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 contains 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] and its follow-ups, e.g., [3, 38].

In the past decade, in a framework developed intermittently starting with the second author's PhD [90] and moving into the first author's ongoing PhD, a series of results in logic and λ-calculus have been formalized in Isabelle/HOL [72, 74]. These include classic results (e.g., FOL completeness and soundness of Skolemization [24, 26, 28], strong normalization [93]), as well as novel results in the meta-theory of Isabelle's Sledgehammer tool [18, 24]. A recent paper [47] gives a concrete flavor of how our framework can be deployed and used

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

for two particular syntaxes, those of λ-calculus and of λ-calculus with emphasized values, where we prove the Church-Rosser and standardization theorems.

In this paper, we present the Isabelle/HOL formalization of the framework itself, which is publicly available [48]. 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 [49]. 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).

Relevant literature: The related work Section 8 has been significantly expanded.

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 [12]. 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

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

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 $\mathsf{Lm}\ (\mathsf{Abs}\ x\ X)$.

2.2 Freshness, substitution and swapping

The three 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 $\mathsf{fresh}\ x\ (\mathsf{Lm}\ (\mathsf{Abs}\ x\ (\mathsf{Var}\ x)))$ 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 $\mathsf{Lm}\ (\mathsf{Abs}\ x\ (\mathsf{App}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ y)))$ and $x \notin \{y,z\}$, then:
  - $Y\,[(\mathsf{Var}\ z)/y] = \mathsf{Lm}\ (\mathsf{Abs}\ x\ (\mathsf{App}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ z)))$
  - $Y\,[(\mathsf{Var}\ z)/x] = Y$ (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 $\mathsf{Lm}\ (\mathsf{Abs}\ x\ (\mathsf{App}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ y)))$ and $z \notin \{x,y\}$
  - $Y\,[x \wedge y] = Y\,[y \wedge x] = \mathsf{Lm}\ (\mathsf{Abs}\ y\ (\mathsf{App}\ (\mathsf{Var}\ y)\ (\mathsf{Var}\ x)))$
  - $Y\,[x \wedge z] = Y\,[z \wedge x] = \mathsf{Lm}\ (\mathsf{Abs}\ z\ (\mathsf{App}\ (\mathsf{Var}\ z)\ (\mathsf{Var}\ y)))$
  - $Y\,[y \wedge z] = Y\,[z \wedge y] = \mathsf{Lm}\ (\mathsf{Abs}\ x\ (\mathsf{App}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ z)))$

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 quasiabstraction 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 [16, 22]. 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 compositional) and (2) Barendregt's variable convention [12] (of assuming, w.l.o.g., the bound variables fresh for the parameters) can be invoked in proofs.[1]

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 [85, 92, 112].

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) [52] 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 [71] 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." Note that good terms are only an intermediate concept, not showing in the final product of our theory.

2.5 Possibly infinite branching

Nominal logic's [84, 112] 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 [70] 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 [62] and Hennessey-Milner logic [54]. In our theory, we cover such infinitely branching syntaxes.

[1] On the other hand, some authors have shown that, using a clever bookkeeping of the free and bound variables, several constructions, including parallel substitution, can work smoothly on quasiterms [88, 104].

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).

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**:

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

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*. Note that the same variable can appear in a quasiterm with two different sorts. This is not a problem, since we always treat variables in a sorted manner: we inject them into quasiterms with explicit sort, $\mathsf{qVar}\ xs\ x$, and we bind them with explicit sort, $\mathsf{qAbs}\ xs\ x\ t$. This way, given $xs \neq ys$, we treat "x at sort xs" and "x at sort ys" as unrelated entities.

We use $(\alpha,\beta)\,\textbf{input}$ as a type synonym for $\alpha \to \beta\ \textbf{option}$, 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 $(\textbf{index},\textbf{qterm})\,\textbf{input}$ and (2) a varying number of bound inputs, i.e., families of quasiabstractions modeled as members of $(\textbf{index},\textbf{qabs})\,\textbf{input}$. 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) of which varsorts should be embedded in which term sorts and (2) of which operation symbols are allowed to be applied to which sorts of terms. But, until then, we will develop the interesting part of the theory of bindings without sorting the terms.

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** $\to (\textbf{varsort} \to \textbf{var} \to \textbf{qterm option}) \to$ **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.

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

Fig. 1 Alpha-Equivalence

3.2 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 [85].

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:

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

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

Thus, for good items, we hope to always have a supply of fresh variables. Namely, we hope to prove $\mathsf{qGood}\ X \Longrightarrow \forall xs.\ \exists x.\ \mathsf{qFresh}\ xs\ x\ X$. But goodness is not enough. We also need a special property for the type **var** of variables. In the case of finitary syntax, it suffices to take **var** to be countably infinite, since a finitely branching term will contain fewer than $|\mathbf{var}|$ 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 $|\mathbf{nat}|$ being the smallest regular cardinal. (In addition, we know that for every cardinal k there exists a regular cardinal larger than k.) Thus, the desirable general-

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

Operators		
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**

Fig. 2 Constructors and operators on terms and abstractions

ization of countability is regularity (which is available from Isabelle's cardinal library [25]). 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. $\mathsf{qGood}\ X \Longrightarrow \forall xs.\ |\{x.\ \mathsf{qFresh}\ xs\ x\ X\}| = |\mathbf{var}|$

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 [10, 69, 77].)

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. Figure 2 shows the types of all these term constructors and operators.

To establish an abstraction barrier that sets terms free from their quasiterm origin, we prove that the syntactic constructors mostly behave like free constructors, in that Var, Op and Abs are exhaustive and Var and Op are injective and nonoverlapping. True to the quarantine principle expressed in 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, $\uparrow$good inp, $\uparrow$good $binp$, $|\mathsf{dom}\ inp| < |\mathbf{var}|$ and $|\mathsf{dom}\ binp| < |\mathbf{var}|$. The following hold:
(1) $(ys, y) \neq (xs, x) \Longrightarrow \mathsf{fresh}\ ys\ y\ (\mathsf{Var}\ xs\ x)$
(2) $\uparrow(\mathsf{fresh}\ ys\ y)\ inp \wedge \uparrow(\mathsf{freshAbs}\ ys\ y)\ binp \implies \mathsf{fresh}\ ys\ y\ (\mathsf{Op}\ \delta\ inp\ binp)$
(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, $\uparrow$good inp, $\uparrow$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, $\uparrow$good inp, $\uparrow$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, $\uparrow$good inp, $\uparrow$good $binp$, $\uparrow$good ρ, $|\mathsf{dom}\ inp| < |\mathbf{var}|$, $|\mathsf{dom}\ binp| < |\mathbf{var}|$ and $|\mathsf{dom}\ \rho| < |\mathbf{var}|$. The following hold:

(1) $(\mathsf{Var}\ xs\ x)\,[\rho]\ =\ (\mathsf{if}\ \rho\ xs\ x = \mathsf{Some}\ Y\ \mathsf{then}\ Y\ \mathsf{else}\ \mathsf{Var}\ xs\ x)$
(2) $(\mathsf{Op}\ \delta\ inp\ binp)\,[\rho] = \mathsf{Op}\ \delta\ (\uparrow(\_[\rho])\ inp)\ (\uparrow(\_[\rho])\ binp)$
(3) $\uparrow(\mathsf{fresh}\ xs\ x)\ \rho\ \Longrightarrow\ (\mathsf{Abs}\ xs\ x\ X)\,[\rho]\ =\ \mathsf{Abs}\ xs\ x\ (X\,[\rho])$

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

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

In the unary case we obtain three separate properties:

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 $\rho[y \leftarrow Y[\rho]]_{ys}$ is the assignment ρ updated with value $\mathsf{Some}\,(Y[\rho])$ for (ys, 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 domains. The principles we formalize are generalizations to an arbitrary syntax of results that have been previously described for the particular syntax of λ-calculus [76, 92]. 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). Several iteration/recursion examples for particular syntaxes can be found in [92, §4], [24, §6,§7], [47, §3.2,§3.3,§4] and [90, §3.3,§3.4].

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, **T** and **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:
  - $\mathsf{FRESH} : \mathbf{varsort} \to \mathbf{var} \to \mathbf{T} \to \mathbf{bool}$
  - $\mathsf{FRESHABS} : \mathbf{varsort} \to \mathbf{var} \to \mathbf{A} \to \mathbf{bool}$
  - $\mathsf{SUBST} : \mathbf{T} \to \mathbf{T} \to \mathbf{var} \to \mathbf{varsort} \to \mathbf{T}$
  - $\mathsf{SUBSTABS} : \mathbf{A} \to \mathbf{T} \to \mathbf{var} \to \mathbf{varsort} \to \mathbf{A}$

and it is required to satisfy:

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

Fig. 3 The freshness clauses

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

Fig. 4 The substitution and substitution-renaming clauses

- the freshness clauses F1–F5 shown in Figure 3 (analogous to the implicational simplification rules for freshness in Prop. 7)
- substitution clauses Sb1–Sb4 shown in Figure 4 (analogous to the simplification rules for substitution in Prop. 10)
- the substitution-renaming clause SbRn also shown in Figure 4 (analogous to the substitution-based abstraction equality rule in Prop. 6(2))

Theorem 19. The good terms and abstractions form the *initial FSb model*. Namely, for any FSb model as above, there exist the functions f : **term** $\to$ **T** and $fAbs$: **abs** $\to$ **A** that commute, on good terms, with the constructors and with substitution and preserve freshness:

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

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

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

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

Fig. 5 The swapping and swapping-based congruence clauses

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

Michael Norrish has developed a similar principle that employs swapping instead of substitution [76]. We have also formalized this in our framework—in a slightly restricted form, namely without factoring in fixed variables and parameters.

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

$\mathsf{SWAP} : \mathbf{T} \to \mathbf{var} \to \mathbf{var} \to \mathbf{varsort} \to \mathbf{T}$
$\mathsf{SWAPABS} : \mathbf{A} \to \mathbf{var} \to \mathbf{var} \to \mathbf{varsort} \to \mathbf{A}$

assumed to satisfy the clauses Sw1–Sw3 corresponding to those for simplifying term swapping and, instead of the substitution-renaming property, a swapping-based congruence rule for abstractions SwCong—all shown in Figure 5.

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.

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

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

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

4.2 Primitive recursion

Iteration is a simplified form of primitive recursion. The difference between the two is illustrated by the following simple example:[2] 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 [90, §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:

– OP takes term and abstraction inputs in addition to inputs from the model, i.e., has type $\mathbf{opsym} \to (\mathbf{index},\mathbf{term})\ \mathbf{input} \to (\mathbf{index},\mathbf{T})\ \mathbf{input} \to (\mathbf{bindex},\mathbf{abs})\ \mathbf{input} \to (\mathbf{bindex},\mathbf{A})\ \mathbf{input} \to \mathbf{T}$
– ABS takes an additional term argument, i.e., has type $\mathbf{varsort} \to \mathbf{var} \to \mathbf{term} \to \mathbf{T} \to \mathbf{A}$
– The freshness and substitution operators take additional term and/or abstraction arguments; e.g., the types for the term versions of these are:
  $\mathsf{FRESH} : \mathbf{varsort} \to \mathbf{var} \to \mathbf{term} \to \mathbf{T} \to \mathbf{bool}$
  $\mathsf{SUBST} : \mathbf{term} \to \mathbf{T} \to \mathbf{term} \to \mathbf{T} \to \mathbf{var} \to \mathbf{varsort} \to \mathbf{T}$
– 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 $\mathbf{T}$, whereas X' and Z' are terms.

Theorem 22. For any FSb recursion 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 substitu-

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

tion 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 $\mathsf{skel} : \textbf{term} \to (\textbf{index},\textbf{bindex})\,\textbf{tree}$ and $\mathsf{skelAbs} : \textbf{abs} \to (\textbf{index},\textbf{bindex})\,\textbf{tree}$ 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** → **var** → **tree** → **bool**
  FRESHABS : **varsort** → **var** → **atree** → **bool**
- SWAP : **tree** → **var** → **var** → **varsort** → **tree**
  SWAPABS : **atree** → **var** → **var** → **varsort** → **atree**
  and/or
- SUBST : **tree** → **tree** → **var** → **varsort** → **tree**
  SUBSTABS : **atree** → **tree** → **var** → **varsort** → **atree**

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

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

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

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

Of these, the intended freshness and swapping properties are clearly suitable: The former is vacuously true, and the latter states that the skeleton does not change after swapping two variables. However, the substitution property cannot work, since it states the wrong/undesired property that the skeleton of a term Y does not change after substituting a term X for one of its variables x—which contradicts our intuition that the skeleton may in fact grow (specifically, at the tVar leaves that correspond to free occurrences of $\mathsf{Var}\ xs\ x$ 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, $\mathsf{VAR}\ xs\ x = \mathsf{tVar}$, $\mathsf{OP}\ \delta = \mathsf{tOp}$ and $\mathsf{ABS}\ xs\ x = \mathsf{tAbs}$.) 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):

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

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 Iteration example: 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,

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

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 [85, 110, 112], 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** $\to$ **varsort** $\to$ **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** $\to$ **param** $\to$ **bool** and φAbs : **abs** $\to$ **param** $\to$ **bool**. Assume:
(1) $\forall xs, p.\ |\mathsf{varsOf}\ p\ xs| < |\mathbf{var}|$
(2) $\forall xs, x, p.\ \varphi\ (\mathsf{Var}\ xs\ x)\ p$
(3) $\forall \delta, inp, binp, p.\ |\mathsf{dom}\ inp| < |\mathbf{var}| \wedge |\mathsf{dom}\ binp| < |\mathbf{var}| \wedge \uparrow(\lambda X.\ \mathsf{good}\ X \wedge (\forall q.\ \varphi\ X\ q))$
$inp \wedge \uparrow(\lambda A.\ \mathsf{goodAbs}\ A \wedge (\forall q.\ \varphi Abs\ A\ q))\ binp \implies \varphi\,(\mathsf{Op}\ \delta\ inp\ binp)\ p$
(4) $\forall xs, x, X, p.\ \mathsf{good}\ X \wedge (\forall q.\ \varphi\ X\ q) \wedge\ x \notin \mathsf{varsOf}\ p\ xs \implies \varphi Abs\,(\mathsf{Abs}\ xs\ x\ X)\ p$
Then $\forall X, p.\ \mathsf{good}\ X \implies \varphi\ X\ p$ and $\forall A, p.\ \mathsf{goodAbs}\ A \implies \varphi Abs\ A\ p$.

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 $\mathsf{varsOf}\ ((ys,y),(zs,z),Y)\ xs = \{y \mid xs = ys\} \cup \{z \mid xs = zs\} \cup \{x \mid \neg\,\mathsf{fresh}\ xs\ x\ Y\}$.

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

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 (6.6).

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 described 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 $\mathbf{index} = \mathbf{bindex} = \mathbf{nat}$, $\mathbf{varsort} = \mathbf{sort} = \{\mathsf{lterm}\}$ (a singleton datatype), $\mathbf{opsym} = \{\mathsf{app}, \mathsf{lam}\}$, asSort to be the identity and stOf to be the unique function to $\{\mathsf{lterm}\}$. 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: $\mathsf{arOf}\ \mathsf{app} = (0 \mapsto \mathsf{lterm}, 1 \mapsto \mathsf{lterm})$, $\mathsf{barOf}\ \mathsf{app} = ()$. 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: $\mathsf{arOf}\ \mathsf{lam} = ()$, $\mathsf{barOf}\ \mathsf{lam} = (0 \mapsto (\mathsf{lterm}, \mathsf{lterm}))$.

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 $\mathbf{sort} = \{\mathsf{exp}, \mathsf{proc}\}$. Since we have expression variables but no process variables, we take $\mathbf{varsort} = \{\mathsf{varexp}\}$ and asSort to send varexp to exp. We define **opsym** as the following datatype: $\mathbf{opsym} = \mathsf{Zero} \mid \mathsf{Plus} \mid \mathsf{Inp}\ \mathbf{chan} \mid \mathsf{Out}\ \mathbf{chan} \mid \sum(\mathbf{index\ set})$. The free and bound arities and sorts of the operation symbols are as expected. For example, $\mathsf{Inp}\ c$ acts similarly to λ-abstraction, but binds, in proc terms, variables of a different sort, varexp: $\mathsf{arOf}\ (\mathsf{Inp}\ c) = ()$, $\mathsf{barOf}\ (\mathsf{Inp}\ c) = (0 \mapsto (\mathsf{varexp}, \mathsf{proc}))$.

For $\sum I$ with I : **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 $\mathsf{wls} : \mathbf{sort} \to \mathbf{term} \to \mathbf{bool}$ and $\mathsf{wlsAbs} : \mathbf{varsort} \times \mathbf{sort} \to \mathbf{abs} \to \mathbf{bool}$, 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 $Y = \mathsf{Op}\ \mathsf{Lm}\ ()\ (0 \mapsto A)$. These correspond to what, in the unsorted BNF notation from Section 2.1, we would write $\mathsf{Var}\ x$, $\mathsf{Abs}\ x\ X$ and $\mathsf{Lm}\ (\mathsf{Abs}\ x\ X)$. 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 $\mathsf{Out}\ c$ in Q, we obtain $\mathsf{Op}\ (\mathsf{Out}\ c)\ (0 \mapsto P, 1 \mapsto F)\ ()$, which is not well-sorted: In the inductive clause for wls, the input $(0 \mapsto P, 1 \mapsto F)$ fails to match the arity of $\mathsf{Out}\ c$, $(0 \mapsto \mathsf{exp}, 1 \mapsto \mathsf{proc})$.

6.3 From good to well-sorted

Recall that goodness means "does not branch beyond $|\mathbf{var}|$." 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 $|\mathbf{var}|$. We will more strongly assume that the types of sorts and indexes (the latter subsuming the arity domains) are all smaller than $|\mathbf{var}|$:

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 particular, when we work with well-sorted terms we can take advantage of all the properties we proved about good terms without having to carry with us the cardinality constraint on the size of the inputs. This is because the constraint is implied by the assumption about the cardinality of the operation arities. On the other hand, for well-sorted terms we must of course carry sort and/or arity information.

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. For example, Prop. 18(d) becomes:

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

As a slightly more involved example, here is the well-sorted version of the fresh induction Theorem 24, where the predicates φ and φAbs take sorting information as an additional parameter:

Theorem 27. Let φ : **sort** $\to$ **term** $\to$ **param** $\to$ **bool** and φAbs : **varsort** $\times$ **sort** $\to$ **abs** $\to$ **param** $\to$ **bool**. Assume:
(1) $\forall xs, p.\ |\mathsf{varsOf}\ p\ xs| < |\mathbf{var}|$
(2) $\forall xs, x, p.\ \mathsf{isInBar}\ (xs, s) \implies \varphi\,(\mathsf{asSort}\ xs)\,(\mathsf{Var}\ xs\ x)\ p$
(3) $\forall \delta, inp, binp, p.$
$\uparrow(\lambda s, X.\ \mathsf{wls}\ s\ X \wedge (\forall q.\varphi\ s\ X\ q))\,(\mathsf{arOf}\ \delta)\ inp \wedge$
$\uparrow(\lambda(xs, s), A.\ \mathsf{wlsAbs}\ (xs, s)\ A \wedge (\forall q.\varphi Abs\ (xs, s)\ A\ q))\,(\mathsf{barOf}\ \delta)\ binp \implies$
$\varphi\,(\mathsf{stOf}\ \delta)\,(\mathsf{Op}\ \delta\ inp\ binp)\ p$
(4) $\forall xs, x, s, X, p.\ \mathsf{wls}\ s\ X \wedge (\forall q.\ \varphi\ X\ q) \wedge x \notin \mathsf{varsOf}\ p\ xs \implies \varphi Abs\,(xs, s)\,(\mathsf{Abs}\ xs\ x\ X)\ p$
Then $\forall s, X, p.\ \mathsf{wls}\ s\ X \implies \varphi\ s\ X\ p$ and $\forall s, xs, A, p.\ \mathsf{wlsAbs}\ (xs, s)\ A \implies \varphi Abs\ (xs, s)\ A\ p$.

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.

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 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 recursion model with the following data:

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

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

Theorem 28. For any sorted FSb recursion model, there exist the functions f : **term** $\to$ **T** and $fAbs$: **abs** $\to$ **A** that satisfy the same properties as in Theorem 22 and additionally preserve sorting:

Constructors
Var : **var** → **Lterm**
App : **Lterm** → **Lterm** → **Lterm**
Lm : **Labs** → **Lterm**
Abs : **var** → **Lterm** → **Labs**

Operators		
Freshness	terms	fresh : **var** → **Lterm** → **bool**
	abstractions	freshAbs : **var** → **Labs** → **bool**
Substitution	terms	_[_/_]_ : **Lterm** → **Lterm** → **var** → **Lterm**
	abstractions	_[_/_]_ : **Labs** → **Lterm** → **var** → **Labs**
Parallel Substitution	terms	_[_] : **Lterm** → (**var** → **Lterm option**) → **Lterm**
	abstractions	_[_] : **Labs** → (**var** → **Lterm option**) → **Labs**
Swapping	terms	_[_∧_]_ : **Lterm** → **var** → **var** → **Lterm**
	abstractions	_[_∧_]_ : **Labs** → **var** → **var** → **Labs**

Fig. 6 Constructors and operators on λ-terms and λ-abstractions

– $\mathsf{wls}\ s\ X \implies \mathsf{wls}^{\mathbf{T}}\ s\ (f\ X)$
– $\mathsf{wlsAbs}\ (xs, s)\ A \implies \mathsf{wlsAbs}^{\mathbf{T}}\ (xs, s)\ (fAbs\ A)$

Similarly, we obtain a sorted version of the semantic interpretation theorem. We define a *sorted semantic domain* to have the same components as a semantic domain from Section 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 Instantiating the framework

In Section 6.1 we have seen how our general binding signature can be straightforwardly instantiated to particular syntaxes, such as those of λ-calculus and CCS. However, such raw instances are not convenient for the end user, because they represent too *deep* embeddings. Thus, the constructors would have to be applied indirectly, via the generic operator Op, to which the desired constructor symbol must be passed; sortedness information would have to be carried around; etc.—in other words, one would incur the usual inconvenience arising from instantiating universal algebra to fixed-signature algebras, such as groups or rings.

To address this, we have designed a systematic method for transferring a signature instance to the expected more *shallowly* embedded version of the instance. This involves creating native Isabelle/HOL types of terms for each sort of the signature and transferring all the term constructors and operators and all facts about them to these native types. The process is conceptually straightforward, but is quite tedious, and currently must be done by hand since we have not yet automated it. (But [100, §5] presents the successful automation of a similar kind of transfer.)

For the λ-calculus instance, this results in the countable type **var** and the types **Lterm** and **Labs**, of λ-terms and λ-abstractions, together with the constructors and operators shown in Figure 6. In comparison to the types of the general-theory operators in Figure 2, we see that in Figure 6 the varsort information has disappeared and the generic constructor-managing operator Op has been replaced by the concrete constructors App and Lm. The latter are defined by applying Op to the corresponding constructor symbols and to inputs matching their arities: $\mathsf{App}\ X\ Y$ from $\mathsf{Op}\ \mathsf{app}\ (0 \mapsto X, 1 \mapsto Y)\ ()$ and $\mathsf{Lm}\ A$ from $\mathsf{Op}\ \mathsf{lm}\ ()\ (0 \mapsto A)\ ()$. Because the λ-calculus syntax is single-sorted, the instance has only one freshness,

one (parallel) substitution and one swapping operator for terms; for many-sorted syntaxes, we would have one such operator for each varsort–sort combination.

The theorems of the deep instance of the general theory are (isomorphically) transferred to the shallow embedding's types. For example, here is what Theorem 27 becomes for the λ-calculus syntax, where varsOf : **param** $\to$ **var set**:

Theorem 29. Let φ : **Lterm** $\to$ **param** $\to$ **bool** and φAbs : **Labs** $\to$ **param** $\to$ **bool**. Assume:
(1) $\forall p.$ finite (varsOf p)[3]
(2) $\forall x, p.\ \varphi\ (\mathsf{Var}\ x)\ p$
(3)$_{\mathsf{App}}$ $\forall X, Y, p.\ (\forall q.\ \varphi\ X\ q) \wedge (\forall q.\ \varphi\ Y\ q) \Longrightarrow \varphi\ (\mathsf{App}\ X\ Y)\ p$
(3)$_{\mathsf{Lm}}$ $\forall A, p.\ (\forall q.\ \varphi Abs\ A\ q) \Longrightarrow \varphi\ (\mathsf{Lm}\ A)\ p$
(4) $\forall x, X, p.\ (\forall q.\ \varphi\ X\ q) \wedge x \notin \mathsf{varsOf}\ p \Longrightarrow \varphi Abs\ (\mathsf{Abs}\ x\ X)\ p$
Then $\forall X, p.\ \varphi\ X\ p$ and $\forall A, p.\ \varphi Abs\ A\ p$.

In comparison to the general Theorem 27, again the sorts and varsorts have disappeared and the Op-clause (3) has been replaced by concrete-constructor clauses: one for App and one for Lm. In the recent draft [47] we give more details about two λ-calculus instances of our framework: the above one-sorted syntax, as well as a two-sorted syntax that distinguishes values from arbitrary terms.

6.6 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 [61] 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 have 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. For these syntaxes, we performed several

[3] Note that requiring $|\mathsf{varsOf}\ p| < |\mathbf{var}|$ is the same as requiring that varsOf p be finite.

formal developments that take advantage of the theory's infrastructure. These developments, spanning several years, have used different versions of the general theory, which itself has evolved taking inspiration from them.

The first development took place in 2010, when we formalized a proof of strong normalization for System F [93]. 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 particular, the strong HOAS recursion principle stated in Prop. 4 from [93] was inferred using our semantic interpretation Theorem 23.

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 [86], an adequate HOAS representation of the calculus into itself, a sound interpretation via De Bruijn encodings [34], and the isomorphism between different definitions of λ-terms: ours, the Nominal one [112], the locally named one [88] and the Hybrid one [37]. These results are centered around some translation/representation functions: CPS, HOAS, Church-Rosser complete development [106], De Bruijn, etc.—these functions and their desirable properties (e.g., preservation of substitution, crucial in HOAS [52]) were obtained as instances of our recursion theorems (Section 4).[4]

Finally, in the context of certifying Sledgehammer's HOL to FOL encodings [19], we formalized fundamental results in many-sorted FOL [24], including the completeness, compactness, Skolemization and downward Löwenheim-Skolem theorems.[5] Besides the ubiquitous employment of the properties of freshness and substitution listed in Section 3.4, we used again the semantic interpretation Theorem 23 for the FOL semantics and the recursion Theorem 22 for bootstrapping quickly the (technically quite tricky) Skolemization function.

8 Related Work and Future Work

There is a large amount of literature on formal approaches to syntax with bindings, many of which are supported by proof assistants or logical frameworks. See [1, §2], [37, §6] and [90, §2.10,§3.7] for overviews. Our work follows a *nameful* paradigm to representing binders (8.1) and takes a *universe*-like approach to capture syntax specified by an arbitrary binding signature (8.2), which is restricted to single-variable binders but caters for infinitely branching terms (8.3). We formalize a theory covering not only the syntactic constructors, but also other standard generic operators (8.4), and featuring nominal-logic-style induction and operator-sensitive recursion (8.5).

8.1 Binding representation paradigm

There are three main paradigms of reasoning about bindings: (1) the (first-order) nameful paradigm, (2) the nameless paradigm and (3) higher-order abstract syntax. To illustrate the differences between them, in what follows we slightly depart from this paper's notation and let **term** denote the type of λ-terms—i.e., terms for the particular syntax of λ-calculus.

[4] The formalization work mentioned in this paragraph is mostly unpublished, although aspects concerning the involved recursive definitions are discussed in [90, 92]. Our recent draft [47] gives a detailed account of (an updated version of) the Church-Rosser and Standardization developments.

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

Moreover, we will write Lm for the binding constructor of this syntax, and think of it as operating directly on terms (bypassing any explicit notion of abstraction).

In the *nameful paradigm*, binding variables are passed as arguments to the binding operator, so that Lm has type **var** $\to$ **term** $\to$ **term**; and terms are usually equated modulo alpha-equivalence. The nameful paradigm is followed by most of the informal, pen-and-paper developments of logic, λ-calculi and programming languages, and is systematically pursued by Barendregt in his λ-calculus monograph [12]—where he introduced his famous variable convention. The best known rigorous account of this paradigm is offered by Gabbay and Pitts's nominal logic. Originally developed within a non-standard axiomatization of set theory [43, 44], nominal logic was subsequently cast in a standard foundation [84, 85], and also significantly developed in a proof assistant context—most extensively by Urban and collaborators [108–112]. Our work belongs to the nameful paradigm, giving a formal expression to many ideas from nominal logic—but, as discussed below, departing from nominal logic by exploring notions such as infinitely-branching terms and substitution-aware recursion.

In the *nameless paradigm* originating with De Bruijn [29], the bindings are indicated through nameless pointers to positions in a term. The Lm constructor has type **term** $\to$ **term** or a scope-safe variation of this, such as (**var option**) **term** $\to$ **var term** where **var** is a parameter on which **term** depends (a type variable) and **option** is used to mark the binding positions, or using dependent types, $\mathbf{term}_{n+1} \to \mathbf{term}_n$, to a similar effect. Thus, given a term t, Lm t introduces in t a binding without specifying a variable to be bound, but relying on some pre-identified binding positions in t. Major exponents of the scope-safe nameless paradigm are representations based on presheaves [42, 57] and nested datatypes [7, 17]—interestingly, with seminal papers published in the same year (1999), and two at the same conference (LICS) as the seminal paper on nominal logic [43]. The presheaf approach has been generalized and refined in many subsequent works, e.g., [4–6, 41, 46, 56, 60].

A paradigm quite significantly different from the other two is *higher-order abstract syntax (HOAS)* [31, 35, 37, 39, 52, 78, 80, 82]. Based on ideas going back as far as Church [32], Huet and Lang [58] and Martin-Löf [75, Chapter 3], HOAS has truly gained traction with the works of Harper et. al [52], Pfenning and Elliott [81] and Paulson [78] in the late eighties. HOAS essentially embeds the binders of the represented system (referred to as the *object* system) shallowly into the meta-logic's binder. This shallow embedding leads to higher-order arguments to the binding constructors. There are two main versions of HOAS. The original version, sometimes called *strong HOAS*, endows Lm with the type (**term** $\to$ **term**) $\to$ **term**—where the negative occurrence of **term** makes it impossible to regard Lm as an inductive datatype constructor, as supported by general-purpose proof assistants; consequently, strong HOAS is best pursued in HOAS-dedicated logical frameworks, such as Abella [11], Beluga [83], Delphin [97] and Twelf [82]. On the other hand, the *weak* version of HOAS [35], endowing Lm with the type (**var** $\to$ **term**) $\to$ **term**, is readily compatible with standard logical foundations and has been formalized in general-purpose proof assistants such as Coq [31, 35] and Isabelle [51]. HOAS often allows for lighter formalizations, thanks to borrowing binding mechanisms and sometimes structural properties from the meta-level. Unary substitution is also built in the representation: term-for-variable substitution in strong HOAS and variable-for-variables substitution in weak HOAS. Formalizations in this paradigm are often accompanied by pen-and-paper proofs of the representations' adequacy (which involve informal reasoning about substitution) [52, 79]. In weak HOAS, the adequate representation relies on the exclusion of the so-called *exotic terms*, i.e., terms obtained by applying Lm to non-uniform functions that do not correspond to abstractions—exclusion that can be achieved by either defining a custom predicate [35, 51] or keeping the type of variables generic and relying on parametricity [31].

Some approaches in the literature combine two paradigms. For example, the locally nameless approach [10, 30, 87] employs a nameless representation of bindings, but stores a distinct type of variables that can occur free; this enables some essentially nameful techniques for dealing with free variables (similar to those of nominal logic). Other examples are the Hybrid system [37] and the "HOAS on top of FOAS" approach [93], which develop HOAS reasoning techniques over locally nameless and nameful representation substrata.

Moreover, it should be emphasized that the three paradigms do not differ fundamentally in the employed datatype of terms. Regardless of whether terms are defined by quotienting quasiterms to alpha-equivalence, defined as a free datatype using nameless binders, or encoded using a meta-level binder, they yield essentially the same concept. The isomorphisms between different nameful and nameless representations are well-known, and have also been established formally [77] [8, §3] [90, §2.9.6]. For HOAS developments, the equivalence with an (informal) nameless definition is usually expressed through an adequacy theorem.

The essential differences between the paradigms lie in their respective definitional and reasoning styles, which are customized and optimized for the different types of the binding constructors. Thus (as also illustrated in this paper), defining functions recursively amounts to endowing the target type T with operators matching the arities of the term constructors. For example, in the case of Lm, the types of these operators are (variations of) $\mathbf{var} \to T \to T$, $T \to T$, $\prod_n (T_{n+1} \to T_n)$ and $(\mathbf{var} \to T) \to T$ for the nameful, (classic) nameless, well-scoped nameless and weak HOAS representations, respectively—these entail different styles of recursive definitions (whose correctness may or may not require discharging proof obligations) and inductive proofs (which may or may not require auxiliary lemmas). A deep analysis of these tradeoffs is beyond the scope of this paper, but Section 8.5 offers some high-level points of comparison.

8.2 Approach to handling arbitrary signatures

There are many formal developments in proof assistants that focus on particular syntaxes, e.g., that of a chosen logic or programming language. In this subsection and the next one we only discuss developments that, like ours, cover an entire class of syntaxes.

An expressive type theory such as Agda's or Coq's easily caters for generic developments using *universes*, where types can be referred to and manipulated via *codes*. Most of the universe constructions for syntax with bindings follow the (locally) nameless paradigm, e.g., [4, 63, 65, 66]. However, universes for the other two paradigms have also been explored: [63] also features and alternative (parametric) weak HOAS representation, and [33] features a nameful (nominal) representation.

An alternative to generic development via universes is the *code generator* approach,[6] which is preferred in proof assistants based on weaker logics, such as HOL. Code generators, also known as *definitional packages* in the HOL literature, produce the datatypes and associated theorems dynamically (using tactics), for each user-specified syntax. This is the approach taken for Nominal Isabelle, but also for several tools for Coq, such as LNgen [8] (supporting a locally nameless representation), DBGen [89] (supporting the classic nameless representation of De Bruijn), and Autosubst [99, 103] and Needle&Knot [64] (supporting scope-safe nameless representations).

Our work targets a weak logic, but is close to the universe approach—except that instead of type codes we use deeply embedded sorts, and develop our whole theory, including universal-algebra-style recursion principles, in a deep embedding.

[6] Here, by "code generator" we refer to a tool for producing code (definitions, theorems and proofs) in a proof assistant, not in a programming language.

From a theoretical perspective, the universe approach has a wider appeal, as it models "statically" the meta-theory in its entire generality. However, a code generator is often more practical, since most proof assistant users only care about the particular instance syntax used in their development—and code generators can deliver fine-tuned solutions here. By contrast, in our case, as we discuss in Section 6.5, simply instantiating the signature with a particular syntax is not entirely satisfactory, for which reason we need to engage in a process of transferring theorems to a more shallow representation. In the future, we plan to have this transfer fully automated, obtaining the best of both worlds, namely a universal algebra theory that caters for a statically certified code generator.

Note that much of our instantiation overhead comes from the lack of expressiveness of HOL. Indeed, had the logic allowed us to speak of families of types indexed by sorts, the "raw" instances would have already been quite shallow.[7] In addition, dependent type families would have made our generic development conceptually more direct, avoiding the application of operation symbols outside their arities (which currently require correction via sortedness predicates); however, we believe this would not have represented a decisive boost in automation, since the proof obligations arising from dependent types would have had a similar complexity to those incurred by the sortedness predicates.

8.3 Generality of the represented syntax

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 in part 2B of the POPLmark challenge. By contrast, the specification language Ott [102], Isabelle's Nominal2 definitional package [111], Coq's Needle&Knot tool and our own recent "bindings as functors" representation [21] were specifically designed to address such complex, possibly recursive binders. Incidentally, Nominal2 separates abstractions from terms, like we do (but their abstractions are significantly more expressive).

On the other hand, our formalization seems to be the first to cover 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 Gabbay's work on infinitely supported objects in nonstandard set theory [45]. This flexibility is needed when formalizing systems such as infinitary logics [62] and λ-calculi [59], as well as infinite-choice process algebra (for which infinitary structures have been previously employed to give semantics [68]).

8.4 Support for generic operators

The main goal of our work was the development of *as much as possible from the theory of syntax, for an arbitrary syntax*. In particular, we wanted to define and study important binding-datatype-generic operators, in the spirit of polytipic programming [55] and universal-algebraic theories of syntax [42, 96, 105].

This development, performed in Isabelle, joins a series of generic formalizations and tools in Coq that offer support, i.e., definitions and theorems, for unary substitution (e.g., LNgen, GMeta, DBGen and Needle&Knot) and parallel substitution (Autosubst). In addition, Needle&Knot and GMeta go further and handle contexts and their associated term well-scoping predicates and variable lookup operators.

Our work seems to be the first to formalize generic support for the interpretation of terms in semantic domains—which in the meantime has also been developed in Agda within the well-scoped nameless paradigm, using a universe [4]. While widely applicable, our seman-

[7] However, any generic development, even in dependent type theory, seems to require some code generation in order to offer truly usable instances—as explained, e.g., by the authors of GMeta [65, §3.1].

tic interpretation principle incurs the usual limitations of (and can benefit from the usual workarounds for) HOL. For example, if we want to assign semantics to the simply-typed λ-calculus, we would first need to have all the simple types embedded in a larger type (which is possible by either restricting the full function space or employing a gentle extension of HOL [53]). Then we could invoke our principle to obtain an untyped interpretation, which in a second stage could be proved type-correct. This is clearly a more bureaucratic solution than what is offered by the state of the art in Agda [4].

8.5 Definition and reasoning principles

Within the nameful paradigm, a major strand in the literature is connected to the quest, pioneered by Gordon and Melham [50], for understanding terms with bindings modulo alpha as an abstract datatype. The state of the art is currently represented by the expressive induction and recursion principles of nominal logic [9, 33, 85, 108, 109] and by the essential variation of nominal recursion due to Norrish [76].

In this paper we formalized the nominal logic structural induction principle from [108], which was implemented in Nominal Isabelle (for a different notion of binding signature). By contrast, we did not formalize the nominal logic *recursion* principle. Instead, we chose to stay more faithful to a classical abstract datatype desideratum, generalizing to an arbitrary syntax our own schema for substitution-aware recursion [92] and a parameter-free version of Norrish's schema for swapping-aware recursion [76]—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 [92, §3,§8]. The difference between our recursion principles and the nominal logic ones rests in the identification of terms as forming initial objects in different categories: that of models of a Horn theory, versus that of some enriched nominal sets. Our recursion principles are better suited to handle situations where the finite support assumption no longer holds, as in one of our main applications: interpretation in semantic domains.[8] On the other hand, unlike our recursion principles, the nominal principle (as well as Norrish's) observes Barendregt's convention in the recursion clauses, allowing the presence of additional parameters for which the binding variables are fresh. Finally, the nominal recursion principle is currently better supported in Isabelle by the automation delivered through the Nominal definitional package.

A different line of attack to recursion is taken within the nameless paradigm. The nameless recursion principles are more readily available thanks to the freeness of the term datatype (no quotienting involved). On the other hand, the nameful recursion principles stay closer to informal practice, but this happens at the price of the user (or ideally, the proof assistant) having to prove some conditions that ensure the definition is correct—in our case, these are the FSb/FSw/FSbSw (recursion) model conditions. We refer to [15] for a detailed analysis of the relative merits of the nameful and nameless paradigms (in two particular variants). More insight into possibilities for automating or reducing the proof obligations resulted from quotienting could be offered by general-purpose datatype quotienting mechanisms such as Isabelle's nonfree datatypes [100] or the higher inductive types of homotopy type theory implemented in Coq [13] and Lean [36].

In the nameless presheaf approaches, the recursion principles come from exhibiting the terms as initial models in specific presheaf toposes. To apply these principles, a substitution/renaming structure needs to be provided on the intended target domain, and some conditions need to be verified, with the reward that the defined function is guaranteed to preserve the additional structure—which leads to a situation somewhat similar to the nameful case,

[8] The difficulties of achieving this with nominal logic recursion are analyzed in [85, §6.3].

although admittedly the conditions to be verified are quite different. Recent formalization-supported investigations by Allais et al. [4, 5] and Kaiser et al. [60] have revealed lighter variations of the presheaf-based recursors (incurring fewer proof obligations for the user).

A quite different take on binding-aware recursion is pursued in HOAS-dedicated frameworks [40, 101], whose relationship with the other two paradigms is difficult to grasp due to the use of a substantially different logical foundation. A study of this relationship would be interesting future work. On the other hand, the weak HOAS induction and recursion principles are easier to compare since they are developed within standard foundations. Like the De Bruijn ones, they employ a free datatype of terms—namely, considering Lm as an (infinitary) injective constructor of type $(\mathbf{var} \to \mathbf{term}) \to \mathbf{term}$. In fact, by virtue of working in a suitable presheaf topos where context extension is isomorphic to the function space from the presheaf of variables, some nameless presheaf representations yield a weak-HOAS-like recursor [42, 57] (but again, in a topos different from $\mathcal{S}et$). As another point of convergence, the recursor developed by Gordon and Melham [50] for alpha-quotiented terms (within the nameful paradigm) is essentially the weak HOAS one [35].

8.6 Ongoing and future work

Our theory currently addresses mostly *structural* aspects of terms. A next step would be to cover *behavioral* aspects, such as binding-aware proof rules associated to SOS formats, perhaps building on existing Isabelle formalizations of process algebras and programming languages (e.g., [14, 67, 73, 91, 94, 95]).

A major test for our theory will be its application to a realistic programming language, featuring several syntactic categories with many constructors and a large operational semantics. We have no reason to doubt that our theory would scale for such a development, but we cannot know for a fact before we try. One aspect that could prove helpful when dealing with a large syntax may be refraining from switching to a shallow representation (as described in Section 6.5), but staying within a deep representation, i.e., having a single type of terms classified by a sortedness predicate. For example, it may be more proof-engineering efficient (incurring a smaller amount of proof scripts) to have a single generic substitution operator as in the deep representation rather than a combinatorial explosion of such operators for each combination of variable type and term type. (See [98, §7] for the discussion of a similar phenomenon for a medium-sized syntax.)

Another line of work (which we pursue together with other colleagues) is the development of support for bindings that gets full traction from Isabelle's existing infrastructure—namely its definitional package for inductive and coinductive datatypes [22] based on bounded natural functors (BNFs), which follows a compositional design [107] and provides flexible ways to nest types [23] and mix recursion with corecursion [20, 27]. Theoretical progress with our ongoing effort on this front is reported in the recent paper [21], where BNFs are refined into what we call map-restricted BNFs (MRBNFs), a notion of functor that accommodates bindings as first-class citizens. In addition to inheriting the modularity of the BNF framework, this work covers complex bindings à la POPLmark and beyond. Moreover, it further stretches our regular-cardinal technique to allow non-well-founded, coinductive terms (such as Böhm trees [12]) in addition to infinitely branching terms. This development is only in a prototype phase, with a formalization having been developed for a fixed signature (modeled as a fixed MRBNF). It does not follow the current paper's universe-like approach, but will be implemented as a definitional package.

References

1. The POPLmark challenge (2009). `https://www.seas.upenn.edu/~plclub/poplmark/`

2. IsaFoL (Isabelle Formalization of Logic) project (2018). https://bitbucket.org/isafol/isafol/wiki/Home
3. Abel, A., Momigliano, A., Pientka, B.: POPLMark Reloaded. In: LFMTP (2017)
4. Allais, G., Atkey, R., Chapman, J., McBride, C., McKinna, J.: A type and scope safe universe of syntaxes with binding: their semantics and proofs. PACMPL **2**(ICFP), 90:1–90:30 (2018)
5. Allais, G., Chapman, J., McBride, C., McKinna, J.: Type-and-scope safe programs and their proofs. In: CPP, pp. 195–207 (2017)
6. Altenkirch, T., Ghani, N., Hancock, P., McBride, C., Morris, P.: Indexed containers. J. Funct. Program. **25** (2015)
7. Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: CSL, pp. 453–468 (1999)
8. Aydemir, B., Weirich, S.: LNgen: Tool support for locally nameless representations. Tech. rep., UPenn (2010)
9. Aydemir, B.E., Bohannon, A., Weirich, S.: Nominal reasoning techniques in Coq (extended abstract). Electr. Notes Theor. Comput. Sci. **174**(5), 69–77 (2007)
10. Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: POPL 2008, pp. 3–15 (2008)
11. Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A., Wang, Y.: Abella: A system for reasoning about relational specifications. J. Formalized Reasoning **7**(2), 1–89 (2014)
12. Barendregt, H.P.: The Lambda Calculus. North-Holland (1984)
13. Bauer, A., Gross, J., Lumsdaine, P.L., Shulman, M., Sozeau, M., Spitters, B.: The HoTT library: a formalization of homotopy type theory in Coq. In: CPP, pp. 164–172 (2017)
14. Bengtson, J., Parrow, J., Weber, T.: Psi-calculi in Isabelle. J. Autom. Reasoning **56**(1), 1–47 (2016)
15. Berghofer, S., Urban, C.: A head-to-head comparison of De Bruijn indices and names. Electr. Notes Theor. Comput. Sci. **174**(5), 53–67 (2007)
16. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL—Lessons learned in formal-logic engineering. In: TPHOLs, pp. 19–36 (1999)
17. Bird, R.S., Paterson, R.: De Bruijn notation as a nested datatype. J. Funct. Program. **9**(1)
18. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. In: TACAS, pp. 493–507 (2013)
19. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. Logical Methods in Computer Science **12**(4) (2016)
20. 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)
21. Blanchette, J.C., Gheri, L., Popescu, A., Traytel, D.: Bindings as bounded natural functors. PACMPL **3**(POPL), 22:1–22:34 (2019)
22. 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)
23. Blanchette, J.C., Meier, F., Popescu, A., Traytel, D.: Foundational nonuniform (co)datatypes for higher-order logic. In: LICS, pp. 1–12 (2017)
24. Blanchette, J.C., Popescu, A.: Mechanizing the metatheory of Sledgehammer. In: FroCoS, pp. 245–260 (2013)
25. Blanchette, J.C., Popescu, A., Traytel, D.: Cardinals in Isabelle/HOL. In: ITP, pp. 111–127 (2014)
26. Blanchette, J.C., Popescu, A., Traytel, D.: Unified classical logic completeness—A coinductive pearl. In: IJCAR 2014, pp. 46–60 (2014)
27. Blanchette, J.C., Popescu, A., Traytel, D.: Foundational extensible corecursion: a proof assistant perspective. In: ICFP, pp. 192–204 (2015)
28. Blanchette, J.C., Popescu, A., Traytel, D.: Soundness and completeness proofs by coinductive methods. J. Autom. Reasoning **58**(1), 149–179 (2017)
29. de Bruijn, N.: λ-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indag. Math **34**(5), 381–392 (1972)
30. Charguéraud, A.: The locally nameless representation. J. Autom. Reasoning **49**(3), 363–408 (2012)
31. Chlipala, A.J.: Parametric higher-order abstract syntax for mechanized semantics. In: ICFP, pp. 143–156 (2008)
32. Church, A.: A formulation of the simple theory of types. J. Symb. Logic **5**(2), 56–68 (1940)
33. Copello, E., Szasz, N., Tasistro, Á.: Formalisation in constructive type theory of Barendregt's variable convention for generic structures with binders. In: LFMTP, pp. 11–26 (2018)
34. Curien, P.L.: Categorical combinators. Information and Control **69**(1-3), 188–254 (1986)
35. Despeyroux, J., Felty, A.P., Hirschowitz, A.: Higher-order abstract syntax in Coq. In: TLCA, pp. 124–138 (1995)

36. van Doorn, F.: On the formalization of higher inductive types and synthetic homotopy theory. Ph.D. thesis, Carnegie Mellon University (2018)
37. 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)
38. Felty, A.P., Momigliano, A., Pientka, B.: An open challenge problem repository for systems supporting binders. In: LFMTP, pp. 18–32 (2015)
39. Felty, A.P., Pientka, B.: Reasoning with higher-order abstract syntax and contexts: A comparison. In: ITP, pp. 227–242 (2010)
40. Ferreira, F., Pientka, B.: Programs using syntax with first-class binders. In: ESOP, pp. 504–529 (2017)
41. Fiore, M., Gambino, N., Hyland, M., Winskel, G.: The cartesian closed bicategory of generalised species of structures. J. London Math. Soc. (1), 203–220 (2008)
42. Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding (extended abstract). In: LICS, pp. 193–202 (1999)
43. Gabbay, M., Pitts, A.M.: A new approach to abstract syntax involving binders. In: LICS, pp. 214–224 (1999)
44. Gabbay, M., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Asp. Comput. **13**(3-5), 341–363 (2002)
45. Gabbay, M.J.: A general mathematics of names. Information and Computation **205**(7), 982 – 1011 (2007)
46. Gambino, N., Hyland, M.: Wellfounded trees and dependent polynomial functors. In: TYPES, pp. 210–225 (2003)
47. Gheri, L., Popescu, A.: A case study in reasoning about syntax with bindings: The Church-Rosser and standardization theorems. Submitted to the Journal of Automated Reasoning. Available at `http://andreipopescu.uk/papers/cbncbv.pdf`
48. Gheri, L., Popescu, A.: The formalization associated to this paper. `http://andreipopescu.uk/papers/Binding_Syntax.zip`
49. Gheri, L., Popescu, A.: A formalized general theory of syntax with bindings. In: ITP (2017)
50. Gordon, A.D., Melham, T.F.: Five axioms of alpha-conversion. In: TPHOLs, pp. 173–190 (1996)
51. 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)
52. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. In: LICS, pp. 194–204 (1987)
53. Harrison, J.: Towards self-verification of HOL Light. In: IJCAR, pp. 177–191 (2006)
54. Hennessy, M., Milner, R.: On observing nondeterminism and concurrency. In: ICALP, pp. 299–309 (1980)
55. Hinze, R.: Polytypic programming with ease. Journal of Functional and Logic Programming **2001**(3) (2001)
56. Hirschowitz, A., Maggesi, M.: Modules over monads and initial semantics. Inf. Comput. **208**(5), 545–564 (2010)
57. Hofmann, M.: Semantical analysis of higher-order abstract syntax. In: LICS, p. 204 (1999)
58. Huet, G.P., Lang, B.: Proving and applying program transformations expressed with second-order patterns. Acta Inf. **11**, 31–55 (1978). URL `https://doi.org/10.1007/BF00264598`
59. Joachimski, F.: Reduction properties of ΠIE-systems. Ph.D. thesis, LMU München (2001)
60. Kaiser, J., Schäfer, S., Stark, K.: Binder aware recursion over well-scoped De Bruijn syntax. In: CPP, pp. 293–306 (2018)
61. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales—a sectioning concept for Isabelle. In: TPHOLs, pp. 149–166 (1999)
62. Keisler, H.J.: Model Theory for Infinitary Logic. North-Holland (1971)
63. Keuchel, S., Jeuring, J.: Generic conversions of abstract syntax representations. In: Workshop on Generic Programming, pp. 57–68 (2012)
64. Keuchel, S., Weirich, S., Schrijvers, T.: Needle & Knot: Binder boilerplate tied up. In: ESOP, pp. 419–445 (2016)
65. Lee, G., d. S. Oliveira, B.C., Cho, S., Yi, K.: GMeta: A generic formal metatheory framework for first-order representations. In: ESOP, pp. 436–455 (2012)
66. Licata, D.R., Harper, R.: A universe of binding and computation. In: ICFP '09, pp. 123–134 (2009)
67. 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)
68. Luttik, B.: Choice quantification in process algebra. Ph.D. thesis, University of Amsterdam (2002)
69. Miller, D., Tiu, A.: A proof theory for generic judgments. ACM Transactions on Computational Logic **6**(4), 749–783 (2005)
70. Milner, R.: Communication and concurrency. Prentice Hall (1989)
71. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge (2001)

72. Nipkow, T., Klein, G.: Concrete Semantics: With Isabelle/HOL. Springer (2014)
73. Nipkow, T., von Oheimb, D.: Java$_{light}$ is type-safe - definitely. In: POPL, pp. 161–170 (1998)
74. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer (2002)
75. Nordström, B., Petersson, K., Smith, J.M.: Programming in Martin-Löf's Type Theory: An Introduction. Oxford University Press (1990)
76. 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)
77. Norrish, M., Vestergaard, R.: Proof pearl: De Bruijn terms really do work. In: TPHOLs, pp. 207–222 (2007)
78. Paulson, L.C.: The foundation of a generic theorem prover. J. Autom. Reason. **5**(3) (1989)
79. Pfenning, F.: Computation and Deduction (2001)
80. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: PLDI, pp. 199–208 (1988)
81. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: PLDI, pp. 199–208 (1988)
82. Pfenning, F., Schürmann, C.: System description: Twelf - A meta-logical framework for deductive systems. In: CADE, pp. 202–206 (1999)
83. Pientka, B.: Beluga: Programming with dependent types, contextual data, and contexts. In: FLOPS, pp. 1–12 (2010)
84. Pitts, A.M.: Nominal logic: A first order theory of names and binding. In: TACS, pp. 219–242 (2001)
85. Pitts, A.M.: Alpha-structural recursion and induction. J. ACM **53**(3) (2006)
86. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2), 125–159 (1975)
87. Pollack, R.: Closure under alpha-conversion. In: TYPES, pp. 313–332 (1993)
88. Pollack, R., Sato, M., Ricciotti, W.: A canonical locally named representation of binding. J. Autom. Reasoning **49**(2), 185–207 (2012)
89. Polonowski, E.: Automatically generated infrastructure for de Bruijn syntaxes. In: ITP, pp. 402–417 (2013)
90. Popescu, A.: Contributions to the theory of syntax with bindings and to process algebra (2010). PhD thesis, Univ. of Illinois. Available at `andreipopescu.uk/thesis.pdf`
91. Popescu, A., Gunter, E.L.: Incremental pattern-based coinduction for process algebra and its Isabelle formalization. In: FoSSaCS (2010)
92. Popescu, A., Gunter, E.L.: Recursion principles for syntax with bindings and substitution. In: ICFP, pp. 346–358 (2011)
93. 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)
94. Popescu, A., Hölzl, J., Nipkow, T.: Proving concurrent noninterference. In: CPP, pp. 109–125 (2012)
95. Popescu, A., Hölzl, J., Nipkow, T.: Formalizing probabilistic noninterference. In: CPP, pp. 259–275 (2013)
96. Popescu, A., Rosu, G.: Term-generic logic. Theor. Comput. Sci. **577**, 1–24 (2015)
97. Poswolsky, A., Schürmann, C.: System description: Delphin—A functional programming language for deductive systems. Electr. Notes Theor. Comput. Sci. **228**, 113–120 (2009)
98. Rossberg, A., Russo, C.V., Dreyer, D.: F-ing modules. In: TLDI, pp. 89–102 (2010)
99. Schäfer, S., Tebbi, T., Smolka, G.: Autosubst: Reasoning with De Bruijn terms and parallel substitutions. In: ITP (2015)
100. Schropp, A., Popescu, A.: Nonfree datatypes in Isabelle/HOL – animating a many-sorted metatheory. In: CPP, pp. 114–130 (2013)
101. Schurmann, C., Despeyroux, J., Pfenning, F.: Primitive recursion for higher-order abstract syntax. Theor. Comput. Sci. **266**(1-2), 1–57 (2001)
102. 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)
103. Stark, K., Schäfer, S., Kaiser, J.: Autosubst 2: Reasoning with multi-sorted De Bruijn terms and vector substitutions. In: CPP (2019). To appear.
104. Stoughton, A.: Substitution revisited. Theor. Comput. Sci. **59**, 317–325 (1988)
105. Sun, Y.: An algebraic generalization of frege structures-binding algebras. Theor. Comput. Sci. **211**(1-2), 189–232 (1999)
106. Takahashi, M.: Parallel reductions in lambda-calculus. Inf. Comput. **118**(1), 120–127 (1995)
107. Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In: LICS, pp. 596–605 (2012)
108. Urban, C.: Nominal techniques in Isabelle/HOL. J. Autom. Reason. **40**(4), 327–356 (2008)
109. Urban, C., Berghofer, S.: A recursion combinator for nominal datatypes implemented in Isabelle/HOL. In: IJCAR, pp. 498–512 (2006)

110. Urban, C., Berghofer, S., Norrish, M.: Barendregt's variable convention in rule inductions. In: CADE, pp. 35–50 (2007)
111. Urban, C., Kaliszyk, C.: General bindings and alpha-equivalence in Nominal Isabelle. In: ESOP, pp. 480–500 (2011)
112. Urban, C., Tasson, C.: Nominal techniques in Isabelle/HOL. In: CADE, pp. 38–53 (2005)