A Formalized General Theory of Syntax with Bindings

Lorenzo Gheri[1] and Andrei Popescu[1,2]

[1] Department of Computer Science, Middlesex University London, UK
[2] Institute of Mathematics Simion Stoilow of the Romanian Academy, Bucharest, Romania

Abstract. We present the formalization of a theory of syntax with bindings that has been developed and refined over the last decade to support several large formalization efforts. Terms are defined for an arbitrary number of constructors of varying numbers of inputs, quotiented to alpha-equivalence and sorted according to a binding signature. The theory includes a rich collection of properties of the standard operators on terms, such as substitution and freshness. It also includes induction and recursion principles and support for semantic interpretation, all tailored for smooth interaction with the bindings and the standard operators.

1 Introduction

Syntax with bindings is an essential ingredient in the formal specification and implementation of logics and programming languages. However, correctly and formally specifying, assigning semantics to, and reasoning about bindings is notoriously difficult and error-prone. This fact is widely recognized in the formal verification community and is reflected in manifestos and benchmarks such as the influential POPLmark challenge [1].

In the past decade, in a framework developed intermittently starting with the second author's PhD [42] and moving into the first author's ongoing PhD, a series of results in logic and λ-calculus have been formalized in Isabelle/HOL [35, 37]. These include classic results (e.g., FOL completeness and soundness of Skolemization [7, 13, 15], λ-calculus standardization and Church-Rosser theorems [42, 44], System F strong normalization [45]), as well as the meta-theory of Isabelle's Sledgehammer tool [7, 8].

In this paper, we present the Isabelle/HOL formalization of the framework itself (made available from the paper's website [21]). 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 and recursion schemes (Section 4), including support for the semantic interpretation of syntax, and the sorting of terms according to a signature (Section 5). Within the large body of formalizations in the area (Section 6), distinguishing features of our work are the general setting (many-sorted signature, possibly infinitary syntax), a rich theory of the standard operators, and operator-aware recursion. More details on this paper's results can be found in an extended technical report [22].

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 [4]. We assume an infinite supply of variables, $x \in \mathbf{var}$. The λ-terms, $X, Y \in \mathbf{term}_\lambda$, are defined by the following BNF grammar:

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

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

2.1 Standalone Abstractions

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

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

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

2.2 Freshness and Substitution

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

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

And there are corresponding operators for abstractions—e.g., $\mathsf{freshAbs}\ x\ (\mathsf{Abs}\ x\ (\mathsf{Var}\ x))$ holds. Freshness and substitution 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 [6, 10]. 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 [4] (of assuming, w.l.o.g., the bound variables fresh for the parameters) can be invoked in proofs.

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

A framework that deals gracefully with these obligations can make an important difference in applications—enabling the formalizer to quickly leave behind low-level "bootstrapping" issues and move to the interesting core of the results. To address these obligations, we formalize state-of-the-art techniques from the literature [41, 44, 57].

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) [25] 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 [34] we have channel names but no process variables, etc.

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

2.5 Possibly Infinite Branching

Nominal Logic's [40, 57] 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 [33] 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 [29] and Hennessey-Milner logic [26]. In our theory, we cover such infinitely branching syntaxes.

3 General Terms with Bindings

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

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**:

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

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*.

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 $\delta : \textbf{opsym}$ 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 5, when we introduce signatures. Then, we will gain control (1) on which varsorts should be embedded in which term sorts and (2) on which operation symbols are allowed to be applied to which sorts of terms. But, until then, we will develop the interesting part of the theory of bindings without sorting the terms.

On quasiterms, we define freshness, $\mathsf{qFresh} : \textbf{varsort} \to \textbf{var} \to \textbf{qterm} \to \textbf{bool}$, substitution, $_[_/_]_ : \textbf{qterm} \to \textbf{qterm} \to \textbf{var} \to \textbf{varsort} \to \textbf{qterm}$, parallel substitution, $_[_] : \textbf{qterm} \to (\textbf{varsort} \to \textbf{var} \to \textbf{qterm option}) \to \textbf{qterm}$, swapping, $_[_\wedge_]_ : \textbf{qterm}$

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

Fig. 1. Alpha-Equivalence

$\rightarrow$ **var** $\rightarrow$ **var** $\rightarrow$ **varsort** $\rightarrow$ **qterm**, and alpha-equivalence, $\mathsf{alpha} : \mathbf{qterm} \rightarrow \mathbf{qterm} \rightarrow \mathbf{bool}$—and corresponding operators on quasiabstractions: qFreshAbs, alphaAbs, etc.

The definitions proceed as expected, with picking suitable fresh variables in the case of substitutions and alpha. For parallel substitution, given a (partial) variable-to-quasiterm assignment $\rho : \mathbf{varsort} \rightarrow \mathbf{var} \rightarrow \mathbf{qterm\ option}$, the quasiterm $X[\rho]$ is obtained by substituting, for each free variable x of sort xs in X for which ρ is defined, the quasiterm Y where $\rho\ xs\ x = \mathsf{Some}\ Y$. We only show the formal definition of alpha.

3.2 Alpha-Equivalence

We define the predicates alpha (on quasiterms) and alphaAbs (on quasiabstractions) mutually recursively, as shown in Fig. 1. For variable quasiterms, we require equality on both the variables and their sorts. For qOp quasiterms, we recurse through the components, *inp* and *binp*. Given any predicate $P : \beta^2 \rightarrow \mathbf{bool}$, we write $\uparrow P$ for its lifting to $(\alpha,\beta)\mathbf{input}^2 \rightarrow \mathbf{bool}$, defined as $\uparrow P\ inp\ inp' \Longleftrightarrow \forall i.$ 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, 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 [41].

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:

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

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. Thus, the desirable generalization of countability is regularity (which is available from Isabelle's cardinal library [12]). 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 [3, 32, 39].)

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: Var : **varsort** $\to$ **var** $\to$ **term**, Op : **opsym** $\to$ (**index**, **term**) **input** $\to$ (**bindex**, **abs**) **input** $\to$ **term**, Abs : **varsort** $\to$ **var** $\to$ **term** $\to$ **abs**, fresh : **varsort** $\to$ **term** $\to$ **bool**, _[_/_]_ : **term** $\to$ **term** $\to$ **var** $\to$ **varsort** $\to$ **term**, etc.

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 $\mathsf{good}\ X$ and $\mathsf{good}\ X'$. 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 $\mathsf{good}\ X$ and $\mathsf{good}\ X'$. 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. For example, the following facts resemble an inductive definition of freshness (as a predicate):

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

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

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

Prop 8. Prop. 7 stays true if the implications are replaced by equivalences ($\Longleftrightarrow$).

For substitution, we prove facts with a similarly primitive recursion flavor:

Prop 9. Assume $\mathsf{good}\ X$, $\mathsf{good}\ Y$, $\uparrow\mathsf{good}\ inp$, $\uparrow\mathsf{good}\ binp$, $|\mathsf{dom}\ inp| < |\mathbf{var}|$ and $|\mathsf{dom}\ binp| < |\mathbf{var}|$. The following hold:
(1) $(\mathsf{Var}\ xs\ x)\,[Y/y]_{ys} = (\text{if } (xs, x) = (ys, y) \text{ then } Y \text{ else } \mathsf{Var}\ xs\ x)$
(2) $(\mathsf{Op}\ \delta\ inp\ binp)\,[Y/y]_{ys} = \mathsf{Op}\ \delta\ (\uparrow(_[Y/y]_{ys})\ inp)\ (\uparrow(_[Y/y]_{ys})\ binp)$
(3) $(xs, x) \neq (ys, y) \wedge \mathsf{fresh}\ xs\ x\ Y \implies (\mathsf{Abs}\ xs\ x\ X)\,[Y/y]_{ys} = \mathsf{Abs}\ xs\ x\ (X\,[Y/y]_{ys})$

We also prove generalizations of Prop. 9's facts for parallel substitution, for example, $\uparrow(\mathsf{fresh}\ xs\ x)\ \rho \Longrightarrow (\mathsf{Abs}\ xs\ x\ X)\,[\rho] = \mathsf{Abs}\ xs\ x\ (X\,[\rho])$.

Note that, for properties involving Abs, the simplification rules require freshness of the bound variable: $\mathsf{freshAbs}\ ys\ y\ (\mathsf{Abs}\ xs\ x\ X)$ is reducible to $\mathsf{fresh}\ ys\ y\ X$ only if (xs, x) is distinct from (ys, y), $(\mathsf{Abs}\ xs\ x\ X)\,[Y/y]_{ys}$ is expressible in terms of $X\,[Y/y]_{ys}$ only if (xs, x) is distinct from (ys, y) and fresh for Y, etc.

Finally, we prove lemmas that regulate the interaction between the standard operators, in all possible combinations: freshness versus swapping, freshness versus substitution, substitution versus substitution, etc. Here are a few samples:

Prop 10. If the terms X, Y, Y_1, Y_2, Z are good and the assignments ρ, ρ' are $\uparrow\mathsf{good}$, then:
(1) Swapping distributes over all operators, including, e.g., substitution:

$$Y\,[X/x]_{xs}\,[z_1 \wedge z_2]_{zs} = (Y\,[z_1 \wedge z_2]_{zs})\,[(X[z_1 \wedge z_2]_{zs})\,/\,(x[z_1 \wedge z_2]_{xs,zs})]_{xs}$$

where $x[z_1 \wedge z_2]_{xs,zs} = (\text{if } xs = zs \text{ then } x[z_1 \wedge z_2] \text{ else } x)$

(2) Substitution of the same variable (and of the same varsort) distributes over itself:

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

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

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

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

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

(5) 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

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

In summary, we have formalized quite exhaustively the general-purpose properties of all syntactic constructors and standard operators. Some of these properties are subtle. In 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 on the particular syntax.

4 Reasoning and Definition Principles

We formalize schemes for induction (4.1), recursion and semantic interpretation (4.2) that realize the Barendregt convention and are compatible with the standard operators.

4.1 Fresh Induction

We introduce fresh induction by an example. To prove Prop. 10(4), we use (mutual) structural induction over terms and abstractions, proving the statement together with the corresponding statement for abstractions, $\mathsf{freshAbs}\ zs\ z\ (A[Y/y]_{ys}) \Longleftrightarrow ((zs,z) = (ys,y) \vee \mathsf{freshAbs}\ zs\ z\ A) \wedge (\mathsf{freshAbs}\ ys\ y\ A \vee \mathsf{fresh}\ zs\ z\ Y)$. 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. 9(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 [41,55,57], 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 |**var**| (in particular, if they have only finitely many).

Theorem 11. Let φ : **term** $\to$ **param** $\to$ **bool** and φAbs : **abs** $\to$ **param** $\to$ **bool**. Assume:
(1) $\forall xs, p.\ |\mathsf{varsOf}\ xs\ p| < |\mathbf{var}|$
(2) $\forall xs, x, p.\ \varphi\ (\mathsf{Var}\ xs\ x)\ p$
(3) $\forall \delta, inp, binp, p.\ |\mathsf{dom}\ inp| < |\mathbf{var}| \wedge |\mathsf{dom}\ binp| < |\mathbf{var}| \wedge \uparrow(\lambda X.\ \mathsf{good}\ X \wedge (\forall q.\ \varphi\ X\ q))\ inp \wedge \uparrow(\lambda A.\ \mathsf{goodAbs}\ A \wedge (\forall q.\ \varphi Abs\ A\ q))\ binp \Longrightarrow \varphi\ (\mathsf{Op}\ \delta\ inp\ binp)\ p$
(4) $\forall xs, x, X, p.\ \mathsf{good}\ X \wedge \varphi\ X\ p \wedge \boxed{x \notin \mathsf{varsOf}\ xs\ p} \Longrightarrow \varphi Abs\ (\mathsf{Abs}\ xs\ x\ X)\ p$

Then $\forall X, p.\ \mathsf{good}\ X \Longrightarrow \varphi\ X\ p$ and $\forall A, p.\ \mathsf{goodAbs}\ A \Longrightarrow \varphi Abs\ A\ p$.

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

Back to Prop. 10(4), this follows automatically by fresh induction (plus the shown simplifications), after recognizing as parameters the variables (ys, y) and (zs, z) and the term Y—formally, taking **param** $= (\mathbf{varsort} \times \mathbf{var})^2 \times \mathbf{term}$ and $\mathsf{varsOf}\ xs\ ((ys, y), (zs, z), Y) = \{y \mid xs = ys\} \cup \{z \mid xs = zs\} \cup \{x \mid \neg\, \mathsf{fresh}\ xs\ x\ Y\}$.

4.2 Freshness- and Substitution- Sensitive Recursion

A *freshness-substitution (FS) 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: VAR : **varsort** $\to$ **var** $\to$ **T**, OP : **opsym** $\to$ (**index**, **T**) **input** $\to$ (**bindex**, **A**) **input** $\to$ **T**, ABS : **varsort** $\to$ **var** $\to$ **T** $\to$ **A**
- operations corresponding to freshness and substitution: FRESH : **varsort** $\to$ **var** $\to$ **T** $\to$ **bool**, $\mathsf{FRESHABS}$: **varsort** $\to$ **var** $\to$ **A** $\to$ **bool**, _[_/_]_ : **T** $\to$ **T** $\to$ **var** $\to$ **varsort** $\to$ **T** and _[_/_]_ : **A** $\to$ **T** $\to$ **var** $\to$ **varsort** $\to$ **A**

and it is required to satisfy the analogues of:

- the implicational simplification rules for fresh from Prop. 7 (for example, $(ys, y) \neq (xs, x) \Longrightarrow \mathsf{FRESH}\ ys\ y\ (\mathsf{VAR}\ xs\ x)$)
- the simplification rules for substitution from Prop. 9
- the substitution-based abstraction equality rules from Prop. 6

Theorem 12. The good terms and abstractions form the initial FS model. Namely, for any FS model as above, there exist the functions f : **term** $\to$ **T** and $fAbs$: **abs** $\to$ **A** that commute, on good terms, with the constructors and with substitution and preserve

freshness:

$f\,(\mathsf{Var}\ xs\ x) = \mathsf{VAR}\ xs\ x$ $\qquad$ $f\,(\mathsf{Op}\ \delta\ inp\ binp) = \mathsf{OP}\ \delta\ (\uparrow f\ inp)\ (\uparrow fAbs\ binp)$

$fAbs\,(\mathsf{Abs}\ xs\ x\ X) = \mathsf{ABS}\ xs\ x\ (f\ X)$

$f\,(X\,[Y/y]_{ys}) = (f\ X)\,[(f\ Y)/y]_{ys}$ $\qquad$ $fAbs\,(A\,[Y/y]_{ys}) = (fAbs\ A)\,[(f\ Y)/y]_{ys}$

$\mathsf{fresh}\ xs\ x\ X \implies \mathsf{FRESH}\ xs\ x\ (f\ X)$ $\qquad$ $\mathsf{freshAbs}\ xs\ x\ A \implies \mathsf{FRESHABS}\ xs\ x\ (fAbs\ A)$

In addition, the two functions are uniquely determined on good terms and abstractions, in that, for all other functions g : **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 a primitive recursion principle. Consider first 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 primitive recursion, 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 recursion principle 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."

This recursion principle was discussed in [44] for the syntax of λ-calculus and shown to have many useful applications. Perhaps the most useful one is the seamless interpretation of syntax in semantic domains, in a manner that is guranteed to be compatible with alpha, substitution and freshness. We formalize this in our general setting:

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 13. 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:

- $\mathsf{sem}\,(\mathsf{Var}\ xs\ x)\,\rho = \rho\ xs\ x$
- $\mathsf{sem}\,(\mathsf{Op}\ \delta\ inp\ binp)\,\rho = \mathsf{op}\ \delta\ (\uparrow(\lambda X.\,\mathsf{sem}\ X\ \rho)\ inp)\ (\uparrow(\lambda A.\,\mathsf{semAbs}\ A\ \rho)\ binp)$
- $\mathsf{semAbs}\,(\mathsf{Abs}\ xs\ x\ X)\,\rho = \mathsf{abs}\ xs\,(\lambda d.\,\mathsf{sem}\ X\ (\rho[(xs,x) \leftarrow d]))$

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 but on (xs,x)

Theorem 13 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 nothing but the notion that the interpretation only depends on the free variables, whereas 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.

This theorem follows by an instantiation of the recursion Theorem 12: 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—e.g., VAR $xs\ x$ is $\lambda\rho.\ \rho\ xs\ x$.

5 Sorting the Terms

So far, we have a framework where the operations take as free and bound inputs partial families of terms and abstractions. All theorems refer to good (i.e., sufficiently low-branching) terms and abstractions. However, we promised a theory that is applicable to terms over many-sorted binding signatures. Thanks to the choice of a flexible notion of input, it is not hard to cast our results into such a many-sorted setting. Given a suitable notion of signature (5.1), we classify terms according to sorts (5.2) and prove that well-sorted terms are good (5.3)—this gives us sorted versions of all theorems (5.4).

5.1 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:

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

Thus, a signature prescribes which varsorts correspond to which sorts (as discussed in Section 2.4) and, for each operation symbol, which are the sorts of its free inputs (the arity), of its bound (abstraction) inputs (the bound arity), and of its result.

When we give examples for our concrete syntaxes in Section 2, we will write $(i_1 \mapsto a_1, \ldots, i_n \mapsto a_n)$ for the partial function that sends each i_k to Some a_k and everything else to None. In particular, $()$ denotes the totally undefined function.

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

For the CCS example in Section 2.5, we fix a type **chan** of channels. We choose a cardinal upper bound κ for the branching of sum ($\sum$), and choose a type **index** of cardinality κ. For **bindex**, we do not need anything special, so we take it to be **nat**. We have two sorts, of expressions and processes, so we take **sort** = $\{\mathsf{exp}, \mathsf{proc}\}$. Since we have

expression variables but no process variables, we take **varsort** $= \{\mathsf{varexp}\}$ and asSort to send varexp to exp. We define **opsym** as the following datatype: **opsym** $=$ $\mathsf{Zero} \mid \mathsf{Plus} \mid \mathsf{Inp}$ **chan** $\mid \mathsf{Out}$ **chan** $\mid \sum$ (**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 $\mathsf{arOf}\ (\sum I) = ((i \in I) \mapsto \mathsf{proc})$.

5.2 Well-Sorted Terms Over a Signature

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

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

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

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

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

Let us illustrate sorting for our running examples. In the λ-calculus syntax, let $X = \mathsf{Var}\ \mathsf{lam}\ x$, $A = \mathsf{Abs}\ \mathsf{lam}\ x\ X$, and $Y = \mathsf{Op}\ \mathsf{Lam}\ ()\ (0 \mapsto A)$. These correspond to what, in the unsorted BNF notation from Section 2.1, we would write $\mathsf{Var}\ x$, $\mathsf{Abs}\ x\ X$ and $\mathsf{Lam}\ (\mathsf{Abs}\ x\ X)$. In our sorting system, X and Y are both well-sorted terms at sort lam (written $\mathsf{wls}\ \mathsf{lam}\ X$ and $\mathsf{wls}\ \mathsf{lam}\ Y$) and A is a well-sorted abstraction at sort $(\mathsf{lam}, \mathsf{lam})$ (written $\mathsf{wlsAbs}\,(\mathsf{lam}, \mathsf{lam})\,A$).

For CCS, we have that $E = \mathsf{Op}\ \mathsf{Zero}\,()\,()$ and $F = \mathsf{Op}\ \mathsf{Plus}\ (0 \mapsto E,\ 1 \mapsto E)\,()$ are well-sorted terms of sort exp. Moreover, $P = \mathsf{Op}\,(\sum \emptyset)\,()\,()$ and $Q = \mathsf{Op}\,(\mathsf{Out}\ c)\,(0 \mapsto F, 1 \mapsto P)\,()$ are well-sorted terms of sort proc. (Note that P is a sum over the empty set of choices, i.e., the null process, whereas Q represents a process that outputs the value of $0+0$ on channel c and then stops.) If, e.g., we swap the arguments of $\mathsf{Out}\ c$ 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 $\mathsf{Out}\ c$, $(0 \mapsto \mathsf{exp}, 1 \mapsto \mathsf{proc})$.

5.3 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 14. $|\mathbf{sort}| < |\mathbf{var}| \wedge |\mathbf{index}| < |\mathbf{var}| \wedge |\mathbf{bindex}| < |\mathbf{var}|$

Now we can prove:

Prop 15. $(\mathsf{wls}\ s\ X \Longrightarrow \mathsf{good}\ X) \wedge (\mathsf{wls}\ (xs, s)\ A \Longrightarrow \mathsf{goodAbs}\ A)$

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

Using the preservation properties and Prop. 15, we transfer the entire theory of Sections 3.4 and 4 from good terms to well-sorted terms—e.g., Prop. 10(2) becomes:

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

The transfer is mostly straightforward for all facts, including the induction theorem. (For stating the well-sorted version of the recursion and semantic interpretation theorems, there is some additional bureaucracy since we also need sorting predicates on the target domain—the extended technical report [**?**] gives details.)

There is an important remaining question: Are our two Assumptions (2 and 14) 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**.

5.4 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 [28] 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 14). 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 and recursion schemes presented in Section 4 are not the only schemes we formalized (but are the most useful ones). We also proved a variety of lower-level induction schemes based on the skeleton of the terms (a generalization of depth for possibly infinitely branching terms) and schemes that are easier to instantiate—e.g., by pre-instantiating Theorem 11 with commonly used parameters such as variables, terms and environments. As for the recursion Theorem 12, we additionally proved a more flexible scheme that allows the recursive argument, and not only the recursive result, to be referred—this is *full-fledged primitive recursion*, whereas Theorem 12 only implements *iteration*. Also, we proved schemes for recursion that factor swapping [38] instead of and in addition to substitution. All together, these constitute 8000 LOC.

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 the recursion Theorem 12 and the interpretation Theorem 13, which require conceptually straightforward but technically tedious moves back and forth between sorted terms and sorted elements of the target domain.

6 Discussion, Related Work and Future Work

There is a large amount of literature on formal approaches to syntax with bindings. (See [1, §2], [18, §6] and [42, §2.10,§3.7] for overviews.) Our work, nevertheless, fills a gap in the literature: It is the first theory of binding syntax mechanized in a universal algebra fashion, i.e., with sorts and many-sorted term constructors specified by a binding signature, as employed in several theoretical developments, e.g., [19, 41, 48, 52]. The universal algebra aspects of our approach are the consideration of an *arbitrary signature* and the singling out of the collection of terms and the operations on them as an *initial object in a category of models/algebras* (which yields a recursion principle). We do not consider arbitrary equational varieties (like in [52]), but only focus on selected equations and Horn clauses that characterize the term models (like in [41]).

Alternatives to Universal Algebra A popular alternative to our universal algebra approach is higher-order abstract syntax (HOAS) [16–18, 24, 25]: the reduction of all bindings to a single binding—that of a fixed λ-calculus. Compared to universal algebra, HOAS's advantage is lighter formalizations, whereas the disadvantage is the need to prove the representation's adequacy (which involves reasoning about substitution) and, in some frameworks, the need to rule out the exotic terms.

Another alternative, very successfully used in HOL-based provers such as HOL4 [51] and Isabelle/HOL, is the "package" approach: Instead of deeply embedding sorts and operation symbols like we do, packages take a user specification of the desired types and operations and prove all the theorems for that instance (on a dynamic basis). Nominal Isabelle [54,56] is a popular such package, which implements terms with bindings for Isabelle/HOL. From a theoretical perspective, a universal algebra theory has a wider appeal, as it models "statically" the meta-theory in its whole generality. However, a package is more practical, since most proof assistant users only care about the particular instance syntax used in their development. In this respect, simply instantiating our signature with the particular syntax is not entirely satisfactory, since it is not sufficiently "shallow"—e.g., one would like to have actual operations such as Lam instead of applications of Op to a Lam operation symbol, and would like to have actual types, such as **exp** and **proc**, instead of the well-sortedness predicate applied to sorts, wls exp and wls proc. For our applications, so far we have manually transited from our "deep" signature instances to the more usable shallow version sketched above. In the future, we plan to have this transit process automated, obtaining the best of both worlds, namely a universal algebra theory that also acts as a *statically certified* package. (This approach has been prototyped for a smaller theory: that of nonfree equational datatypes [49].)

Theory of Substitution and Semantic Interpretation The main goal of our work was the development of as much as possible from the theory of syntax for an arbitrary syntax. To our knowledge, none of the existing frameworks provides support for

substitution and the interpretation of terms in semantic domains at this level of generality. Consequently, formalizations for concrete syntaxes, even those based on sophisticated packages such as Nominal Isabelle or the similar tools and formalizations in Coq [2, 3, 27], have to redefine these standard concepts and prove their properties over and over again—an unnecessary consumption of time and brain power.

Induction and Recursion Principles There is a rich literature on these topics, which are connected to the quest, pioneered by Gordon and Melham [23], of understanding terms with bindings modulo alpha as an abstract datatype. We formalized the Nominal structural induction principle from [41], which is also implemented in Nominal Isabelle. By contrast, we did not go after the Nominal recursion principle. Instead, we chose to stay more faithful to the abstract datatype desideratum, generalizing to an arbitrary syntax our own schema for substitution-aware recursion [44] and Michael Norrish's schema for swapping-aware recursion [38]—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 [44, §3,§8].

Generality of the Framework Our constructors are restricted to binding at most one variable in each input—a limitation that makes our framework far from ideal for representing complex binders such as the let patterns of POPLmark's Challenge 2B. In contrast, the specification language Ott [50] and Isabelle's Nominal2 package [56] were specifically designed to address such complex, possibly recursive binders. Incidentally, the Nominal2 package also separates abstractions from terms, like we do, but their abstractions are significantly more expressive; their terms are also quotiented to alpha-equivalence, which is defined via flattening the binders into finite sets or lists of variables (atoms).

On the other hand, to the best of our knowledge, our formalization is the first to capture infinitely branching terms and our foundation of alpha equivalence on the regularity of $|\mathbf{var}|$ is also a theoretical novelty—constituting a less exotic alternative to Murdoch Gabbay's work on infinitely supported objects in nonstandard set theory [20]. This flexibility would be needed to formalize calculi such as infinite-choice process algebra, for which infinitary structures have been previously employed to give semantics [31].

Future Generalizations and Integrations Our theory currently addresses mostly *structural* aspects of terms. A next step would be to cover *behavioral* aspects, such as formats for SOS rules and their interplay with binders, perhaps building on existing Isabelle formalizations of process algebras and programming languages (e.g., [5, 30, 36, 43, 46, 47]).

Another exciting prospect is the integration of our framework with Isabelle's recent package for inductive and coinductive datatypes [10] based on bounded natural functors (BNFs), which follows a compositional design [53] and provides flexible ways to nest types [11] and mix recursion with corecursion [9, 14], but does not yet cover terms with bindings. Achieving compositionality in the presence of bindings will require a substantial refinement of the notion of BNF (since terms with bindings form only partial functors w.r.t. their sets of free variables).

Acknowledgment We thank the anonymous reviewers for suggesting textual improvements. Popescu has received funding from UK's Engineering and Physical Sciences Research Council (EPSRC) via the grant EP/N019547/1, Verification of Web-based Systems (VOWS).

References

1. The POPLmark challenge (2009), http://fling-l.seas.upenn.edu/ plclub/cgi-bin/poplmark/
2. Aydemir, B.E., Bohannon, A., Weirich, S.: Nominal reasoning techniques in coq: (extended abstract). Electr. Notes Theor. Comput. Sci. 174(5), 69–77 (2007)
3. Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: POPL 2008. pp. 3–15 (2008)
4. Barendregt, H.P.: The Lambda Calculus. North-Holland (1984)
5. Bengtson, J., Parrow, J., Weber, T.: Psi-calculi in isabelle. J. Autom. Reasoning 56(1), 1–47 (2016)
6. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL—Lessons learned in formal-logic engineering. In: TPHOLs '99. vol. 1690, pp. 19–36 (1999)
7. Blanchette, J.C., Popescu, A.: Mechanizing the metatheory of Sledgehammer. In: FroCoS. pp. 245–260 (2013)
8. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. In: TACAS. pp. 493–507 (2013)
9. 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)
10. 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)
11. Blanchette, J.C., Meier, F., Popescu, A., Traytel, D.: Foundational nonuniform (co)datatypes for higher-order logic. In: LICS. IEEE (2017)
12. Blanchette, J.C., Popescu, A., Traytel, D.: Cardinals in Isabelle/HOL. In: ITP. pp. 111–127 (2014)
13. Blanchette, J.C., Popescu, A., Traytel, D.: Unified classical logic completeness—A coinductive pearl. In: IJCAR 2014. pp. 46–60 (2014)
14. Blanchette, J.C., Popescu, A., Traytel, D.: Foundational extensible corecursion: a proof assistant perspective. In: ICFP. pp. 192–204 (2015)
15. Blanchette, J.C., Popescu, A., Traytel, D.: Soundness and completeness proofs by coinductive methods. J. Autom. Reasoning 58(1), 149–179 (2017)
16. Chlipala, A.J.: Parametric higher-order abstract syntax for mechanized semantics. In: ICFP. pp. 143–156 (2008)
17. Despeyroux, J., Felty, A.P., Hirschowitz, A.: Higher-order abstract syntax in Coq. In: TLCA. pp. 124–138 (1995)
18. 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)
19. Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding (extended abstract). In: LICS 1999. pp. 193–202 (1999)
20. Gabbay, M.J.: A general mathematics of names. Information and Computation 205(7), 982 – 1011 (2007)
21. Gheri, L., Popescu, A.: This paper's homepage. `http://andreipopescu.uk/papers/BindingTheory.html`
22. Gheri, L., Popescu, A.: A formalized general theory of syntax with bindings. CoRR (2017)
23. Gordon, A.D., Melham, T.F.: Five axioms of alpha-conversion. In: TPHOLs. pp. 173–190 (1996)
24. 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)
25. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. In: LICS 1987. pp. 194–204. IEEE, Computer Society Press (1987)
26. Hennessy, M., Milner, R.: On observing nondeterminism and concurrency. In: ICALP. pp. 299–309 (1980)

27. Hirschowitz, A., Maggesi, M.: Nested abstract syntax in Coq. Journal of Automated Reasoning 49(3), 409–426 (2012)
28. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales—a sectioning concept for Isabelle. In: TPHOLs. pp. 149–166 (1999)
29. Keisler, H.J.: Model Theory for Infinitary Logic. North-Holland (1971)
30. Lochbihler, A.: Java and the Java memory model—A unified, machine-checked formalisation. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 497–517. Springer (2012)
31. Luttik, B.: Choice Quantification in Process Algebra. Ph.D. thesis, University of Amsterdam (April 2002)
32. Miller, D., Tiu, A.: A proof theory for generic judgments. ACM Transactions on Computational Logic 6(4), 749–783 (2005)
33. Milner, R.: Communication and concurrency. Prentice Hall (1989)
34. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge (2001)
35. Nipkow, T., Klein, G.: Concrete Semantics: With Isabelle/HOL. Springer (2014)
36. Nipkow, T., von Oheimb, D.: Java$_{light}$ is type-safe - definitely. In: POPL. pp. 161–170 (1998)
37. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer (2002)
38. 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)
39. Norrish, M., Vestergaard, R.: Proof pearl: De bruijn terms really do work. In: TPHOLs
40. Pitts, A.M.: Nominal logic: A first order theory of names and binding. In: TACS. pp. 219–242 (2001)
41. Pitts, A.M.: Alpha-structural recursion and induction. J. ACM 53(3) (2006)
42. Popescu, A.: Contributions to the theory of syntax with bindings and to process algebra, phD thesis, Univ. of Illinois, 2010. Available at `andreipopescu.uk/thesis.pdf`
43. Popescu, A., Gunter, E.L.: Incremental pattern-based coinduction for process algebra and its Isabelle formalization. In: FOSSACS'10 (2010)
44. Popescu, A., Gunter, E.L.: Recursion principles for syntax with bindings and substitution. In: ICFP. pp. 346–358 (2011)
45. 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)
46. Popescu, A., Hölzl, J., Nipkow, T.: Proving concurrent noninterference. In: CPP. pp. 109–125 (2012)
47. Popescu, A., Hölzl, J., Nipkow, T.: Formalizing probabilistic noninterference. In: CPP. pp. 259–275 (2013)
48. Popescu, A., Rosu, G.: Term-generic logic. Theor. Comput. Sci. 577, 1–24 (2015)
49. Schropp, A., Popescu, A.: Nonfree datatypes in isabelle/hol - animating a many-sorted metatheory. In: CPP. pp. 114–130 (2013)
50. 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)
51. Slind, K., Norrish, M.: A brief overview of HOL4. In: TPHOLs. pp. 28–32 (2008)
52. Sun, Y.: An algebraic generalization of Frege structures—binding algebras. Theor. Comput. Sci. 211(1-2), 189–232 (1999)
53. Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In: LICS 2012, pp. 596–605. IEEE (2012)
54. Urban, C.: Nominal techniques in Isabelle/HOL. J. Autom. Reasoning 40(4), 327–356 (2008)
55. Urban, C., Berghofer, S., Norrish, M.: Barendregt's variable convention in rule inductions. In: CADE. pp. 35–50 (2007)
56. Urban, C., Kaliszyk, C.: General bindings and alpha-equivalence in Nominal Isabelle. In: ESOP. pp. 480–500 (2011)
57. Urban, C., Tasson, C.: Nominal techniques in Isabelle/HOL. In: CADE. pp. 38–53 (2005)