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

Case Studies in Formal Reasoning About Lambda-Calculus: Semantics, Church-Rosser, Standardization and HOAS

Lorenzo Gheri · **Andrei Popescu**

Received: date / Accepted: date

Abstract We have previously published the Isabelle/HOL formalization of a general theory of syntax with bindings. In this paper, we instantiate the general theory to the syntax of lambda-calculus and formalize the development leading to several fundamental constructions and results: sound semantic interpretation, the Church-Rosser and standardization theorems, and higher-order abstract syntax (HOAS) encoding. For Church-Rosser and standardization, our work covers both the call-by-name and call-by-value versions of the calculus, following classic papers by Takahashi and Plotkin. During the formalization, we were able to stay focused on the high-level ideas of the development—thanks to the arsenal provided by our general theory: a wealth of basic facts about the substitution, swapping and freshness operators, as well as recursive-definition and reasoning principles, including a specialization to semantic interpretation of syntax.

1 Introduction

Formal reasoning about syntax with bindings is a notoriously challenging problem, due to the difficulty of handling binding-specific aspects such as alpha-equivalence (also known as naming equivalence), capture-avoiding substitution of terms for variables, and the generation of variables that are fresh in certain contexts.

Informal techniques aimed at easing the reasoning tasks have turned out to be very difficult to represent formally, partly due to their reliance on unstated assumptions without which they would be unsound. For example, the majority of textbooks on λ-calculi (including the most standard one [12]) employ the principle of primitive recursion to define functions on λ-terms, after which they tacitly assume these functions to be invariant under alpha-equivalence; as another example, the so-called Barendregt variable convention assumes that, in a proof or definition context, the bound variables are fresh for all the param-

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

eters located outside the scope of their binders. Both these principles are unsound in general, that is, if employed without checking some sanity conditions on the defining clauses or on the definition and proof context.

Formal reasoning frameworks have been designed to recover such informal principles on a sound basis. The approaches range from a clever manipulation of the bound variables as in nominal logic and the locally named representation [65, 69, 77] to the removal of the very notion of bound variable—by either encoding away bound variables as numeric positions in terms as in de Bruijn-style and locally nameless representations [21, 22, 31] or by representing them using meta-variables as in higher-order abstract syntax (HOAS) [29, 40].

Our own framework [37] takes a nominal-style approach. The framework is formalized in the Isabelle/HOL proof assistant as a many-sorted theory parameterized over a binding signature. Its distinguishing features (some of which also set it apart from nominal logic) are a rich built-in theory of substitution, swapping and freshness, as well as recursion principles that are sensitive to these operators.

In previous work, we have deployed our framework to formalize classic results in many-sorted first-order logic (completeness of deduction and soundness of Skolemization [17, 19, 20]) and System F (strong normalization [72]) and novel results about the meta-theory of Isabelle's Sledgehammer tool [17, 18]. However, in the papers describing these applications we have emphasized neither (1) the general theory underlying our framework nor (2) the framework's deployment to support reasoning within these applications. The first gap has been filled in a recent paper [37]. The second gap is being filled by the current paper, which is intended as a companion to [37].

This paper presents the instantiation of our framework to support the development of some fundamental constructions and results in λ-calculus with β-reduction: soundness of semantic interpretation, the Church-Rosser and standardization theorems, and adequacy of a higher-order abstract syntax (HOAS) encoding.[1] The Church-Rosser and standardization theorems are established for both the call-by-name and call-value variants.[2]

The first step we take is instantiating the framework to the syntaxes of call-by-name and call-by-value λ-calculus, the latter differing from the former by the existence of an additional syntactic category of special terms called values. These instantiations provide us with a rich theory of the standard operators on terms, namely freshness, substitution and swapping, as well as a freshness-aware induction proof principle and operator-aware recursive definition principles, including a variant specialized to semantic interpretation (Section 2).

Then we proceed with the formal development of our specific target results. We only show in detail the development for the call-by-name calculus (Section 3). The similar Church-Rosser and standardization development for the call-by-value calculus is only sketched by pointing out the differences, including the use of a two-sorted instantiation of our framework (Section 4).

The results require the definition of standard β-reduction and β-equivalence (Section 3.1), including variations such as parallel and left β-reduction. Semantic interpretation is defined in Henkin-style models, and takes full advantage of our framework's built-in semantic features (Section 3.2). The Church-Rosser theorem (Section 3.3) is proved by formalizing the parallel-reduction technique of Tait [12], enhanced with the complete parallel reduction operator trick due to Takahashi [76]. For standardization (Section 3.4), we follow closely Plotkin's original paper [67]. As higher-order abstract syntax case study, we consider a

[1] We emphasize that this is a case study in formalizing the meta-theory of HOAS-style encoding; our framework itself does not follow the HOAS methodology.

[2] Our formalization is publicly available from the paper's website [36].

simple encoding of λ-calculus in itself (Section 3.5); we believe our employed technique would scale to formalizing the similar (though admittedly more complex) phenomena in full-fledged logical frameworks.

Our presentation emphasizes the use of the various principles provided by our framework, as well as some difficulties arising from representing formally some informal definition and proof idioms—such as recursing over alpha-equated terms (or, equivalently, recursing in an alpha-equivalence preserving manner) and inversion rules obeying Barendregt's variable convention. Some of the lessons learned during the formalization effort, as well as some statistics, are presented in Section 5. We conclude with an overview of related work (Section 6).

2 Instantiation of the General Framework

Our framework [37] is parameterized by a binding signature, which essentially specifies the following data: a collection of term sorts, a collection of variable sorts,[3] an embedding relationship between variable sorts and term sorts, and a collection of (term) constructors, each with an assigned arity and an assigned result sorts.

The theory was developed over an arbitrary signature, which is represented as an Isabelle locale [49]. Namely, "quasi-terms" were defined as being freely generated by the constructors, then terms were defined by quotienting quasi-terms to the notion of alpha-equivalence obtained standardly from the signature-specified bindings of the term constructors. Thus, what we call "terms" in this paper are alpha-equivalence classes. Several standard operators were defined on terms, including capture-avoiding substitution of terms for variables, freshness of a variable for a term, and swapping of two variables in a term. The theory provides many properties of these operators, as well as binding-aware and standard-operator-aware structural recursion and induction principles and a principle for interpreting syntax in a semantic domain.

Our companion paper [37] gives details about this general framework. However, understanding these details is not necessary for following the rest of this paper, which gives a self-contained description of two instances of the framework.

2.1 The syntax of λ-calculus

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

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

where X and Y range over terms (the ones generated by the grammar), x over a given infinite type **var** of variables of variables and c over a given type **const** of constants—where Var and Ct are the embeddings of variables and constants into terms, App is application and Lm is λ-abstraction. Terms are assumed to be equated modulo alpha-equivalence, defined standardly by assuming that, in $\mathsf{Lm}\ x\ X$, the λ-constructor Lm binds the variable x in the term X. Thus, for example, $\mathsf{Lm}\ x\ (\mathsf{Var}\ x) = \mathsf{Lm}\ y\ (\mathsf{Var}\ y)$ even if $x \neq y$.

We obtain the above syntax by picking a particular binding signature (with a single sort of variables and a single sort of terms, and, with the desired constructors). In Isabelle, picking a signature corresponds to instantiating the corresponding locale. In addition to this straightforward instantiation, we also perform a formal transfer of all the concepts and results to a more shallow (and hence more usable) Isabelle representation. This involves

[3] Even though variables of all sorts behave essentially the same, it is useful that they are presented as different collections, belonging to different sorts—e.g., individual and set variables in second-order logic, or channel names and process names in process calculi.

creating native Isabelle/HOL types of terms for each sort of the signature and transferring all the term constructors and operators and all facts about them to these native types. The process is conceptually straightforward, but is quite tedious, and must be done by hand since we have not yet automated it. [37, §6.5] offers more details, and [74, §5] presents the automation of a similar kind of transfer (for nonfree datatypes).

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

- the constructors, namely Var : **var** $\to$ **term**, Ct : **const** $\to$ **term**, App : **term** $\to$ **term** $\to$ **term** and Lm : **var** $\to$ **term** $\to$ **term**
- and the standard operators:
  - depth (height) of a term, depth : **term** $\to$ **nat**
  - freshness of a variable in a term,[4] fresh : **var** $\to$ **term** $\to$ **bool**
  - (capture-avoiding) substitution of a term for a variable in a term, _[_/_] : **term** $\to$ **term** $\to$ **var** $\to$ **term**
  - (capture-avoiding) parallel substitution of multiple terms for multiple variables in a term, _[_] : **term** $\to$ (**var** $\to$ **term option**) $\to$ **term**
  - swapping of two variables in a term,[5] _[_$\leftrightharpoons$_] : **term** $\to$ **var** $\to$ **var** $\to$ **term**

From our general theory, we also obtain for free:

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

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

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

Another example is the compositionality of substitution:

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

Fresh structural induction. Our framework also offers a structural induction principle in the style of nominal logic [66, 79, 82]. It differs from standard structural induction in that, in the inductive Lm-case, it allows one to additionally assume freshness of the Lm-bound variable with respect to any potential parameters of the to-be-proved statement. For the λ-calculus instance, it becomes:

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

[5] While not explicitly present in the traditional λ-calculus [12], swapping has been popularized by nominal logic as a very convenient operator in bootstrapping definitions—thanks to the fact that bijective renamings behave better than arbitrary renamings with respect to bindings [66].

Prop 3. (Fresh structural induction principle) *Let* **param** *be a type (of items called parameters) endowed with a function* varsOf : **param** → **var set** *such that* varsOf *p is finite for all p* : **param**. *Let* φ : **term** → **param** → **bool** *be a predicate on terms and parameters.*

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

Then φ *X p holds for all X* : **term** *and p* : **param**.

For details on the wide applicability of this parameter-based fresh induction principle we refer the reader to [82]. The parameters are typically the other terms and variables appearing in a statement, different from the term on which we induct. A classic example is the proof of substitution compositionality, our Prop. 2(2)—which can be done by fresh induction on X taking as parameters all the other terms and variables, namely Y, y, Z and z. In the Lm-case, thanks to the extra freshness assumption, we can soundly invoke Barendregt's variable convention and assume, for example, that in the expression $(\mathsf{Lm}\ x\ X)\,[Y/y]\,[Z/z]$ we have x fresh for Y, y, Z and z—which allows reducing the expression to $\mathsf{Lm}\ x\ (X\ [Y/y]\,[Z/z])$ and then applying the induction hypothesis. By contrast, applying standard induction would have brought serious complications concerning variable renaming.

Prop. 3 immediately implies the following fresh case distinction principle. It states that any term is either a variable, or a constant, or an application, or an abstraction whose bound variable can be taken to be fresh for a given parameter.

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

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

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

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

Def 5. *A* freshness-substitution model (FSb model) *is a type* **D** *endowed with the following:*

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

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

F1: $\mathsf{FRESH}\ x\ (\mathsf{Ct}\ c)\ (\mathsf{CT}\ c)$
F2: $x \neq z$ *implies* $\mathsf{FRESH}\ z\ (\mathsf{Var}\ x)\ (\mathsf{VAR}\ x)$
F3: $\mathsf{FRESH}\ z\ X'\ X$ *and* $\mathsf{FRESH}\ z\ Y'\ Y$ *implies* $\mathsf{FRESH}\ z\ (\mathsf{App}\ X'\ Y')\ (\mathsf{APP}\ X'\ X\ Y'\ Y)$
F4: $\mathsf{FRESH}\ z\ (\mathsf{Lm}\ z\ X')\ (\mathsf{LM}\ z\ X'\ X)$
F5: $\mathsf{FRESH}\ z\ X'\ X$ *implies* $\mathsf{FRESH}\ z\ (\mathsf{Lm}\ x\ X')\ (\mathsf{LM}\ x\ X'\ X)$
Sb1: $\mathsf{SUBST}\ (\mathsf{Var}\ z)\ (\mathsf{VAR}\ z)\ Z'\ Z\ z = Z$
Sb2: $x \neq z$ *implies* $\mathsf{SUBST}\ (\mathsf{Var}\ x)\ (\mathsf{VAR}\ x)\ Z'\ Z\ z = \mathsf{VAR}\ x$
Sb3: $\mathsf{SUBST}\ (\mathsf{App}\ X'\ Y')\ (\mathsf{APP}\ X'\ X\ Y'\ Y)\ Z'\ Z\ z =$
$\mathsf{APP}\ (X'[Z'\ /\ z])\ (\mathsf{SUBST}\ X'\ X\ Z'\ Z\ z)\ (Y'[Z'\ /\ z])\ (\mathsf{SUBST}\ Y'\ Y\ Z'\ Z\ z)$
Sb4: $x \neq z$ *and* $\mathsf{FRESH}\ x\ Z'\ Z$ *implies*
$\mathsf{SUBST}\ (\mathsf{Lm}\ z\ X')\ (\mathsf{LM}\ x\ X'\ X)\ Z'\ Z\ z = \mathsf{LM}\ x\ (X'[Z'\ /\ z])\ (\mathsf{SUBST}\ X'\ X\ Z'\ Z\ z)$
SbRn: $x \neq y$ *and* $\mathsf{FRESH}\ y\ X'\ X$ *implies*
$\mathsf{LM}\ y\ (X'[(\mathsf{Var}\ y)/x])\ (\mathsf{SUBST}\ X'\ X\ (\mathsf{Var}\ y)\ (\mathsf{VAR}\ y)\ x) = \mathsf{LM}\ x\ X'\ X$

Def 6. *A* freshness-swapping model (FSw model) *is similar to an FSb model, except that it has a swapping-like function* $\mathsf{SWAP} : \mathbf{term} \to \mathbf{D} \to \mathbf{var} \to \mathbf{var} \to \mathbf{D}$ *instead of the substitution-like function* SUBST *and satisfies the following swapping clauses (Sw1)–(Sw4) and swapping-congruence clause (SwCg) instead of the substitution-related clauses (Sb1)–(Sb4) and (SbRn):*

Sw1: $\mathsf{SWAP}\ (\mathsf{Ct}\ c)\ (\mathsf{CT}\ c)\ z_1\ z_2\ =\ \mathsf{CT}\ c$
Sw2: $\mathsf{SWAP}\ (\mathsf{Var}\ x)\ (\mathsf{VAR}\ x)\ z_1\ z_2\ =\ \mathsf{VAR}\ (x[z_1 \leftrightharpoons z_2])$
Sw3: $\mathsf{SWAP}\ (\mathsf{App}\ X'\ Y')\ (\mathsf{APP}\ X'\ X\ Y'\ Y)\ z_1\ z_2\ =$
$\mathsf{APP}\ (X'[z_1 \leftrightharpoons z_2])\ (\mathsf{SWAP}\ X'\ X\ z_1\ z_2)\ (Y'[z_1 \leftrightharpoons z_2])\ (\mathsf{SWAP}\ Y'\ Y\ z_1\ z_2)$
Sw4: $\mathsf{SWAP}\ (\mathsf{Lm}\ x\ X')\ (\mathsf{LM}\ x\ X'\ X)\ z_1\ z_2 = \mathsf{LM}\ (x[z_1 \leftrightharpoons z_2])\ (X'[z_1 \leftrightharpoons z_2])\ (\mathsf{SWAP}\ X'\ X\ z_1\ z_2)$
SwCg: $\mathsf{FRESH}\ z\ X'\ X$ *and* $\mathsf{FRESH}\ z\ Y'\ Y$ *and* $z \notin \{x, y\}$ *and* $\mathsf{SWAP}\ X'\ X\ z\ x = \mathsf{SWAP}\ Y'\ Y\ z\ y$
implies $\mathsf{LM}\ x\ X'\ X = \mathsf{LM}\ y\ Y'\ Y$

To simplify notation, in what follows we will often refer to FSb models and FSw models simply by their carriers and leave the additional structure implicit, thus writing, e.g., "Let **D** be an FSb model." The framework's recursion principles essentially say that terms form the initial FSb and FSw models:[6]

Prop 7. *Let* **D** *be an FSb model (FSw model, respectively). Then there exists a unique function* $H : \mathbf{term} \to \mathbf{D}$ *commuting with the constructors, i.e.,*
- $H\ (\mathsf{Var}\ x) = \mathsf{VAR}\ x$
- $H\ (\mathsf{Ct}\ c) = \mathsf{CT}\ c$
- $H\ (\mathsf{App}\ X\ Y) = \mathsf{APP}\ X\ (H\ X)\ Y\ (H\ Y)$
- $H\ (\mathsf{Lm}\ x\ X) = \mathsf{LM}\ x\ X\ (H\ X)$

Additionally, H preserves freshness and commutes with substitution (respectively, swapping):
- $\mathsf{fresh}\ x\ X$ *implies* $\mathsf{FRESH}\ x\ X\ (H\ X)$
- $H\ (X[Z\,/\,z]) = \mathsf{SUBST}\ X\ (H\ X)\ Z\ (H\ Z)\ z$
(respectively, $H\ (X[z_1 \leftrightharpoons z_2]) = \mathsf{SWAP}\ X\ (H\ X)\ z_1\ z_2$*)*

[6] The reason why we define our models' operations to act not only on the models' carrier type **D** but also on **term** is to achieve the higher flexibility of *primitive recursion* compared to *iteration*—see [70, §1.4.2] for a detailed discussion of this distinction.

The principle is much easier to use in practice than its elaborate formulation might suggest: Say one wishes to define a function H from **term** to a type **D**. Then the functions on **D** corresponding to the term constructors can be determined from the desired recursive clauses for H. Moreover, the functions on **D** corresponding to freshness and substitution or swapping are determined by the desired behavior of H with respect to these operators, obtained from answering questions such as "How can $H\,(X[Z\,/\,x])$ be expressed in terms of $H\,X$, $H\,Z$ and x?".

We illustrate this methodology by a simple example. (More explanations and examples can be found in [71] and [37], and in this paper's Section 3.3.) Namely, we define no : **term** $\to$ **var** $\to$ **nat**, where no $X\,x$ counts the number of (free) occurrences of the variable x in the term X. We do this using our recursion principle:

Def 8. $\mathsf{no} : \mathbf{term} \to (\mathbf{var} \to \mathbf{nat})$ *is the unique function satisfying the following properties:*

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

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

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

Before formally justifying this definition (i.e., proving that there exists a unique function no satisfying the above clauses), let us explain how the clauses have been produced. First, the clauses for the constructors (Var, Ct, App and Lm) are simply describing the desired recursive behavior of no—which would have been the same had the terms not been considered modulo alpha-equivalence, but as a datatype freely generated from these constructors. However, the problem here is that the terms *are* quotiented, so the constructor clauses are not *a priori* guaranteed to form a correct definition. This is where the remaining clauses, for freshness and substitution, come into play. They have been produced by answering to the following questions: *If* the operator no was already defined, how would it behave w.r.t. freshness and substitution? More precisely:

- What would fresh $x\,X$ imply about the value of no X? Answer: It would imply that this value is 0 at x.
- What would the value of no $(X[Y\,/\,y])$ be, expressed in terms of no X and no Y? Answer: For each variable x, the formula depends on whether x is equal to y, and is the one shown in Def. 8. (This can be easily discovered by drawing a picture of a term X and the free occurrences of y in it, all of which are to be substituted by Y.)

In short, performing a recursive definition in our framework requires:

- a routine part, namely providing the clauses for the constructors, which are immediate if one knows what one wants to define, and
- a somewhat creative (although often easy) "anticipatory" part, namely describing the behavior of the desired operator w.r.t. freshness and substitution or swapping

To justify the above definition, we extract an FSb model obtained from the above clauses in a completely routine fashion. Namely, we take $\mathbf{D} = \mathbf{var} \to \mathbf{nat}$, we define $\mathsf{VAR} : \mathsf{Var} \to \mathbf{D}$ and $\mathsf{SUBST} : \mathbf{term} \to \mathbf{D} \to \mathbf{term} \to \mathbf{D} \to \mathbf{var} \to \mathbf{D}$ by

$$\mathsf{VAR}\ y\ x = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{if } x \neq y \end{cases} \qquad \mathsf{SUBST}\ X\ u\ Y\ v\ y = \lambda x.\ \begin{cases} u\ y * v\ y, & \text{if } x = y \\ u\ x + u\ y * v\ x, & \text{if } x \neq y \end{cases}$$

and similarly for the other constructors and operators.

Verifying Prop. 7's conditions is routine—some simple arithmetics that has been discharged by Isabelle's "auto" proof method. This allows us to apply the conclusion of Prop. 7, obtaining a unique function $\mathsf{no} : \mathbf{term} \to \mathbf{D}$ commuting with the constructors, freshness and substitution—which precisely means satisfying the clauses listed in Def. 8.

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

Now, let us look at some (partial) non-examples. First, consider a function $h : \mathbf{term} \to \mathbf{nat}$ such that $h\ X$ counts the number of free variables of X. It can be of course immediately defined as the cardinal of $\{x \mid \neg\ \mathsf{fresh}\ x\ X\}$, but trying to define it recursively would be difficult (and unnatural)—since we do not have enough information to compute $h\ (\mathsf{App}\ X\ Y)$ from $h\ X$ and $h\ Y$. (We could "force" such a definition by initially counting the variable overlap between X and Y, but this would defeat our purpose, since it would require a function more complicated than h.)

The above non-example applies to our recursion principle, but also to the standard recursion for free datatypes. A more subtle non-example is the depth operator, which we discuss in [71].[7] This can be easily defined recursively for the free datatatype of non-quotiented terms, as well as for the quotiented terms if we use the swapping-based variant of our recursion principle (with FSw-models). However, it cannot be immediately defined using our substitution-based variant (with FSb models), since we cannot express the value of $\mathsf{depth}\ (X[Y\,/\,y])$ from those of $\mathsf{depth}\ X$ and $\mathsf{depth}\ Y$; so in this case the problem is created not by the constructors, but by the substitution operator.

Refinements of recursion. One advantage of our framework's systematic, clause-based take on recursion[8] is the possibility to add optional "packages" that deliver additional properties about the defined functions.

Def 9. *An FSb model (FSw model, respectively) is called* freshness-reversing*, if it satisfies the converses of the clauses F2–F5 in Def. 5 (Def. 6, respectively), namely:*

F2c: $\mathsf{FRESH}\ z\ (\mathsf{Var}\ x)\ (\mathsf{VAR}\ x)$ *implies* $x \neq z$
F3c: $\mathsf{FRESH}\ z\ (\mathsf{App}\ X'\ Y')\ (\mathsf{APP}\ X'\ X\ Y'\ Y)$ *implies* $\mathsf{FRESH}\ z\ X'\ X$ *and* $\mathsf{FRESH}\ z\ Y'\ Y$
F4_5c: $\mathsf{FRESH}\ z\ (\mathsf{Lm}\ x\ X')\ (\mathsf{LM}\ z\ X'\ X)$ *implies* $x = z$ *or* $\mathsf{FRESH}\ z\ X'\ X$

It is called constructor-injective *if its constructor-like operators are injective and mutually exclusive, in that*

- CT *c,* VAR *x,* APP *X Y and* LM *z Z are all distinct*
- CT, VAR, APP *and* LM *are all injective (if we regard* APP *and* LM *as uncurried binary operators)*

The clauses in the above definition are of course satisfied by the term FSb and FSw models. F1c–F3c and F4_5c correspond to inversion properties of freshness w.r.t. the constructors. Note that, being the converse of the "direct" clauses F4 and F5, the clause F4_5c has a disjunction as its conclusion.

Prop 10. *Let* $\mathbf{D}$ *be an FSb model (FSw model, respectively) and let H be the induced recursive function described in Prop. 7. Then the following hold:*

[7] Incidentally, this operators is actually built in our framework, so the user has no need to define it.

[8] More precisely, what we have here are first-order theories consisting of Horn clauses [70].

– *If* **D** *is freshness-reversing, then H (not only preserves, but also) reflects freshness, in that* FRESH $x\ X\ (H\ X)$ *implies* fresh $x\ X$.
– *If* **D** *is constructor-injective, then H is injective.*

The two points of Prop. 10 are, just like Prop. 7, statements of initiality properties (in different categories). This time, terms are being characterized as the initial object in:
– the category of freshness-reversing FSb (FSw) models and freshness-reflecting model morphisms
– the category of constructor-injective FSb (FSw) models and injective model morphisms

Interpretation in semantic domains. Our general framework caters for the semantic interpretation of terms. A semantic domain is a structure consisting of a type for each sort and of a function for each constructor except for the variable-injection one—in such as way that binding inputs in the constructors become second-order inputs in the associated functions. For our particular λ-calculus syntax, this instantiates to the following concept:

Def 11. *A* semantic domain *is a type* **S** *endowed with the functions* $\mathsf{ct} : \mathbf{const} \to \mathbf{S}$, $\mathsf{app} : \mathbf{S} \to \mathbf{S} \to \mathbf{S}$ *and* $\mathsf{lm} : (\mathbf{S} \to \mathbf{S}) \to \mathbf{S}$ *(corresponding to the term constructors* Ct, App *and* Lm*).*

Just like for FSb and FSw models, we will often refer to semantic domains simply by their carriers **S**, leaving the additional structure implicit. The following proposition allows for the interpretation of terms in any semantic domain. It was established generally, for an arbitrary syntax, by appealing to the substitution-based recursion principle. Here is the instance for this syntax:[9]

Prop 12. *Let* **S** *be a semantic domain, and let* **val** *be the type of valuations of variables in the domain,* $\mathbf{var} \to \mathbf{S}$. *Then there exists the unique function* $\mathsf{sem} : \mathbf{term} \to \mathbf{val} \to \mathbf{S}$ *such that:*
– $\mathsf{sem}\ (\mathsf{Var}\ x)\ \rho = \rho\ x$
– $\mathsf{sem}\ (\mathsf{Ct}\ c)\ \rho = \mathsf{ct}\ c$
– $\mathsf{sem}\ (\mathsf{App}\ X\ Y)\ \rho = \mathsf{app}\ (\mathsf{sem}\ X\ \rho)\ (\mathsf{sem}\ Y\ \rho)$
– $\mathsf{sem}\ (\mathsf{Lm}\ x\ X)\ \rho = \mathsf{lm}\ (\lambda s.\ \mathsf{sem}\ X\ (\rho[x \leftarrow s]))$
where $\rho[(x \leftarrow s]$ *is the function* ρ *updated at* x *with* d*—which sends* x *to* d *and any other* y *to* $\rho\ y$.

In addition, the interpretation satisfies the following properties:
– $\mathsf{sem}\ (X[Y\,/\,y])\ \rho = \mathsf{sem}\ X\ (\rho[y \leftarrow \mathsf{sem}\ Y\ \rho])$
– $\mathsf{fresh}\ x\ X$ *and* $\rho =_x \rho'$ *imply* $\mathsf{sem}\ X\ \rho = \mathsf{sem}\ X\ \rho'$
where "$=_x$*" means "equal everywhere except perhaps on* x*"; namely* $\rho =_x \rho'$ *holds iff* $\rho\ y = \rho'\ y$ *for all* $y \neq x$.

The first additional property above states the so-called "substitution lemma," connecting the interpretation of a substituted term to the interpretation of the original term in an updated environment—thus, roughly speaking, connecting syntactic and semantic substitution. The second additional property states that the interpretation of a term is oblivious to how its fresh (non-free) variables are evaluated.

2.2 The two-sorted syntax of λ-calculus with values emphasized

We can split the syntax of λ-calculus in two syntactic categories, by distinguishing the subcategory of values, which consist of variables, constants and Lm-terms. This distinction

[9] In the following definition, we write λ for meta-level functional abstraction, and of course continue to use Lm for the syntactic constructor.

is quite customary when modeling higher-order programming language semantics, where values are the only programs that have a "static" identity (whereas the non-values must be run/evaluated). Thus, we consider the mutually recursive syntactic categories of values, ranged over V, W and (arbitrary) terms, ranged over by X, Y, Z:

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

where Val is the injection of values into terms.

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

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

Applying the same systematic deep-to-shallow transfer process as for the previous one-sorted syntax, we obtain:

- the "native" types **value** and **term** for values and terms
- the expected constructors, e.g., Val : **value** $\to$ **term**
- the standard operators, one for either syntactic category, e.g., $\mathsf{fresh}_{\mathbf{value}}$: **var** $\to$ **value** $\to$ **bool** and $\mathsf{fresh}_{\mathbf{term}}$: **var** $\to$ **term** $\to$ **bool**.

in what follows, we will omit the sort index for the operators, writing, e.g., fresh for both $\mathsf{fresh}_{\mathbf{value}}$ and $\mathsf{fresh}_{\mathbf{term}}$.

The framework-provided induction, recursion and semantic interpretation principles now refer to these mutually recursive types: Induction allows us to prove two simultaneous predicates and recursion/interpretation allows us two define two simultaneous functions, one on values and one on terms—recursion for this syntax will be illustrated in Section 4.

For example, here are the corresponding instances of semantic domain and interpretation:

Def 13. *A* semantic domain *consists of two types,* **S** *and* **Sv***, endowed with the functions* val : **Sv** $\to$ **S***,* app : **S** $\to$ **S** $\to$ **S***,* ct : **const** $\to$ **Sv***, and* lm : (**Sv** $\to$ **S**) $\to$ **S** *(corresponding to the term and value constructors* Val*,* App*,* Ct *and* Lm*).*

Prop 14. *Let* $(\mathbf{S}, \mathbf{Sv})$ *be a semantic domain, and let* **val** *be the type of valuations of variables in the semantic-value carrier of the domain,* **var** $\to$ **Sv**. *Then there exist the unique functions* $\mathsf{sem}_{\mathbf{term}}$: **term** $\to$ **val** $\to$ **S** *and* $\mathsf{sem}_{\mathbf{value}}$: **value** $\to$ **val** $\to$ **S** *such that:*

- $\mathsf{sem}_{\mathbf{term}}\ (\mathsf{Val}\ V)\ \rho = \mathsf{val}\ (\mathsf{sem}_{\mathbf{value}}\ V\ \rho)$
- $\mathsf{sem}_{\mathbf{term}}\ (\mathsf{App}\ X\ Y)\ \rho = \mathsf{app}\ (\mathsf{sem}_{\mathbf{term}}\ X\ \rho)\ (\mathsf{sem}_{\mathbf{term}}\ Y\ \rho)$
- $\mathsf{sem}_{\mathbf{value}}\ (\mathsf{Var}\ x)\ \rho = \rho\ x$
- $\mathsf{sem}_{\mathbf{value}}\ (\mathsf{Ct}\ c)\ \rho = \mathsf{ct}\ c$
- $\mathsf{sem}_{\mathbf{value}}\ (\mathsf{Lm}\ x\ X)\ \rho = \mathsf{lm}\ (\lambda s.\ \mathsf{sem}_{\mathbf{term}}\ X\ (\rho[x \leftarrow s]))$

In addition, the interpretation satisfies the following properties:

- $\mathsf{sem}_{\mathbf{term}}\ (X[V\,/\,y]_{\mathbf{term}})\ \rho = \mathsf{sem}_{\mathbf{term}}\ X\ (\rho[y \leftarrow \mathsf{sem}_{\mathbf{value}}\ V\ \rho])$
- $\mathsf{sem}_{\mathbf{value}}\ (X[V\,/\,y]_{\mathbf{value}})\ \rho = \mathsf{sem}_{\mathbf{value}}\ X\ (\rho[y \leftarrow \mathsf{sem}_{\mathbf{value}}\ V\ \rho])$
- fresh $x\ X$ *and* $\rho =_x \rho'$ *imply* $\mathsf{sem}_{\mathbf{term}}\ X\ \rho = \mathsf{sem}_{\mathbf{term}}\ X\ \rho'$
- fresh $x\ V$ *and* $\rho =_x \rho'$ *imply* $\mathsf{sem}_{\mathbf{value}}\ V\ \rho = \mathsf{sem}_{\mathbf{value}}\ V\ \rho'$

Note that this particular syntax has two sorts of terms (λ-calculus terms and values) and one sort of variables. Consequently, we have two semantic interpretation functions parameterized by one type of valuations.

3 Call-By-Name λ-Calculus

In this section, we show how we have used our framework's infrastructure to formalize some results in the theory of call-by-name (CBN) λ-calculus. We start with defining the CBN β-reduction relation (Section 3.1) and proving its soundness with respect to the semantic interpretation of terms in Henkin-style models (Section 3.2). We continue with proving the Church-Rosser theorem [12], which states that the order in which CBN redexes are reduced is irrelevant "in the long run" (Section 3.3). Then, in a more substantial technical development, we prove the standardization theorem [67], which states that reducibility is not restricted if we impose a canonical reduction strategy, based on identifying left-most redexes (Section 3.4). Finally, we develop and prove adequate a simple HOAS encoding—of λ-calculus into itself (Section 3.5). In each case, we emphasize the use of our framework's various features to leverage the formalization.

All throughout this section, we employ the (single-sorted) syntax of λ-calculus with constants described in Section 2.1. We also fix a partial function Ctapp that shows how to apply a constant c_1 to another constant c_2; $\mathsf{Ctapp}\ c_1\ c_2$ can be either None, meaning "no result," or $\mathsf{Some}\ X$, meaning "the result is X."

3.1 Call-by-name β-reduction

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

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

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

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

Def 15. *The* one-step (CBN) reduction *relation* $\to$: **term** $\to$ **term** $\to$ **bool** *is defined inductively by the following rules:*

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

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

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

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

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

3.2 Soundness of β-equivalence with respect to Henkin-style models

As discussed in Section 2.1, our framework's notion of semantic domain is generic to any binding syntax. In particular cases, it yields meaningful semantic concepts after suitable customization. For example, if we instantiate the framework to first-order logic and choose the semantic operators properly, we obtain the standard notion of first-order model with the Tarskian satisfaction relation [17, §6].

For our syntax of interest, a different kind of customization is necessary. In order to obtain Henkin-style standard notions of set-theoretic models for the λ-calculus [12, 41, 52, 53], we do not need to choose particular semantic operators, but only to axiomatize their behavior. As an example, we pick one such notion, called *environment model* in [52].

Def 16. *An* environment model *is a tuple* $(\mathbf{S}, \mathsf{ct}, \mathsf{app}, \mathsf{Im}, \mathsf{ValidFuns})$ *where* $(\mathbf{S}, \mathsf{ct}, \mathsf{app}, \mathsf{Im})$ *is a semantic domain and* $\mathsf{ValidFuns} \subseteq (\mathbf{S} \to \mathbf{S})$ *a set of functions such that following hold:*

(1) $\mathsf{Ctapp}\ c_1\ c_2 = \mathsf{Some}\ c$ *implies* $\mathsf{app}\,(\mathsf{ct}\ c_1)\,(\mathsf{ct}\ c_2) = \mathsf{ct}\ c$
(2) $f \in \mathsf{ValidFuns}$ *implies* $\mathsf{app}\,(\mathsf{Im}\ f) = f$
(3) $\lambda s.\ \mathsf{sem}\ X\ (\rho[x \leftarrow s]) \in \mathsf{ValidFuns}$

We think of the functions in ValidFuns as those that represent valid semantic behavior of functions induced by λ-terms. The three conditions express that (1) the semantic constants behave like the syntactic ones, (2) app is the left inverse of Im on valid functions (the semantic version of β) and (3) certain term-induced functions are valid. The motivation for (3) is the standard one in Henkin-style semantics: It ensures that the recursively defined semantic interpretation (Prop. 14) employs valid functions in the Lm-case.

With our available infrastructure, the formal statement and proof of the soundness theorem is easy:

Theorem 17. *Let* $(\mathbf{S}, \mathsf{ct}, \mathsf{app}, \mathsf{Im}, \mathsf{ValidFuns})$ *be an environment model and let* sem *be its corresponding interpretation function. Then* $X \equiv Y$ *implies* $\mathsf{sem}\ X = \mathsf{sem}\ Y$.

The theorem follows from the soundness of one-step reduction, i.e., the fact that $X \to Y$ implies $\mathsf{sem}\ X = \mathsf{sem}\ Y$. The proof of the latter goes by rule induction on the definition of $\to$ (Def. 15). The substitution lemma (built in our framework as the last-but-one point of Prop. 14) plays a key role when dealing with the (β) case. Here is the standard argument, cast in our framework: We must prove

$$\mathsf{sem}\,(\mathsf{App}\,(\mathsf{Lm}\ y\ X)\ Y)\rho = \mathsf{sem}\,(X\ [Y\ /\ y])\rho$$

To this end, we apply the Prop. 14 clauses for App, Lm and substitution, which reduces our goal to

$$\mathsf{app}\,(\mathsf{Im}\,(\lambda s.\,\mathsf{sem}\ X\ (\rho[y \leftarrow s])))\,(\mathsf{sem}\ Y\ \rho) = \mathsf{sem}\ X\ (\rho[y \leftarrow \mathsf{sem}\ Y\ \rho])$$

The last is true by points (2) and (3) of the environment model definition.

In conclusion, our framework's infrastructure facilitates the formalization of statements about the semantic interpretation of syntax.

3.3 The Church-Rosser theorem

A binary relation $\succ$ is called *confluent* provided it satisfies the following "diamond" property: For all u, v_1, v_2 such that $u \succ v_1$ and $u \succ v_2$, there exists w such that $v_1 \succ w$ and $v_2 \succ w$.

In other words, every span can be joined. The Church-Rosser theorem states that this is the case for multi-step reduction:

Theorem 18. *$\to^*$ is confluent.*

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

Dealing with multiple steps in the proof is possible, but the reasoning becomes intricate. A more elegant solution, due to William Tait, proceeds along the following lines [12]:

(1) First define a relation $\Rightarrow$ allowing the reduction of multiple (zero or more) redexes in parallel and prove that its transitive closure, $\Rightarrow^*$, is the same as $\to^*$.
(2) Then prove that $\Rightarrow$ is confluent—which should be possible thanks to parallelism. In the above example, we would have $Y_1 \Rightarrow Z$ by the parallel reduction of two Z-redexes.

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

Def 19. *The* one-step parallel reduction *relation* $\Rightarrow$: **term** $\to$ **term** $\to$ **bool** *is defined inductively by the following rules:*

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

$$\frac{X \Rightarrow X' \qquad Y \Rightarrow Y'}{\mathsf{App}\; X\; Y \Rightarrow \mathsf{App}\; X'\; Y'}\;(\mathsf{App}) \qquad \frac{X \text{ has the form } \mathsf{Var}\; x \text{ or } \mathsf{Ct}\; c}{X \Rightarrow X}\;(\mathsf{Refl})$$

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

The key technical differences between the definition of $\Rightarrow$ and that of $\to$ are the following. $\to$ has distinct left and right rules for application, (AppL) and (AppR), which (together with (ξ)) navigate towards the single redex to be targeted for reduction via the (β) rule, which forms the base case. By contrast, $\Rightarrow$ deals with the immediate subterms X and Y of terms App X Y in parallel, through two alternative routes:

– either by processing both subterms, via the (App) rule
– or, if the term happens to form a redex, optionally reducing that top redex *and* processing both subterms, via the (β) rule (which is no longer a base case)

In addition, $\Rightarrow$ has a reflexivity rule, (Refl), that allows dealing with the idle components of the term (those not affected by reduction). (Refl) only applies to variables and constants, but could have been allowed to apply to arbitrary terms, to the same effect:

Lemma 20 *$X \Rightarrow X$ holds for any term X.*

It is not difficult to prove (by standard rule induction, using Lemma 20) that $X \to Y$ implies $X \Rightarrow Y$ and that $X \Rightarrow Y$ implies $X \to^* Y$, which ensure that $\Rightarrow^* \;=\; \to^*$. This concludes task (1). Our formal proof required no special binding-aware type of reasoning, but only standard inductive definitions and rule-induction proofs.

Moving on to task (2), proving that $\Rightarrow$ is confluent, the simplest known approach is due to Takahashi [76]. Let us assume that $X \Rightarrow Y_1$ and $X \Rightarrow Y_2$, which means that both Y_1 and Y_2 have been obtained from X by the parallel reduction of a number of redexes—it is the choice of which redexes have been reduced and which have been ignored (via the (Refl) rule) that constitutes the difference between Y_1 and Y_2. Hence, if Z is the term obtained from X by a *complete* parallel reduction (with no redexes ignored)—which we write as $Z = \mathsf{cdev}\ X$—then Z would be a valid join for Y_1 and Y_2. Indeed, Z would be obtained from both Y_1 and Y_2 by reducing the redexes that had been ignored during the reductions of X to Y_1 and Y_2.

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

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

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

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

$$\mathsf{fresh}\ y\ X \ \text{ implies } \mathsf{fresh}\ y\ (\mathsf{cdev}\ X)$$

and commute with swapping:

$$\mathsf{cdev}\ (X[z_1 \leftrightharpoons z_2]) = (\mathsf{cdev}\ X)[z_1 \leftrightharpoons z_2].$$

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

Prop 21. *There exists a unique function* $\mathsf{cdev} : \textbf{term} \to \textbf{term}$ *satisfying all the above clauses (for the term constructors as well as the freshness and swapping operators).*

Indeed, rewriting these clauses to make the required structure on the target type explicit, we see that they simply state the commutation of cdev with the constructors and the operators as described in Prop. 7, where:

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

Verifying the FSw model clauses for the above is completely routine. (Again, the desired facts follow by Isabelle's "auto" proof method, which in this case applies the natural sim-

plification rules for term constructors and operators.) Note that Prop. 7 does not require the target type of the defined function to be a "syntactic" domain such as **term** (or to satisfy any finite-support property)—although this happens to be the case here. With the definition of cdev in place, it remains to prove the following:

Lemma 22 *$X \Rightarrow X'$ implies $X' \Rightarrow \mathsf{cdev}\ X$*

The informal proof of this lemma would go by induction on X, applying the Barendregt convention in the Lm-case, i.e., when X has the form $\mathsf{Lm}\ y\ Y$, to ensure that the bound variable y is fresh for X'. One might expect that the structural fresh induction principle (Prop. 3) is ideal for formalizing this task. However, the problem is that cdev analyzes X more than one-level deep—when testing if X is a β-redex, i.e., has the form $\mathsf{App}\ (\mathsf{Lm}\ x_1\ X_1)\ X_2$. This means that, in an inductive proof, we know that the fact holds for X_1 and X_2 and must prove that it holds for $\mathsf{App}\ (\mathsf{Lm}\ x_1\ X_1)\ X_2$—this goes one notch beyond structural induction. We therefore use induction on the depth of X, and take advantage of Barendregt's variable convention by means of the fresh case distinction principle (Prop. 4) instead.

3.4 The standardization theorem

The relation $\rightarrow$ makes a completely nondeterministic choice of the redex it reduces. The standardization theorem [67] refers to enforcing, without loss of expressiveness, a "standard" reduction strategy, which prioritizes leftmost redexes.

Def 23. *The* one-step left reduction *relation* $\not\rightarrow : \mathbf{term} \rightarrow \mathbf{term} \rightarrow \mathbf{bool}$ *is defined inductively by the following rules:*

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

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

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

Def 24. *The* standard reduction (s.r.) sequence *predicate* $\mathsf{srs} : \mathbf{term\ list} \rightarrow \mathbf{bool}$ *is defined inductively by the following rules:*

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

$$\frac{X \not\rightarrow \mathsf{hd}\ Xs \qquad \mathsf{srs}\ Xs}{\mathsf{srs}\ (X \cdot Xs)}\ (\mathsf{Red}) \qquad \frac{\mathsf{srs}\ Xs}{\mathsf{srs}\ (\mathsf{map}\ (\mathsf{Lm}\ x)\ Xs)}\ (\mathsf{Lm}) \qquad \frac{\mathsf{srs}\ Xs \qquad \mathsf{srs}\ Ys}{\mathsf{srs}\ (\mathsf{zipApp}\ Xs\ Ys)}\ (\mathsf{App})$$

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

A standard reduction sequence $[X_1,\ldots,X_n]$ represents a systematic way of performing reduction, prioritizing left reduction, but also eventually exploring rightward located redexes. Thus, the rule (App) merges two s.r. sequences under the App constructor, scheduling the

left one first and the right one second. The standardizaton theorem states that standard reduction sequences cover all possible reductions.

Theorem 25. *$X \to^* X'$ iff there exists a s.r. sequence starting in X and ending in X'.*

The "if" direction, stating that s.r. sequences are subsumed by arbitrary reduction sequences, follows immediately by rule induction on the definition of srs. So let us focus on the "only if" direction. It turns out that it is easier to use the multi-step parallel reduction $\Rightarrow^*$ instead of $\to^*$—which is OK since we know from Section 3.3 that they are equal. To have better control over $\Rightarrow$ (and over $\Rightarrow^*$), we need to be able to count the number of redexes that are being reduced in a step $X \Rightarrow Y$. In his informal proof, Plotkin defines this number by a recursive traversal of the derivation tree for $X \Rightarrow Y$. Since we defined the relation $\Rightarrow$ inductively, i.e., as a least fixed point, we do not have direct access to the derivation trees. Instead, we introduce this number in a labeled variation of $\Rightarrow$, defined inductively as follows:

Def 26. *The* labeled one-step parallel reduction *relation* $\Rightarrow\_$: **term** $\to$ **term** $\to$ **nat** $\to$ **bool** *is defined inductively by the following rules:*

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

$$\frac{X \Rightarrow_m X' \qquad Y \Rightarrow_n Y'}{\mathsf{App}\ X\ Y \Rightarrow_{m+n} \mathsf{App}\ X'\ Y'}\ (\mathsf{App}) \qquad \frac{X \text{ has the form } \mathsf{Var}\ x \text{ or } \mathsf{Ct}\ c}{X \Rightarrow_0 X}\ (\mathsf{Refl})$$

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

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

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

(We recall that $\mathrm{no}\ X'\ y$ counts the number of (free) occurrences of the variable y in X', via the operator no defined at the end of Section 2.1.)

Now, using an easy lemma stating that $X \Rightarrow Y$ is equivalent to the existence of n : **nat** such that $X \Rightarrow_n Y$, we are left with proving the following:

Prop 27. *If $X \Rightarrow^*_m X'$, then there exists a s.r. sequence starting in X and ending in X'.*

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

Prop 28. *If $X \Rightarrow_m X'$ and Xs is a s.r. sequence starting in X', then there exists a s.r. sequence starting in X and ending in the last term of Xs.*

Prop. 28 easily implies Prop. 27 by rule induction on the definition of the reflexive-transitive closure; in the base case, one uses the fact that $\mathsf{src}\ [X]$ holds for all terms X, which follows immediately by rule induction on the definition of of src.

So it remains to prove Prop. 28. The proof requires a quite elaborate induction, namely lexicographic induction on three measures: the length of Xs, the number (of X-to-X' reduc-

tion steps) m and the depth of X. Inside the induction proof, there is a case distinction on the form of X.

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

Lemma 29 *If $X \Rightarrow_m X'$ and $Y \Rightarrow_n Y'$, then there exists k such that $k \leq m + \mathsf{no}\ X'\ y * n$ and $X[Y/y] \Rightarrow_k X'[Y'/y]$.*

It is proved by induction on the depth of X, making essential use of the property that connects no with substitution, which is built in our definition of no (Def. 8). The second expresses commutation between (labeled) parallel reduction and left reduction:

Lemma 30 *If $X \Rightarrow_m Y$ and $Y \leftarrowtail Z$, then there exist Y' and n such that $X \leftarrowtail^* Y'$ and $Y' \Rightarrow_n Z$.*

It is proved by lexicographic induction on m and the depth of X. Back to the proof of Prop. 28, the other cases (different from App) are conceptually quite straightforward. However, the formal treatment of the Lm-case raises a subtle issue, which we describe next.

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

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

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

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

Lemma 31 *If $\mathsf{Lm}\ y\ Y \Rightarrow_m X'$, then one of the following holds:*
- *$X' = \mathsf{Lm}\ y\ Y$ (meaning (Refl) must have been applied)*
- *There exists Y' such that $X' = \mathsf{Lm}\ y\ Y'$ and $Y \Rightarrow_m Y'$ (meaning a y-synchronized (ξ) must have been applied)*

Proving the above is not straightforward, and relies on some properties of $\Rightarrow_m$ that are global, i.e., depend on the behavior of its rules different from (ξ). All we can get from the standard inversion rule (coming from the inductive definition of $\Rightarrow_m$) is, in the second case, the existence of z, Z and Z' such that $\mathsf{Lm}\ y\ Y = \mathsf{Lm}\ z\ Z$, $X' = \mathsf{Lm}\ z\ Z'$ and $Z \Rightarrow_m Z'$. Using the properties of equality between Lm-terms, we obtain that $Y = Z[y \leftrightharpoons z]$. To complete the proof of Lemma 31, we further need the following:

Lemma 32 $\Rightarrow\_$ *is equivariant, i.e., $Z \Rightarrow_m Z'$ implies $Z[y \leftrightharpoons z] \Rightarrow_m Z'[y \leftrightharpoons z]$.*

Lemma 33 $\Rightarrow\_$ *preserves freshness, i.e.,* $\mathsf{fresh}\ y\ Z$ *and $Z \Rightarrow_m Z'$ implies* $\mathsf{fresh}\ y\ Z'$.

Using these lemmas and the basic properties of freshness and swapping, we define Y' to be $Z'\,[y \leftrightharpoons z]$ and obtain $\mathsf{Lm}\ y\ Y' = \mathsf{Lm}\ z\ Z'$ and $Y \Rightarrow_m Y'$; in particular, $X' = \mathsf{Lm}\ y\ Y'$ and $Y \Rightarrow_m Y'$, as desired. This concludes our outline of the proof of Prop 28 and overall of the standardization theorem.

3.5 Adequate HOAS encoding

Next we describe another case study, which takes advantage of our framework's increased substitution-awareness: the formal definition and proof of an adequate HOAS encoding of CBN λ-calculus into itself. The technique we describe here would also apply to more complex encodings in logical frameworks.

HOAS encoding of syntax. A feature of our formalized syntax of λ-calculus is that the type **const** of constants is not fixed; rather, the type **term** is parameterized by an unspecified type **const**. It is formalized in Isabelle as a polymorphic type. This feature has not been very important so far, but becomes crucial for our HOAS application. We will use to instances of this polymorphic type:

- one as before, with constants from a type **const**, which we still denote by **term**, and
- one with constants from **const** $\cup \{\mathsf{ctapp}, \mathsf{ctlm}\}$ (i.e., **const** enriched with two new constants, ctapp and ctlm, corresponding to the term constructors App and Lm), which we denote by **term**$'$

Switching to standard λ-notation for a moment, the natural HOAS encoding of **term** in **term**$'$ should be a characterized by the following equations:

(1) $\mathsf{enc}\ x = x$
(2) $\mathsf{enc}\ c = c$
(3) $\mathsf{enc}\,(X\ Y) = \mathsf{ctapp}\ (\mathsf{enc}\ X)\,(\mathsf{enc}\ Y)$
(4) $\mathsf{enc}\,(\lambda x.X) = \mathsf{ctlm}\ (\lambda x.\ \mathsf{enc}\ X)$

In our formalization, these equations are:

(1) $\mathsf{enc}\,(\mathsf{Var}\ x) = \mathsf{Var}\ x$
(2) $\mathsf{enc}\ (\mathsf{Ct}\ c) = \mathsf{Ct}\ c$
(3) $\mathsf{enc}\,(\mathsf{App}\ X\ Y) = \mathsf{App}\,(\mathsf{App}\ \mathsf{ctapp}\,(\mathsf{enc}\ X))\,(\mathsf{enc}\ Y)$
(4) $\mathsf{enc}\,(\mathsf{Lm}\ x\ X) = \mathsf{App}\ \mathsf{ctlm}\ (\mathsf{Lm}\ x\ (\mathsf{enc}\ X))$

Two central properties of HOAS encodings are preservation of freshness and commutation with substitution, the latter usually called *compositionality* [40, 61]—here is their statement for our case:

(5) $\mathsf{fresh}\ x\ X$ implies $\mathsf{fresh}\ x\ (\mathsf{enc}\ X)$
(6) $\mathsf{enc}\,(X[Y\,/\,y]) = (\mathsf{enc}\ X)[(\mathsf{enc}\ Y)\,/\,y]$

As usual, the problem with the equations (1)–(4) is that they are not guaranteed to be valid on alpha-equated terms. Our framework again offers an immediate resolution via Prop. 7: In exchange for some trivial term properties to check, it provides a function enc satisfying not only (1)–(4), but also (5) and (6).

Def 34. $\mathsf{enc} : \textbf{term} \to \textbf{term}'$ *is the unique function satisfying clauses (1)–(6).*

In fact, here we have an example where Prop. 10 applies too, offering us two additional facts about enc (again in return for the verification of some trivial properties of terms):

(7) enc is injective
(8) The "iff" version of clause (5) holds

Clauses (6) and (7) form what is usually called the *(syntactic) adequacy* property of a HOAS encoding.[10] One could also argue that (8), which is seldom stated explicitly in the HOAS literature, should be verified as well in order to deem an encoding adequate. Our framework's recursion principle seems almost specialized in delivering such adequacy "packages."

Here are the aforementioned basic properties that we have been required to check in order for Prop. 7 and 10 to apply, guaranteeing the above properties of enc. The clauses (1)–(6) indicate the following FSb model structure having carrier type **term**′. The constructor-like functions are:

- Var
- Ct
- the function mapping X, X', Y, Y' to $\mathsf{App}\,(\mathsf{App}\ \mathsf{ctapp}\ X)\ Y$
- the function mapping x, X and X' to $\mathsf{App}\ \mathsf{ctlm}\ (\mathsf{Lm}\ x\ (\mathsf{enc}\ X))$

Note that the last two functions above ignore the "primed" arguments (members of **term**); this is because only iteration is needed here (rather than full-fledged recursion). The freshness- and substitution-like operators are the usual fresh and _[_/_], again ignoring the primed arguments.

The fact that the above forms an FSb model amounts to the following:

F1: $\mathsf{fresh}\ x\ (\mathsf{Ct}\ c)$
F2: $x \neq z$ implies $\mathsf{fresh}\ z\ (\mathsf{Var}\ x)$
F3: $\mathsf{fresh}\ z\ X$ and $\mathsf{fresh}\ z\ Y$ implies $\mathsf{fresh}\ z\ (\mathsf{App}\ (\mathsf{App}\ \mathsf{ctapp}\ X)\ Y)$
F4: $\mathsf{fresh}\ x\ (\mathsf{App}\ \mathsf{ctlm}\ (\mathsf{Lm}\ x\ X))$
F5: $\mathsf{fresh}\ z\ X$ implies $\mathsf{fresh}\ z\ (\mathsf{App}\ \mathsf{ctlm}\ (\mathsf{Lm}\ x\ (\mathsf{enc}\ X)))$
Sb1: $(\mathsf{Var}\ z)[Z/z] = Z$
Sb2: $x \neq z$ implies $(\mathsf{Var}\ x)[Z/z] = \mathsf{Var}\ x$
Sb3: $(\mathsf{App}\ (\mathsf{App}\ \mathsf{ctapp} X)\ Y)\,[Z'/z] = \mathsf{App}\ (\mathsf{App}\ \mathsf{ctapp}\,(X[Z'/z]))\ (Y[Z'/z])$
Sb4: $x \neq z$ and $\mathsf{fresh}\ x\ Z$ implies $(\mathsf{App}\ \mathsf{ctlm}\ (\mathsf{Lm}\ x\ X))\,[Z/z] = \mathsf{App}\ \mathsf{ctlm}\ (\mathsf{Lm}\ x\ (X[Z/z]))$
SbRn: $x \neq y$ and $\mathsf{fresh}\ y\ X$ implies $\mathsf{App}\ \mathsf{ctlm}\ (\mathsf{Lm}\ y\ (X[(\mathsf{Var}\ y)/x])) = \mathsf{App}\ \mathsf{ctlm}\ (\mathsf{Lm}\ x\ X)$

The fact that the model is freshness-reversing amounts to the following:

F2c: $\mathsf{fresh}\ z\ (\mathsf{Var}\ x)$ implies $x \neq z$
F3c: $\mathsf{fresh}\ z\ (\mathsf{App}\,(\mathsf{App}\ \mathsf{ctapp}\ X)\ Y)$ implies $\mathsf{fresh}\ z\ X$ and $\mathsf{fresh}\ z\ Y$
F4_5c: $\mathsf{fresh}\ z\ (\mathsf{App}\ \mathsf{ctlm}\ (\mathsf{Lm}\ x\ X))$ implies $x = z$ or $\mathsf{fresh}\ z\ X$

The fact that the model is constructor-injective amounts to the aforementioned constructor-like functions being injective and non-overlapping.

All the above follow immediately (and are proved in Isabelle automatically) from the standard properties of substitution and freshness—commutation with the term constructors, our framework stores as proved lemmas. For example, facts F1–F5 and their converses follow from the standard simplification facts for freshness w.r.t. the term constructors, and SbRn follows from Prop. 1(2).

HOAS encoding of the reduction relation. So far, we have used the **term**′ syntax to adequately encode the **term** syntax. In order to be able to encode inductively defined relations on **term**, we will need to organize **term**′ as miniature logical framework. Unlike in full-fledged logical frameworks such as Edinburgh LF [40] or Generic Isabelle [58], it will not have its own built-in mechanism for specifying logics or calculi—instead, we will use the

[10] In typed frameworks, the adequacy property additionally ensures that the encoding is a bijective correspondence between the terms of the original system and some canonical forms in the host system.

"external" mechanism of inductive definitions of relations over **term**′. The background term equivalence will be β-equivalence, $\equiv$.

With these provisions, we can encode inductively defined n-ary relations R on **term** as inductively defined n-ary relations R_{h} on **term**′, where:

- Each inductive clause in the definition of R is matched by an inductive clause in the definition of R_{h}.
- There is an additional "background" clause in the definition of R_{h} that states compatibility with β-equivalence.

All the relations on **term** defined in this paper can be encoded in this manner. We choose $\twoheadrightarrow$ as an example, which will be encoded as a relation $\twoheadrightarrow_{\mathsf{h}}$.

Def 35. *The relation* $\twoheadrightarrow_h : \mathbf{term}' \to \mathbf{term}' \to \mathbf{bool}$ *is defined inductively by the following rules:*

$$\frac{}{\mathsf{App}\ (\mathsf{App}\ \mathsf{ctapp}\ (\mathsf{App}\ \mathsf{ctlm}\ X))\ Y \twoheadrightarrow_h X\ Y}\quad (\beta\text{'}) \qquad \frac{\mathsf{Ctapp}\ c_1\ c_2\ =\ \mathsf{Some}\ X}{\mathsf{App}\ c_1\ c_2 \twoheadrightarrow_h X}\quad (\delta\text{'})$$

$$\frac{X \twoheadrightarrow_h X'}{\mathsf{App}\ (\mathsf{App}\ \ \mathsf{ctapp}\ X)\ Y \twoheadrightarrow_h \mathsf{App}\ (\mathsf{App}\ \ \mathsf{ctapp}\ X')\ Y}\quad (\mathsf{AppL'})$$

$$\frac{X \text{ has the form } \mathsf{Var}\ x \text{ or } \mathsf{Ct}\ c \qquad Y \twoheadrightarrow_h Y'}{\mathsf{App}\ (\mathsf{App}\ \ \mathsf{ctapp}\ X)\ Y \twoheadrightarrow_h \mathsf{App}\ (\mathsf{App}\ \ \mathsf{ctapp}\ X)\ Y'}\quad (\mathsf{AppR'})$$

$$\frac{X \equiv Y \qquad Y \twoheadrightarrow_h Y' \qquad Y' \equiv X'}{X \twoheadrightarrow_h X'}\quad (\mathsf{Compat}_{\equiv})$$

The difference between the above clauses for $\twoheadrightarrow_{\mathsf{h}}$ and the corresponding ones that define $\twoheadrightarrow$ (in Def. 23) is that now Lm and App are employed as part of the meta-level infrastructure, whereas the object-level behavior of the application and abstraction constructors is tagged with the constants ctapp and ctlm. The object-calculus substitution in rule (β) is replaced by mere meta-level application in rule (β'). The background rule $(\mathsf{Compat}_{\equiv})$ is responsible for "fixing" this mismatch between (β) and (β'): The meta-level application of encoded items will be part of a β-redex, which is β-equivalent to a meta-level term obtained by applying meta-level substitution. This means that, ultimately, the object-level substitution in (β) will correspond to meta-level substitution.

Let us illustrate the above phenomenon, switching for a moment to standard λ-calculus notation. In this notation, the (β) rule for $\twoheadrightarrow$ is $(\lambda y.\ X)\,Y \twoheadrightarrow X[Y/x]$, and the (β') rule for $\twoheadrightarrow_{\mathsf{h}}$ is $\mathsf{ctlm}\ X\ Y \twoheadrightarrow_{\mathsf{h}} X\ Y$. An instance of (β) is $(\lambda x.\ x)\,y \twoheadrightarrow x[y/x]$, i.e., $(\lambda x.\ x)\,y \twoheadrightarrow y$. The corresponding instance of (β') is $\mathsf{ctapp}\ (\mathsf{ctlm}\ (\lambda x.\ x))\ y \twoheadrightarrow_{\mathsf{h}} (\lambda x.\ x)\ y$. The two instances are related as follows:

- $\mathsf{enc}\,((\lambda x.\ x)\,y) = \mathsf{ctapp}\ (\mathsf{ctlm}\ (\lambda x.\ x))\ y$, i.e., the lefthand side of the second is the encoding of the lefthand side of the first
- $\mathsf{enc}\,y = y \equiv (\lambda x.\ x)\ y$, i.e., the righthand side of the second is β-equivalent to the encoding of the lefthand side of the first

This suggests a statement of the adequacy of the encoding of $\twoheadrightarrow$ as $\twoheadrightarrow_{\mathsf{h}}$.

Theorem 36. *The following hold:*

(1) *If* $X \twoheadrightarrow Y$ *then* $\mathsf{enc}\ X \twoheadrightarrow_h \mathsf{enc}\ Y$.
(2) *If* $\mathsf{enc}\ X \equiv X'$ *and* $X' \twoheadrightarrow_h Y'$, *then there exists* Y *such that* $X \twoheadrightarrow Y$ *and* $\mathsf{enc}\ Y \equiv Y'$.
(3) $X \twoheadrightarrow Y$ *iff* $\mathsf{enc}\ X \twoheadrightarrow_h \mathsf{enc}\ Y$.

Point (1) follows by rule induction on the definition of $\twoheadrightarrow$. All cases are completely routine, except for that of the (β) rule. In that case (using again standard λ-calculus notation for

readability), we must prove $\mathsf{enc}\,((\lambda y.\ X)\ Y) \not\mapsto \mathsf{enc}\,(X[Y/y])$. We have the following, using (β') and the properties of enc, including compositionality:

$$\mathsf{enc}\,((\lambda y.\ X)\ Y) = \mathsf{ctapp}\ (\mathsf{ctlm}\ (\lambda y.\ \mathsf{enc}\ X))\ (\mathsf{enc}\ Y) \not\mapsto_{\mathsf{h}} (\lambda y.\ \mathsf{enc}\ X)\ (\mathsf{enc}\ Y) \equiv$$
$$\equiv (\mathsf{enc}\ X)[(\mathsf{enc}\ Y)\,/\,y] = \mathsf{enc}\,(X[Y/y])$$

From this, using $(\mathsf{Compat}_{\equiv})$ we obtain $\mathsf{enc}\,((\lambda y.\ X)\ Y) \not\mapsto_{\mathsf{h}} \mathsf{enc}\,(X[Y/y])$, as desired.

Point (2) follows by rule induction on the definition of as $\not\mapsto_{\mathsf{h}}$, using some inversion rules of $\equiv$ w.r.t. the syntactic constructors. Point (3) has one implication covered by point (1). For the other implication, we use point (2) and the following simple but crucial observation:

Lemma 37 *$\mathsf{enc}\ X$ is a β-normal form (in that, for all Y, $\mathsf{enc}\ X \to^* Y$ implies $Y = \mathsf{enc}\ X$).*

This ensures that $\mathsf{enc}\ X \equiv \mathsf{enc}\ Y$ implies $\mathsf{enc}\ X = \mathsf{enc}\ Y$, which further implies $X = Y$ (by the injectivity of enc). In turn, this immediately allows to prove (3)'s reverse implication from point (2).

This concludes our formal exercise of deploying our framework for adequately encoding both syntax and reduction of CBN λ-calculus in a miniature HOAS framework. In the future, it will be interesting to explore the formalization of more complex frameworks using the same techniques.

4 Call-By-Value λ-Calculus

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

Def 38. *The* one-step CBV reduction *relation* $\to_{\mathsf{v}} : \mathbf{term} \to \mathbf{term} \to \mathbf{bool}$ *is defined inductively by rules similar to those of Def. 15, namely by the rules (AppL) and (AppR) from there (of course, with $\to_{\mathsf{v}}$ replacing $\to$), together with:*

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

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

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

Def 39. *The* one-step parallel CBV reduction *relation* $\Rightarrow_{\mathsf{v}} : \mathbf{term} \to \mathbf{term} \to \mathbf{bool}$ *is defined inductively by rules similar to those of Def. 19, namely by the rules (App) and (Refl) from there (with $\Rightarrow_{\mathsf{v}}$ replacing $\Rightarrow$), together with:*

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

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

Def 40. *The* one-step left CBV reduction *relation* $\not\to_v$: **term** $\to$ **term** $\to$ **term** *is defined inductively by rules similar to those of Def. 23, namely by the rule (AppL) from there (with* $\not\to_v$ *replacing* $\not\to$*), together with:*

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

$$\frac{Y \not\to_v Y'}{\mathsf{App}\ (\mathsf{Val}\ V)\ Y \not\to_v \mathsf{App}\ (\mathsf{Val}\ V)\ Y'}(\mathsf{AppR})$$

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

Moreover, the statements and proofs of the Church-Rosser and standardization theorems are essentially identical, *mutatis mutandis*. Like Plotkin has suggested in his informal development [67], the formal proofs could be easy adapted from CBN to CBV, obtaining:

Theorem 41. *Theorem 18 and Theorem 25 hold with the same statements, after replacing the CBN notions with their CBV counterparts.*

While the CBN and CBV formal developments are conceptually very similar, for the latter we employed our framework's infrastructure for a two-sorted syntax. To illustrate how this two-sorted syntax is handled by the framework, we show the definition of the CBV counterpart of cdev. (We omit the sort annotation, **term** or **value**, form the substitution and swapping operators.)

Def 42. *The* CBV complete parallel reduction operator *of a term X (written* $\mathsf{cdev}_{\mathbf{term}}\ X$*) and of a value V (written* $\mathsf{cdev}_{\mathbf{value}}\ V$*) are the unique pair of functions satisfying:*

$\mathsf{cdev}_{\mathbf{value}}\ (\mathsf{Var}\ x) = \mathsf{Var}\ x \qquad \mathsf{cdev}_{\mathbf{value}}\ (\mathsf{Ct}\ c) = \mathsf{Ct}\ c$

$\mathsf{cdev}_{\mathbf{term}}\ (\mathsf{Val}\ V) = \mathsf{Val}\ (\mathsf{cdev}_{\mathbf{value}}\ V) \qquad \mathsf{cdev}_{\mathbf{value}}\ (\mathsf{Lm}\ y\ X) = \mathsf{Lm}\ y\ (\mathsf{cdev}_{\mathbf{term}}\ X)$

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

$\mathsf{fresh}_{\mathbf{value}}\ y\ V\ \ \textit{implies}\ \ \mathsf{fresh}_{\mathbf{value}}\ y\ (\mathsf{cdev}_{\mathbf{value}}\ V)$

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

$\mathsf{cdev}_{\mathbf{value}}\ (V[z_1 \leftrightharpoons z_2]) = (\mathsf{cdev}_{\mathbf{value}}\ V)[z_1 \leftrightharpoons z_2]$

$\mathsf{cdev}_{\mathbf{term}}\ (X[z_1 \leftrightharpoons z_2]) = (\mathsf{cdev}_{\mathbf{term}}\ X)[z_1 \leftrightharpoons z_2]$

Similarly to the CBN case, this turns out to be a correct definition thanks to a two-sorted version of Prop. 7, that is, via exhibiting a two-sorted FSw model.

5 Overview of the Formalization

As already mentioned, our development is based on a general theory of syntax with bindings, which was presented elsewhere [37]. The development has two parts.

The first part is the instantiation of the general theory to the two syntaxes, of λ-calculus and of λ-calculus with emphasized values, together with the transfer from a deep to a more shallow embedding (reported in Section 2). This is currently a completely routine, but very tedious process—it spans over more than 15000 lines of code (LOC) for each syntax. The reasons for this large size are the sheer number of stated theorems about constructors and substitution (more than 300 facts for the one-sorted syntax and more than 500 for the two-sorted syntax) and the many intermediate facts stated in the process of transferring the recursion theorems. Thanks to using a custom template for the instantiation, the whole process only took us two person-days. However, this is unreasonably long for a process that can be entirely automated—so we leave its automation as a pressing goal for future work.

The second part is the theory of CBN and CBV λ-calculus, culminating with the proofs of the soundness, Church-Rosser, standardization and HOAS adequacy theorems (reported in Sections 3 and 4). This is where our routine effort from the first part fully paid off. Thanks to our comprehensive collection of facts about substitution and freshness, we were able to focus almost entirely on formalizing the high-level ideas present in the informal proofs—notably in Plotkin's sketches of his elaborate proof development for the standardization theorem. On two occasions—for defining Takahashi's complete parallel reduction operator and the number of variable occurrences needed in the Standardization proof development—our recursion principle allowed us to quickly get off the ground, in the second case also offering useful freshness and substitution lemmas needed later in the proof. Altogether, the second part consists of 5500 LOC (2500 for CBN and 3000 for CBV) and took us one person-month. The appendix gives concrete pointers to the Isabelle formalization, including a map of the theorems listed in this paper and their formal counterparts.

An exception to the above general phenomenon (of being able to focus on the high-level proof ideas) was the need to engage in the low-level task of proving custom constructor-directed inversion rules for our reduction relations—illustrated and motivated in the discussion leading to Lemma 31. This lemma is just one example of the several similar inversion rules we proved, corresponding to the inductive rules involving λ-abstraction in the reduction relations' definitions. These rules are essentially the binding-aware version of what Isabelle/HOL offers via the "inductive cases" command [83]. They seem to be generally useful in proof developments that involve inductively defined reductions but require induction over terms. Binding-aware inversion principles form an integral part of higher-order abstract syntax frameworks [11, 63, 64, 73], and have also been discussed in a nominal logic context [15], but unfortunately have never been implemented in Isabelle Nominal.

Finally, our case study illustrates another interesting and apparently not uncommon phenomenon: that fresh structural induction on terms may be too weak in proofs, whereas depth-based induction in conjunction with fresh cases may do the job *while still enabling the use of Barendregt's convention*—as illustrated in our proof of Lemma 22.

6 Related Work

This paper's contribution is twofold: (1) it instantiates our general framework to two particular syntaxes, showing how to deploy the framework's induction and recursion principles and (2) it performs two specific formal reasoning case studies for these syntaxes. We split the discussion of related work in two corresponding subsections.

6.1 Formal approaches to syntax with bindings

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], [28, §6] and [37, §8] for overviews.) These approaches roughly fall under three main paradigms of reasoning about bindings. In the *nameful paradigm*, binding variables are passed as arguments to the binding operator and terms are usually equated modulo alpha-equivalence. 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 [33, 34], nominal logic was subsequently cast in a standard foundation [65, 66], and also significantly developed in a proof assistant context—most extensively by Urban and collaborators [77–79, 81, 82].

In the *nameless paradigm* originating with De Bruijn [21], the bindings are indicated through nameless pointers to positions in a term. Major exponents of the scope-safe nameless paradigm are representations based on presheaves [31, 43] and nested datatypes [7, 16]. The presheaf approach has been generalized and refined in many subsequent works, e.g., [4–6, 30, 35, 42, 47].

Finally, the *higher-order abstract syntax (HOAS)* paradigm, [24, 27–29, 40, 58, 59, 63] based on ideas going back as far as Church [25], Huet and Lang [46] and Martin-Löf [56, Chapter 3], has gained traction with the works of Harper et. al [40], Pfenning and Elliott [62] and Paulson [58] 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. HOAS has been pursued in dedicated logical frameworks such as Abella [11], Beluga [64], Delphin [73] and Twelf [63], and in general-purpose proof assistants such as Coq [24, 27] and Isabelle [39]. HOAS often allows for lighter formalizations, thanks to borrowing binding mechanisms and sometimes structural properties from the meta-level. Formalizations in this paradigm are often accompanied by pen-and-paper proofs of the representations' adequacy (which involve informal reasoning about substitution) [40, 61]; as shown in Section 3.5, our substitution-aware recursion principle can ease the formalization of such proofs. Some approaches in the literature combine two paradigms. For example, the locally nameless approach [10, 22, 68] 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 [28] and the "HOAS on top of FOAS" approach [72], which develop HOAS reasoning techniques over locally nameless and nameful representation substrata.

Our work in this paper belongs to the nameful paradigm, giving a formal expression to many ideas from nominal logic—but departing from nominal logic through its focus on a rich built-in theory of substitution (including substitution-aware recursion) and built-in semantic interpretation. While our structural induction principle (Prop. 3) is essentially the same as the nominal logic one (as implemented in Coq [9] and Isabelle [82]), our recursion principles (Prop. 7) differ from the nominal logic one in two essential ways: First, our FSw-model-based principle, while factoring in freshness and swapping as primitives on the target domain like the nominal one, does *not* assume that the former is defined from the latter—this brings additional generality and has similarities to a principle formalized by Michael Norrish in HOL4 for the syntax of λ-calculus [57]. Second, our FSb-model-based principle factors in substitution rather than swapping, which is arguably a more fundamental operator to syntax with bindings (notwithstanding the nominal logic's convincing case for the fundamental role of swapping). A current limitation of our recursion principles is their inability to handle freshness for parameters. In particular, this means that we could not have used, say, our FSw-model-based principle to define substitution on (quotiented) terms. Instead, our framework

performs a low-level definition of substitution on (unquotiented) quasi-terms and then lifts it to terms. All these details are of course hidden from the user.

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]. In the context of nominal logic, defining semantic interpretations incurs some difficulties due to the absence of finite support [66, page 492].

Another difference between our approach and that of a definitional package such as Nominal Isabelle is that we statically verify the arbitrary-syntax meta-theory whereas they dynamically generate any instance of interest. For a more through discussion of the distinguishing features of our general framework, including universe versus code-generator approaches, we refer the reader to [37].

6.2 Similar case studies in other frameworks

In a development that has become part of the Isabelle standard library, Nipkow and Berghofer [14, 55] have proved several CBN λ-calculus properties, including Church-Rosser and Normalization. They use a de Bruijn encoding of λ-terms, which somewhat impairs the readability of their statements and proofs. The Isabelle Nominal package hosted many developments concerning (variants of) λ-calculus [2], including the CBN Church-Rosser and standardization [8,54], the second fixed point theorem [48] and the meta-theory of Edinburgh's LF [80].

The Church-Rosser and standardization theorems have also been formalized in other provers: the Church-Rosser theorem in Abella [3], Coq [45], HOL [44], LEGO [50], PVS [75] and Twelf [60] and the standardization theorem in Coq [26] and LEGO [13, 51]. All of the above developments consider the call-by-name variant of λ-calculus (or of a more complex calculus)—which means our work provides the first formalization of these results for the call-by-value calculus. However, the call-by-value calculus has been formalized in other contexts, e.g., recently as a model of computation in Coq [32].

Aspects of our framework's approach to semantic interpretation and HOAS encodings have already presented in the second author's PhD thesis [70, §2.3] and in a previous conference paper [71], but are not developed as thoroughly as in this paper; in particular, this paper additionally covers environment models and the soundness of β-reduction, as well as a more principled approach to adequacy. The only other formalization of HOAS adequacy we are aware of is that of Cheney et al. [23] using Nominal Isabelle, which covers a more complex case than ours: that of encoding λ-calculus in HOL. Admittedly, Nominal Isabelle already delivers well for the task of defining HOAS encodings and proving their adequacy. Yet, our framework seems able to target HOAS phenomena even more hands-on: It offers the syntactic adequacy properties (including substitution compositionality and freshness preservation and reflection) as part of the recursion infrastructure, which leads to a very compact formulation and proof of adequacy.

Apart from the novelty of some of the formalized results (e.g., concerning call-by-value), a main motivation for performing these case studies is that they offered us the possibility to test essentially all our framework's features, from built-in substitution to induction and recursion principles to semantic interpretation to many-sortedness. We believe that these features have enabled us to produce a fully formal yet pedagogical presentation of the results.

In the future, it would be interesting to provide a comparison between our development and alternative developments in other frameworks.

Acknowledgments. 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), `https://www.seas.upenn.edu/~plclub/poplmark/`
2. The Nominal Methods group (2018), https://nms.kcl.ac.uk/christian.urban/Nominal/
3. Accattoli, B.: Proof pearl: Abella formalization of λ-calculus cube property. In: Hawblitzel, C., Miller, D. (eds.) Certified Programs and Proofs. pp. 173–187. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
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. Arnaud, M., Berghofer, S., Narboux, J., Nipkow, T., Urban, C.: Properties of Lambda-calculus using isabelle nominal (2018), `https://isabelle.in.tum.de/dist/library/HOL/HOL-Nominal-Examples/index.html`
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. van Benthem Jutting, L.S., McKinna, J., Pollack, R.: Checking algorithms for pure type systems. In: TYPES. pp. 19–61 (1993)
14. Berghofer, S., Nipkow, T.: Fundamental properties of lambda-calculus (2017), `https://isabelle.in.tum.de/library/HOL/HOL-Proofs-Lambda`
15. Berghofer, S., Urban, C.: Nominal inversion principles. In: TPHOLs. pp. 71–85 (2008)
16. Bird, R.S., Paterson, R.: De Bruijn notation as a nested datatype. J. Funct. Program. 9(1)
17. Blanchette, J.C., Popescu, A.: Mechanizing the metatheory of Sledgehammer. In: FroCoS. pp. 245–260 (2013)
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., Popescu, A., Traytel, D.: Unified classical logic completeness—A coinductive pearl. In: IJCAR 2014. pp. 46–60 (2014)
20. Blanchette, J.C., Popescu, A., Traytel, D.: Soundness and completeness proofs by coinductive methods. J. Autom. Reasoning 58(1), 149–179 (2017)
21. 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)
22. Charguéraud, A.: The locally nameless representation. J. Autom. Reasoning 49(3), 363–408 (2012)
23. Cheney, J., Norrish, M., Vestergaard, R.: Formalizing adequacy: A case study for higher-order abstract syntax. J. Autom. Reasoning 49(2), 209–239 (2012)
24. Chlipala, A.J.: Parametric higher-order abstract syntax for mechanized semantics. In: ICFP. pp. 143–156 (2008)
25. Church, A.: A formulation of the simple theory of types. J. Symb. Logic 5(2), 56–68 (1940)
26. Coquand, C.: Combinator shared reduction and infinite objects in type theory (04 1996)
27. Despeyroux, J., Felty, A.P., Hirschowitz, A.: Higher-order abstract syntax in Coq. In: TLCA. pp. 124–138 (1995)
28. 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)
29. Felty, A.P., Pientka, B.: Reasoning with higher-order abstract syntax and contexts: A comparison. In: ITP. pp. 227–242 (2010)

30. 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)
31. Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding (extended abstract). In: LICS. pp. 193–202 (1999)
32. Forster, Y., Smolka, G.: Weak call-by-value lambda calculus as a model of computation in coq. In: Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasilia, Brazil, September 26-29, 2017 (Apr 2017)
33. Gabbay, M., Pitts, A.M.: A new approach to abstract syntax involving binders. In: LICS. pp. 214–224 (1999)
34. Gabbay, M., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Asp. Comput. 13(3-5), 341–363 (2002)
35. Gambino, N., Hyland, M.: Wellfounded trees and dependent polynomial functors. In: TYPES, pp. 210–225 (2003)
36. Gheri, L., Popescu, A.: This paper's homepage. `http://andreipopescu.uk/papers/LambdaCaseStudies.html`
37. Gheri, L., Popescu, A.: A formalized general theory of syntax with bindings: Extended version. Journal of Automated Reasoning pp. 1–35 (2019), published online first at `http://andreipopescu.uk/pdf/theoryOfBindings.pdf`
38. Gheri, L., Popescu, A.: A general theory of syntax with bindings. Archive of Formal Proofs (2019), `http://isa-afp.org/entries/Binding_Syntax_Theory.html`, Formal proof development
39. 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)
40. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. In: LICS. pp. 194–204 (1987)
41. Hindley, J.R., Longo, G.: Lambda calculus models and extensionality. Z. Math. Logik Grundlag Math. 29, 289–310 (1980)
42. Hirschowitz, A., Maggesi, M.: Modules over monads and initial semantics. Inf. Comput. 208(5), 545–564 (2010)
43. Hofmann, M.: Semantical analysis of higher-order abstract syntax. In: LICS. p. 204 (1999)
44. Homeier, P.: A proof of the Church-Rosser theorem for the lambda calculus in higher order logic. In: TPHOLs 2001: Supplemental Proceedings. pp. 207–222 (2001)
45. Huet, G.: Residual theory in lambda-calculus : a formal development. Research Report RR-2009, INRIA (1993), `https://hal.inria.fr/inria-00074663`
46. Huet, G.P., Lang, B.: Proving and applying program transformations expressed with second-order patterns. Acta Inf. 11, 31–55 (1978), `https://doi.org/10.1007/BF00264598`
47. Kaiser, J., Schäfer, S., Stark, K.: Binder aware recursion over well-scoped De Bruijn syntax. In: CPP. pp. 293–306 (2018)
48. Kaliszyk, C., Barendregt, H.: Reasoning about constants in nominal isabelle, or how to formalize the second fixed point theorem. In: CPP. pp. 87–102 (2011)
49. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales—a sectioning concept for Isabelle. In: TPHOLs. pp. 149–166 (1999)
50. McKinna, J., Pollack, R.: Pure type systems formalized. In: TLCA (1993)
51. McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. Journal of Automated Reasoning 23(3), 373–409 (Nov 1999)
52. Meyer, A.R.: What is a model of the lambda calculus? Information and Control 52(1), 87–122 (1982)
53. Mitchell, J.C.: Foundations for Programming Languages. MIT Press (1996)
54. Nagele, J., van Oostrom, V., Sternagel, C.: A short mechanized proof of the Church-Rosser theorem by the Z-property for the $\lambda\beta$-calculus in Nominal Isabelle. CoRR abs/1609.03139 (2016)
55. Nipkow, T.: More church-rosser proofs (in isabelle/hol). In: CADE. pp. 733–747 (1996)
56. Nordström, B., Petersson, K., Smith, J.M.: Programming in Martin-Löf's Type Theory: An Introduction. Oxford University Press (1990)
57. Norrish, M.: Recursive function definition for types with binders. In: TPHOLs. pp. 241–256 (2004)
58. Paulson, L.C.: The foundation of a generic theorem prover. J. Autom. Reason. 5(3) (1989)
59. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: PLDI. pp. 199–208 (1988)
60. Pfenning, F.: A proof of the Church-Rosser theorem and its representation in a Logical Framework. Tech. rep., Pittsburgh, PA, USA (1992)
61. Pfenning, F.: Computation and Deduction (2001)
62. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: PLDI. pp. 199–208 (1988)
63. Pfenning, F., Schürmann, C.: System description: Twelf - A meta-logical framework for deductive systems. In: CADE. pp. 202–206 (1999)
64. Pientka, B.: Beluga: Programming with dependent types, contextual data, and contexts. In: FLOPS. pp. 1–12 (2010)

65. Pitts, A.M.: Nominal logic: A first order theory of names and binding. In: TACS. pp. 219–242 (2001)
66. Pitts, A.M.: Alpha-structural recursion and induction. J. ACM 53(3) (2006)
67. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. 1(2), 125–159 (1975)
68. Pollack, R.: Closure under alpha-conversion. In: TYPES. pp. 313–332 (1993)
69. Pollack, R., Sato, M., Ricciotti, W.: A canonical locally named representation of binding. J. Autom. Reasoning 49(2), 185–207 (2012)
70. 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`
71. Popescu, A., Gunter, E.L.: Recursion principles for syntax with bindings and substitution. In: ICFP. pp. 346–358 (2011)
72. 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)
73. Poswolsky, A., Schürmann, C.: System description: Delphin—A functional programming language for deductive systems. Electr. Notes Theor. Comput. Sci. 228, 113–120 (2009)
74. Schropp, A., Popescu, A.: Nonfree datatypes in Isabelle/HOL – animating a many-sorted metatheory. In: CPP. pp. 114–130 (2013)
75. Shankar, N.: A mechanical proof of the Church-Rosser theorem. J. ACM 35(3), 475–522 (1988)
76. Takahashi, M.: Parallel reductions in lambda-calculus. Inf. Comput. 118(1), 120–127 (1995)
77. Urban, C.: Nominal techniques in Isabelle/HOL. J. Autom. Reason. 40(4), 327–356 (2008)
78. Urban, C., Berghofer, S.: A recursion combinator for nominal datatypes implemented in Isabelle/HOL. In: IJCAR. pp. 498–512 (2006)
79. Urban, C., Berghofer, S., Norrish, M.: Barendregt's variable convention in rule inductions. In: CADE. pp. 35–50 (2007)
80. Urban, C., Cheney, J., Berghofer, S.: Mechanizing the metatheory of lf. In: LICS 2008. pp. 45–56 (2008)
81. Urban, C., Kaliszyk, C.: General bindings and alpha-equivalence in Nominal Isabelle. In: ESOP. pp. 480–500 (2011)
82. Urban, C., Tasson, C.: Nominal techniques in Isabelle/HOL. In: CADE. pp. 38–53 (2005)
83. Wenzel, M.: The Isabelle/Isar reference manual (2018), available at `http://isabelle.in.tum.de/doc/isar-ref.pdf`
84. Wenzel, M.: The Isabelle system manual (2019), `https://isabelle.in.tum.de/doc/system.pdf`

APPENDIX

The Isabelle theories can be downloaded from the paper's website [36] and processed with Isabelle 2019. The general framework (applicable to an arbitrary syntax with bindings and reported in our companion paper [37]) is an entry in the Archive of Formal Proofs [38] and must be imported from there. Our development is based on that entry and is structured in three sessions (provided with their customary ROOT files [84, §2]): Interface, Instance_Lambda_Syntax and Case_Studies

The Interface session

This session pre-instantiates the general framework to several commonly encountered arities. The development is also syntax-independent, and can be regarded as being part of the general framework.

The Instance_Lambda_Syntax session

This session fully instantiate the framework to the two particular syntaxes discussed in this paper: the single-sorted (unsorted) one of λ-calculus (used for the CBN calculus) and the two-sorted variation that distinguishes values from other terms (used for the CBV calculus). It corresponds to Section 2. The relevant theories in this session are called L, L_Inter, LV and LV_Inter.

The theory L contains a wealth of facts that are made available for the (unsorted) syntax of λ-calculus after instantiating our framework (discussed in Section 2.1). The theory file contains detailed comments to guide the reader through these facts. They cover properties of the constructors and the operators (freshness, swapping, unary substitution and parallel substitution), as well as induction and recursion and semantic-interpretation principles. The theory LV has a similar structure and content (though fewer comments), but considers the two-sorted syntax of λ-calculus with emphasized values (discussed in Section 2.2).

The theories L_Inter and LV_Inter further customize the two syntax instances with a few abbreviations and re-formulations of facts that we have deemed more convenient for this particularly simple syntaxes. Notably, they introduce the Lm constructor, which in L_Inter has type **var** $\to$ **term** $\to$ **term** by putting together an abstraction constructor Abs : **var** $\to$ **term** $\to$ **abs** and a one-binding-argument constructor, Lam : **abs** $\to$ **term**. More precisely, Lm x X abbreviates Lam (Abs x X). (Our general framework employs explicit abstractions as a separate syntactic category, whereas here we preferred to inline abstractions as part of a single Lm-constructor.)

Here is a map between this Section 2.1's propositions and their formal counterparts in theory L:[11]

- Prop. 1 corresponds to lemmas "Lam inj" and "Abs_lm_lm swap_vlm_lm ex"
- Prop. 2 corresponds to lemmas "subst_vlm_lm compose 1" and "subst_vlm_lm subst_vlm_lm compose 2"
- Prop. 3 corresponds to lemma "induct fresh" (reformulated as lemma "induct fresh 2" in theory L_Inter)
- Prop. 4 corresponds to lemma "term_lm fresh cases" (reformulated as lemma "term fresh cases" in theory L_Inter)

[11] Note that the paper covers only a small subset of the facts provided in the formalization. The latter are best explored by reading the content of theory L, which includes detailed comments and explanations.

- Prop. 7 corresponds to lemmas "wlsFSb rec term_FSb_morph" and "wlsFSw rec term_FSw_morph"
- Prop. 10 corresponds to lemmas "wlsFSb rec refl_freshAll" and "wlsFSb rec is_injAll"
- Prop. 14 corresponds to lemma "wlsSEM semInt comp_int"

The Case_Studies session

This session contains the four case studies described in Sections 3.2–3.5 and Section 4. The relevant theories of this session are:

- CBN, Henkin CBN_CR, CBN_Std and HOAS for the CBN calculus
- CBV, CBV_CR, and CBV_Std for the CBV calculus

The theory CBN defines Section 3's various reduction relations and proves basic facts about them, including fresh rule induction and fresh inversion principles. The other mentioned theories have self-explanatory names:

- Henkin handles the soundness theorem for Henkin-style models (Section 3.2)
- CBN_CR handles the Church-Rosser theorem (Section 3.3)
- CBN_Std handles the standardization theorem (Section 3.4)
- HOAS handles the HOAS development (Section 3.5)

These theories also define the following recursive functions presented in this paper. In all cases, the end-product formal facts are obtained after expanding the definition of FSb or FSw model morphism.

- Section 2.1's number of free occurrences operator, no, using substitution-aware recursion—Def. 8 corresponds to CBN's lemmas no_simps, no_subst and no_fresh.
- Section 3.3's complete development operator, cdev, using swapping-aware recursion—Prop. 21 corresponds to theory CBN_CR's lemmas "cdev_simps 1", cdev_App_isDred, cdev_App_isLm, cdev_App_not_isDred_isLm and cdev_swap and cdev_fresh.
- Section 3.5's HOAS encoding operator enc—Def. 34 corresponds to theory HOAS's lemmas enc_simps, enc_subst and enc_fresh.

Finally, here is the mapping between main theorems presented in this paper's Section 3 and their formal counterparts:

- The Church-Rosser Theorem 18 corresponds to theory CBN_CR's theorem Mredn_confluent
- The standardization Theorem 25 corresponds to theory CBN_Std's theorem standardization
- The syntactic adequacy theorem represented by clauses (6)–(8) in Def. 34 correspond to HOAS's lemmas enc_subst, enc_fresh and enc_inj.
- The reduction adequacy Theorem 36 corresponds to theory HOAS's theorems enc_preserves_rednL, enc_reflects_MrednL and rednL_enc_MrednL.