam} = (0 \mapsto (\mathsf{lam}, \mathsf{lam}))$.

For the CCS example in Section 2.5, 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 **sort** $= \{\mathsf{exp}, \mathsf{proc}\}$. Since we have expression variables but no process variables, we take **varsort** $= \{\mathsf{varexp}\}$ and asSort to send varexp to exp. We define **opsym** as the following datatype: **opsym** $=$ Zero | Plus | Inp **chan** | Out **chan** | $\sum$ (**index set**). The free and bound arities and sorts of the operation symbols are as expected. For example, 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 : **index set**, the arity is only defined for elements of I, namely $\mathsf{arOf}\,(\sum I) = ((i \in I) \mapsto \mathsf{proc})$.

6.2 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 wls : **sort** $\to$ **term** $\to$ **bool** and wlsAbs : **varsort** $\to$ **sort** $\to$ **abs** $\to$ **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:

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

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 $X = \mathsf{Var}\ \mathsf{lam}\ x$, $A = \mathsf{Abs}\ \mathsf{lam}\ x\ X$, and $Y = \mathsf{Op}\ \mathsf{Lam}\ ()\ (0 \mapsto A)$. These correspond to what, in the unsorted BNF notation from Section 2.1, 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 $\mathsf{wls}\ \mathsf{lam}\ X$ and $\mathsf{wls}\ \mathsf{lam}\ Y$) 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 $E = \mathsf{Op}\ \mathsf{Zero}\,()\,()$ and $F = \mathsf{Op}\ \mathsf{Plus}\ (0 \mapsto E,\ 1 \mapsto E)\,()$ 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 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 Out c, $(0 \mapsto \mathsf{exp}, 1 \mapsto \mathsf{proc})$.

6.3 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 25. $|\mathbf{sort}| < |\mathbf{var}| \wedge |\mathbf{index}| < |\mathbf{var}| \wedge |\mathbf{bindex}| < |\mathbf{var}|$

Now we can prove:

Prop 26. $(\mathsf{wls}\ s\ X \Longrightarrow \mathsf{good}\ X) \wedge (\mathsf{wls}\ (xs, s)\ A \Longrightarrow \mathsf{goodAbs}\ A)$

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. 26, we transfer the entire theory of Sections 3.4, 5 and 4 from good terms to well-sorted terms—e.g., Prop. 18(d) becomes:

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

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 6.4.

There is an important remaining question: Are our two Assumptions (2 and 25) satisfiable? That is, can we find, for any types **sort**, **index** and **bindex**, a type **var** larger than these such that $|\mathbf{var}|$ is regular? Fortunately, the theory of cardinals again provides us with a positive answer: Let $\mathbf{G} = \mathbf{nat} + \mathbf{sort} + \mathbf{index} + \mathbf{bindex}$. Since any successor of an infinite cardinal is regular, we can take **var** to have the same cardinality as the successor of $|\mathbf{G}|$, 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**.

6.4 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 $\mathsf{wls}^{\mathbf{T}} : \mathbf{sort} \to \mathbf{T} \to \mathbf{bool}$ and $\mathsf{wlsAbs}^{\mathbf{T}} : \mathbf{varsort} \to \mathbf{sort} \to \mathbf{A} \to \mathbf{bool}$
- the assumption that all operators preserve sorting, e.g.,
  $\mathsf{wls}\ s\ X' \wedge \mathsf{wls}\ (\mathsf{asSort}\ ys)\ Y' \wedge \mathsf{wls}^{\mathbf{T}}\ s\ X \wedge \mathsf{wls}^{\mathbf{T}}\ (\mathsf{asSort}\ ys)\ Y$
  $\implies \mathsf{wls}^{\mathbf{T}}\ s\ (\mathsf{SUBST}\ X'\ X\ Y'\ Y\ y\ ys)$

The recursion Theorem 22 is now extended to take sorting into account:

Theorem 27. For any sorted FSb recursion model, there exist the functions $f : \mathbf{term} \to \mathbf{T}$ and $fAbs : \mathbf{abs} \to \mathbf{A}$ that satisfy the same properties as in Theorem 22 and additionally preserve sorting:

- $\mathsf{wls}\ s\ X \implies \mathsf{wls}^{\mathbf{T}}\ s\ (f\ X)$
- $\mathsf{wls}\ s\ X \implies \mathsf{wlsAbs}^{\mathbf{T}}\ s\ (fAbs\ X)$

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 4.4, plus sorting predicates $\mathsf{wls}^{\mathbf{Dt}}$ and $\mathsf{wls}^{\mathbf{Da}}$. Again, it is assumed that the semantic operators preserve sorting. Then Theorem 23 is adapted to sorted domains, additionally ensuring the sort preservation of sem and semAbs.

6.5 End product

All in all, our formalization provides a theory of syntax with bindings over an arbitrary many-sorted signature. The signature is formalized as an Isabelle locale [35] 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 2 and 25). 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 5 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 24 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 4.1 and 4.2) and the interpretation Theorem 23, which require conceptually straightforward but technically tedious moves back and forth between sorted terms and sorted elements of the target domain.

7 Applications of the Framework

So far, we instantiated our theory to the syntaxes of the call-by-name and call-by-value variants of the λ-calculus (the latter differing from the former by a separate syntactic category for values) and to that of many-sorted FOL.

The first application was developed in 2010, when we formalized a proof of strong normalization for System F [54]. The two employed syntaxes, of System F's Curry-style terms and types, are two copies of the λ-calculus syntax. The logical relation technique required in the proof made essential use of parallel substitution—and in fact was the incentive for us to go beyond unary substitution in the general theory. To streamline the development, on top of the first-order syntax we introduced HOAS-like definition and reasoning techniques, which were based on the general-purpose first-order ones shown in Sections 4 and 5.

In subsequent work, we formalized several results about λ-calculus: the standardization and Church-Rosser theorems and the CPS translations between call-by-name and call-by-value calculi [49], an adequate HOAS representation of the calculus into itself, a sound interpretation via de Bruijn encodings [22], and the isomorphism between different definitions of λ-terms: ours, the Nominal one [68], the locally named one [50] and the Hybrid one [24]. These results are centered around some translation/representation functions: CPS, HOAS, Church-Rosser complete development [63], de Bruijn, etc.—these functions and their desirable properties (e.g., preservation of substitution, crucial in HOAS [31]) were obtained as instances of our recursion theorems (Section 4).[2]

Finally, in the context of certifying Sledgehammer's HOL to FOL encodings [11], we formalized fundamental results in many-sorted FOL [15], including the completeness, compactness, Skolemization and downward Löwenheim-Skolem theorems.[3] Besides the ubiquitous employment of the properties of freshness of substitution from Section 3.4, we used

[2] The formalization work mentioned in this paragraph is mostly unpublished, although aspects concerning the involved recursive definitions are discussed in [51, 53].

[3] This work was the first entry in the (today very prolific) IsaFoL project [2].

the interpretation Theorem 23 for the FOL semantics and the recursion Theorem 22 for bootstraping quickly the (technically quite tricky) Skolemization function.

8 Discussion, Related Work and Future Work

There is a large amount of literature on formal approaches to syntax with bindings. (See [1, §2], [24, §6] and [51, §2.10,§3.7] for overviews.) Our work, nevertheless, fills a gap in the literature: It is the first theory of binding syntax mechanized in a universal algebra fashion, i.e., with sorts and many-sorted term constructors specified by a binding signature, as employed in several theoretical developments, e.g., [25, 48, 57, 62]. The universal algebra aspects of our approach are the consideration of an *arbitrary signature* and the singling out of the collection of terms and the operations on them as an *initial object in a category of models/algebras* (which yields a recursion principle). We do not consider arbitrary equational varieties (like in [62]), but only focus on selected equations and Horn clauses that characterize the term models (like in [48]).

Alternatives to universal algebra A popular alternative to our universal algebra approach is higher-order abstract syntax (HOAS) [20, 23, 24, 30, 31]: the reduction of all bindings to a single binding—that of a fixed λ-calculus. Compared to universal algebra, HOAS's advantage is lighter formalizations, whereas the disadvantage is the need to prove the representation's adequacy (which involves reasoning about substitution) and, in some frameworks, the need to rule out the exotic terms.

Another alternative, very successfully used in HOL-based provers such as HOL4 [61] and Isabelle/HOL, is the "package" approach: Instead of deeply embedding sorts and operation symbols like we do, packages take a user specification of the desired types and operations and prove all the theorems for that instance (on a dynamic basis). Nominal Isabelle [65, 67] is a popular such package, which implements terms with bindings for Isabelle/HOL. From a theoretical perspective, a universal algebra theory has a wider appeal, as it models "statically" the meta-theory in its whole generality. However, a package is more practical, since most proof assistant users only care about the particular instance syntax used in their development. In this respect, simply instantiating our signature with the particular syntax is not entirely satisfactory, since it is not sufficiently "shallow"—e.g., one would like to have actual operations such as Lam instead of applications of Op to a Lam operation symbol, and would like to have actual types, such as **exp** and **proc**, instead of the well-sortedness predicate applied to sorts, wls exp and wls proc. For our applications, so far we have manually transited from our "deep" signature instances to the more usable shallow version sketched above. In the future, we plan to have this transit process automated, obtaining the best of both worlds, namely a universal algebra theory that also acts as a *statically certified* package. (This approach has been prototyped for a smaller theory: that of nonfree equational datatypes [59].)

Theory of substitution and semantic interpretation The main goal of our work was the development of as much as possible from the theory of syntax for an arbitrary syntax. To our knowledge, none of the existing frameworks provides support for substitution and the interpretation of terms in semantic domains at this level of generality. Formalizations for concrete syntaxes, even those based on sophisticated packages such as Nominal Isabelle or the similar tools and formalizations in Coq [5,6,33], have to redefine these standard concepts and prove their properties over and over again—an unnecessary consumption of time and brain power. An exception is the recent development of Coq's Autosubst library [58], which automates the definition of parallel substitution for terms in de Bruijn encoding.

Induction and recursion principles There is a rich literature on these topics, which are connected to the quest, pioneered by Gordon and Melham [29], of understanding terms with bindings modulo alpha as an abstract datatype. We formalized the Nominal structural induction principle from [48], which has also been implemented in Nominal Isabelle, Coq [5] and recently in Agda [21]. By contrast, we did not go after the Nominal recursion principle. Instead, we chose to stay more faithful to the abstract datatype desideratum, generalizing to an arbitrary syntax our own schema for substitution-aware recursion [53] and Michael Norrish's schema for swapping-aware recursion [45]—both of which can be read as stating that terms with bindings are Horn-abstract datatypes, i.e., are initial models of certain Horn theories [53, §3,§8].

A different line of attack on recursion in the literature employs de Bruijn techniques to capture terms and their bijective variable renamings as initial models in presheaf categories [25]. This has been formalized in Isabelle/HOL [4], and recently in Agda [3] and Coq [34] (the last within the Autosubst library).

Generality of the Framework Our constructors are restricted to binding at most one variable in each input—a limitation that makes our framework far from ideal for representing complex binders such as the let patterns of POPLmark's Challenge 2B. In contrast, the specification language Ott [60] and Isabelle's Nominal2 package [67] were specifically designed to address such complex, possibly recursive binders. Incidentally, the Nominal2 package also separates abstractions from terms, like we do, but their abstractions are significantly more expressive; their terms are also quotiented to alpha-equivalence, which is defined via flattening the binders into finite sets or lists of variables (atoms).

On the other hand, to the best of our knowledge, our formalization is the first to capture infinitely branching terms and our foundation of alpha equivalence on the regularity of $|\mathbf{var}|$ is also a theoretical novelty—constituting a less exotic alternative to Murdoch Gabbay's work on infinitely supported objects in nonstandard set theory [26]. This flexibility would be needed to formalize calculi such as infinite-choice process algebra, for which infinitary structures have been previously employed to give semantics [38].

Future Generalizations and Integrations Our theory currently addresses mostly *structural* aspects of terms. A next step would be to cover *behavioral* aspects, such as formats for SOS rules and their interplay with binders, perhaps building on existing Isabelle formalizations of process algebras and programming languages (e.g., [8, 37, 43, 52, 55, 56]).

Another exciting prospect is the integration of our framework with Isabelle's recent package for inductive and coinductive datatypes [13] based on bounded natural functors (BNFs), which follows a compositional design [64] and provides flexible ways to nest types [14] and mix recursion with corecursion [12, 18], but does not yet cover terms with bindings. Achieving compositionality in the presence of bindings will require a substantial refinement of the notion of BNF (since terms with bindings form only partial functors w.r.t. their sets of free variables).

Acknowledgments We thank the anonymous reviewers of the conference version of this paper for their valuable feedback. Popescu has received funding from UK's Engineering and Physical Sciences Research Council (EPSRC) via the grant EP/N019547/1, Verification of Web-based Systems (VOWS).

References

1. The POPLmark challenge (2009). Http://fling-l.seas.upenn.edu/ plclub/cgi-bin/poplmark/
2. IsaFoL (Isabelle Formalization of Logic) project (2018). `https://bitbucket.org/isafol/isafol/wiki/Home`

3. Allais, G., Chapman, J., McBride, C., McKinna, J.: Type-and-scope safe programs and their proofs. In: CPP, pp. 195–207 (2017)
4. Ambler, S.J., Crole, R.L., Momigliano, A.: A definitional approach to primitive recursion over Higher Order Abstract Syntax. In: MERLIN (2003)
5. Aydemir, B.E., Bohannon, A., Weirich, S.: Nominal reasoning techniques in coq: (extended abstract). Electr. Notes Theor. Comput. Sci. **174**(5), 69–77 (2007)
6. Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: POPL 2008, pp. 3–15 (2008)
7. Barendregt, H.P.: The Lambda Calculus. North-Holland (1984)
8. Bengtson, J., Parrow, J., Weber, T.: Psi-calculi in isabelle. J. Autom. Reasoning **56**(1), 1–47 (2016)
9. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL—Lessons learned in formal-logic engineering. In: TPHOLs '99, vol. 1690, pp. 19–36 (1999)
10. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. In: TACAS, pp. 493–507 (2013)
11. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. Logical Methods in Computer Science **12**(4) (2016)
12. Blanchette, J.C., Bouzy, A., Lochbihler, A., Popescu, A., Traytel, D.: Friends with benefits - implementing corecursion in foundational proof assistants. In: ESOP, pp. 111–140 (2017)
13. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co)datatypes for Isabelle/HOL. In: ITP, pp. 93–110 (2014)
14. Blanchette, J.C., Meier, F., Popescu, A., Traytel, D.: Foundational nonuniform (co)datatypes for higher-order logic. In: LICS. IEEE (2017)
15. Blanchette, J.C., Popescu, A.: Mechanizing the metatheory of Sledgehammer. In: FroCoS, pp. 245–260 (2013)
16. Blanchette, J.C., Popescu, A., Traytel, D.: Cardinals in Isabelle/HOL. In: ITP, pp. 111–127 (2014)
17. Blanchette, J.C., Popescu, A., Traytel, D.: Unified classical logic completeness—A coinductive pearl. In: IJCAR 2014, pp. 46–60 (2014)
18. Blanchette, J.C., Popescu, A., Traytel, D.: Foundational extensible corecursion: a proof assistant perspective. In: ICFP, pp. 192–204 (2015)
19. Blanchette, J.C., Popescu, A., Traytel, D.: Soundness and completeness proofs by coinductive methods. J. Autom. Reasoning **58**(1), 149–179 (2017)
20. Chlipala, A.J.: Parametric higher-order abstract syntax for mechanized semantics. In: ICFP, pp. 143–156 (2008)
21. Copello, E., Tasistro, A., Szasz, N., Bove, A., Fernández, M.: Alpha-structural induction and recursion for the lambda calculus in constructive type theory. Electr. Notes Theor. Comput. Sci. **323**, 109–124 (2016). DOI 10.1016/j.entcs.2016.06.008. URL `https://doi.org/10.1016/j.entcs.2016.06.008`
22. Curien, P.L.: Categorical combinators. Information and Control **69**(1-3), 188–254 (1986)
23. Despeyroux, J., Felty, A.P., Hirschowitz, A.: Higher-order abstract syntax in Coq. In: TLCA, pp. 124–138 (1995)
24. Felty, A.P., Momigliano, A.: Hybrid - A definitional two-level approach to reasoning with higher-order abstract syntax. J. Autom. Reasoning **48**(1), 43–105 (2012)
25. Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding (extended abstract). In: LICS 1999, pp. 193–202 (1999)
26. Gabbay, M.J.: A general mathematics of names. Information and Computation **205**(7), 982 – 1011 (2007)
27. Gheri, L., Popescu, A.: The formalization associated to this paper. `http://andreipopescu.uk/papers/Binding_Syntax.zip`
28. Gheri, L., Popescu, A.: A formalized general theory of syntax with bindings. In: ITP (2017)
29. Gordon, A.D., Melham, T.F.: Five axioms of alpha-conversion. In: TPHOLs, pp. 173–190 (1996)
30. Gunter, E.L., Osborn, C.J., Popescu, A.: Theory support for weak Higher Order Abstract Syntax in Isabelle/HOL. In: LFMTP, pp. 12–20 (2009)
31. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. In: LICS 1987, pp. 194–204. IEEE, Computer Society Press (1987)
32. Hennessy, M., Milner, R.: On observing nondeterminism and concurrency. In: ICALP, pp. 299–309 (1980)
33. Hirschowitz, A., Maggesi, M.: Nested abstract syntax in Coq. Journal of Automated Reasoning **49**(3), 409–426 (2012)
34. Kaiser, J., Schäfer, S., Stark, K.: Binder aware recursion over well-scoped de bruijn syntax. In: CPP, pp. 293–306 (2018)
35. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales—a sectioning concept for Isabelle. In: TPHOLs, pp. 149–166 (1999)

36. Keisler, H.J.: Model Theory for Infinitary Logic. North-Holland (1971)
37. Lochbihler, A.: Java and the Java memory model—A unified, machine-checked formalisation. In: H. Seidl (ed.) ESOP 2012, *LNCS*, vol. 7211, pp. 497–517. Springer (2012)
38. Luttik, B.: Choice quantification in process algebra. Ph.D. thesis, University of Amsterdam (2002)
39. Miller, D., Tiu, A.: A proof theory for generic judgments. ACM Transactions on Computational Logic **6**(4), 749–783 (2005)
40. Milner, R.: Communication and concurrency. Prentice Hall (1989)
41. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge (2001)
42. Nipkow, T., Klein, G.: Concrete Semantics: With Isabelle/HOL. Springer (2014)
43. Nipkow, T., von Oheimb, D.: Java$_{light}$ is type-safe - definitely. In: POPL, pp. 161–170 (1998)
44. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer (2002)
45. Norrish, M.: Mechanising lambda-calculus using a classical first order theory of terms with permutations. Higher-Order and Symbolic Computation **19**(2-3), 169–195 (2006)
46. Norrish, M., Vestergaard, R.: Proof pearl: De bruijn terms really do work. In: TPHOLs
47. Pitts, A.M.: Nominal logic: A first order theory of names and binding. In: TACS, pp. 219–242 (2001)
48. Pitts, A.M.: Alpha-structural recursion and induction. J. ACM **53**(3) (2006)
49. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2), 125–159 (1975)
50. Pollack, R., Sato, M., Ricciotti, W.: A canonical locally named representation of binding. J. Autom. Reasoning **49**(2), 185–207 (2012)
51. Popescu, A.: Contributions to the theory of syntax with bindings and to process algebra. PhD thesis, Univ. of Illinois, 2010. Available at `andreipopescu.uk/thesis.pdf`
52. Popescu, A., Gunter, E.L.: Incremental pattern-based coinduction for process algebra and its Isabelle formalization. In: FOSSACS'10 (2010)
53. Popescu, A., Gunter, E.L.: Recursion principles for syntax with bindings and substitution. In: ICFP, pp. 346–358 (2011)
54. Popescu, A., Gunter, E.L., Osborn, C.J.: Strong normalization of System F by HOAS on top of FOAS. In: LICS, pp. 31–40 (2010)
55. Popescu, A., Hölzl, J., Nipkow, T.: Proving concurrent noninterference. In: CPP, pp. 109–125 (2012)
56. Popescu, A., Hölzl, J., Nipkow, T.: Formalizing probabilistic noninterference. In: CPP, pp. 259–275 (2013)
57. Popescu, A., Rosu, G.: Term-generic logic. Theor. Comput. Sci. **577**, 1–24 (2015)
58. Schäfer, S., Tebbi, T., Smolka, G.: Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In: ITP (2015)
59. Schropp, A., Popescu, A.: Nonfree datatypes in isabelle/hol - animating a many-sorted metatheory. In: CPP, pp. 114–130 (2013)
60. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strnisa, R.: Ott: Effective tool support for the working semanticist. J. Funct. Program. **20**(1), 71–122 (2010)
61. Slind, K., Norrish, M.: A brief overview of HOL4. In: TPHOLs, pp. 28–32 (2008)
62. Sun, Y.: An algebraic generalization of Frege structures—binding algebras. Theor. Comput. Sci. **211**(1-2), 189–232 (1999)
63. Takahashi, M.: Parallel reductions in lambda-calculus. Inf. Comput. **118**(1), 120–127 (1995)
64. Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In: LICS 2012, pp. 596–605. IEEE (2012)
65. Urban, C.: Nominal techniques in Isabelle/HOL. J. Autom. Reasoning **40**(4), 327–356 (2008)
66. Urban, C., Berghofer, S., Norrish, M.: Barendregt's variable convention in rule inductions. In: CADE, pp. 35–50 (2007)
67. Urban, C., Kaliszyk, C.: General bindings and alpha-equivalence in Nominal Isabelle. In: ESOP, pp. 480–500 (2011)
68. Urban, C., Tasson, C.: Nominal techniques in Isabelle/HOL. In: CADE, pp. 38–53 (2005)