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

A Case Study in Reasoning about Syntax with Bindings: The Church-Rosser and Standardization Theorems

Lorenzo Gheri · **Andrei Popescu**

Received: date / Accepted: date

Abstract We have recently 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 two main results: the Church-Rosser and Standardization theorems. Our work covers both the call-by-name and call-by-value versions of the calculus, following a classic paper by Gordon 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.

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, textbooks on λ-calculi employ the principle of primitive recursion to define functions on λ-terms, after which they tacitly assume these functions to be invariant under alpha-equivalence; as another example, the so-called Barendregt variable convention assumes that, in a proof or definition context, the bound variables are fresh for all the parameters located outside the scope of their binders. Both these principles are unsound in general, that is, if employed without checking some sanity conditions on the defining clauses or on the definition and proof context.

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

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 [20,26,29,35] 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 [7,8,11] or by representing them using meta-variables as in Higher-Order Abstract Syntax (HOAS) [10,15].

Our own framework [14] 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.

This paper presents an extensive case study of reasoning about bindings using our framework, namely the formalization of two major λ-calculus results, the Church-Rosser and Standardization theorems for β-reduction, in both the call-by-name and call-value variants.[1]

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 (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 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 a range of reduction relations, including parallel and left, single-step and multi-step reductions. The Church-Rosser theorem (Section 3.2) is proved by formalizing the parallel-reduction technique of Tait [5], enhanced with the complete parallel reduction operator trick due to Takahashi [34]. For Standardization (Section 3.3), we follow closely Plotkin's original paper [28].

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 [14] is parameterized by a binding signature, which essentially specifies the following data: a collection of term sorts, a collection of variable sorts, an embedding relationship between variable sorts and term sorts, and a collection of (term) constructors, each with an assigned arity and an assigned result sorts. The arity consists of zero or more input sorts—where an input sort is either just a term sort (for free inputs) or a pair of a variable sort and a term sort (for bound inputs). The result sort refers to the output of the constructor.

The theory was developed over an arbitrary signature (which is represented as an Isabelle locale [19]). Namely, "quasi-terms" were defined as being freely generated by the construc-

[1] Our formalization is publicly available from the paper's website [13].

tors, 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 paper [14] 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 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 instantiating our signature to consist of:

- a single variable sort, vs, and a single term sort, ts—assuming that vs is embedded in ts gives as the variable-constructor Var, which embeds vs-variables into ts-terms
- the (non-variable) constructors App, Lam, and Ct c for each c : **const**, such that:
  - each Ct c has empty arity and result sort ts, which we can write briefly[2] as $([], \mathsf{ts})$
  - App has its arity consisting of two occurrences of free ts and result sort ts, which we can write as $([\mathsf{ts}, \mathsf{ts}], \mathsf{ts})$
  - Lm has its arity consisting of one vs-bound ts and result sort ts, which we can write as $([(\mathsf{vs}, \mathsf{ts})], \mathsf{ts})$

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

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

To address this, we have designed a systematic method for transferring a signature instance to the expected shallow embedding of the instance. This involves creating native Isabelle/HOL types of terms for each sort of the signature and transferring all the term constructors and operators and all facts about them to these native types. The process is conceptually straightforward, but is quite tedious, and must be done by hand since we have not yet automated it. (But [32] presents the automation of a similar kind of transfer.)

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

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

- the constructors, namely $\mathsf{Var} : \mathbf{var} \to \mathbf{term}$, $\mathsf{Ct} : \mathbf{const} \to \mathbf{term}$, $\mathsf{App} : \mathbf{term} \to \mathbf{term} \to \mathbf{term}$ and $\mathsf{Lm} : \mathbf{var} \to \mathbf{term} \to \mathbf{term}$
- and the standard operators:
  - depth (height) of a term, $\mathsf{depth} : \mathbf{term} \to \mathbf{nat}$
  - freshness of a variable in a term,[3] $\mathsf{fresh} : \mathbf{var} \to \mathbf{term} \to \mathbf{bool}$
  - (capture-free) substitution of a term for a variable in a term, $\_[\_/\_] : \mathbf{term} \to \mathbf{term} \to \mathbf{var} \to \mathbf{term}$
  - (capture-free) parallel substitution of multiple terms for multiple variables in a term, $\_[\_] : \mathbf{term} \to (\mathbf{var} \to \mathbf{term\ option}) \to \mathbf{term}$
  - swapping of two variables in a term,[4] $\_[\_ \wedge \_] : \mathbf{term} \to \mathbf{var} \to \mathbf{var} \to \mathbf{term}$

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 [27, 37, 40]. 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:

Prop 3. (Fresh structural induction principle) *Let* **param** *be a type (of items called parameters) endowed with a function* $\mathsf{varsOf} : \mathbf{param} \to \mathbf{var\ set}$ *such that* $\mathsf{varsOf}\ p$ *is finite for all* $p : \mathbf{param}$. *Let* $\varphi : \mathbf{term} \to \mathbf{param} \to \mathbf{bool}$ *be a predicate on terms and parameters.*

Assume the following four sentences are true for all $x : \mathbf{var}$, $c : \mathbf{const}$ *and* $X, Y : \mathbf{term}$*:*
(1) $\varphi\ (\mathsf{Var}\ x)$ *p holds for all* $p : \mathbf{param}$.
(2) $\varphi\ (\mathsf{Ct}\ c)$ *p holds for all* $p : \mathbf{param}$.

[3] Other frameworks employ a free-variable operator, $\mathsf{FVars} : \mathbf{term} \to \mathbf{term\ set}$. This is of course interdefinable with the freshness operator.

[4] While not explicitly present in the traditional λ-calculus [5], 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 [27].

(3) If $\varphi\ X\ p$ *and* $\varphi\ Y\ p$ *hold for all* p : **param**, *then* $\varphi\,(\mathsf{App}\ X\ Y)\ q$ *holds for all* q : **param**.
(4) If $\varphi\ X\ p$ *holds for all* p : **param**, *then* $\varphi\,(\mathsf{Lm}\ x\ X)\ q$ *holds for all* q : **param** *such that* $x \notin \mathsf{varsOf}\ q$.

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

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

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) $Z = \mathsf{Var}\ x$ *for some* x : **var**.
(2) $Z = \mathsf{Ct}\ c$ *for some* c : **const**.
(3) $Z = \mathsf{App}\ X\ Y$ *for some* X, Y : **term**.
(4) $Z = \mathsf{Lm}\ x\ X$ *for some* x : **var** *and* X : **term** *such that* $x \notin \mathsf{varsOf}\ p$.

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

Operator-aware recursion. Our framework offers structural recursion principles 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* $\mathsf{VAR} : \mathbf{var} \to \mathbf{D}$, $\mathsf{CT} : \mathbf{const} \to \mathbf{D}$, $\mathsf{APP} : \mathbf{term} \to \mathbf{D} \to \mathbf{term} \to \mathbf{D} \to \mathbf{D}$ *and* $\mathsf{Lm} : \mathbf{var} \to \mathbf{term} \to \mathbf{D} \to \mathbf{D}$
– *functions on* **D** *having similar types as the freshness and substitution operators (again, with* **term** *suitably replaced with* **D** *or with* **term** *and* **D***), namely* $\mathsf{FRESH} : \mathbf{var} \to \mathbf{term} \to \mathbf{D} \to \mathbf{bool}$ *and* $\mathsf{SUBST} : \mathbf{term} \to \mathbf{D} \to \mathbf{term} \to \mathbf{D} \to \mathbf{var} \to \mathbf{D}$

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

The framework's recursion principles essentially say that terms form the initial FSb and FSw models:[5]

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, $H\ (X[z_1 \wedge z_2]) = \mathsf{SWAP}\ X\ (H\ X)\ z_1\ z_2$*)*

The principle is much easier to use in practice than its elaborate formulation might suggest: Say one wishes to define a function H from **term** to a type **D**. Then the functions on **D** corresponding to the term constructors are determined from the desired recursive clauses for H. Moreover, the functions on **D** corresponding to freshness and substitution or swapping are determined by the desired behavior of H with respect to these operators, obtained from answering questions such as "How can $H\ (X[Z\ /\ x])$ be expressed in terms of $H\ X$, $H\ Z$ and x?". This methodology is explained in [31] and [14] and will be further illustrated in this paper.

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 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

[5] 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 [30, §1.4.2] for a detailed discussion of this distinction.

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:

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

where Val is the injection of values into terms.

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

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

Applying the same systematic deep-to-shallow transfer process as for the previous one-sorted syntax, we obtain "native" types **val** and **term** for values and terms, the expected constructors (e.g., Val : **val** $\to$ **term**) and the standard operators, one for either syntactic category (e.g., $\mathsf{fresh}_{\mathbf{val}}$: **var** $\to$ **val** $\to$ **bool** and $\mathsf{fresh}_{\mathbf{term}}$: **var** $\to$ **term** $\to$ **bool**). The framework-provided induction and recursion principles now refer to these mutually recursive types: Induction allows us to prove two simultaneous predicates and recursion allows us two define two simultaneous functions, one on values and one on terms—this will be illustrated in Section 4.

3 Call-By-Name λ-Calculus

In this section, we show how we have used our framework's infrastructure to formalize two landmark results in the theory of call-by-name (CBN) λ-calculus: the Church-Rosser theorem [5], stating that the order in which call-by-name redexes are reduced is irrelevant "in the long run" and the standardization theorem [28], stating that reducibility is not restricted if we impose a canonical reduction strategy, based on identifying left-most redexes.

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

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

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

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

Def 8. *The* one-step (CBN) reduction *relation* $\to$: **term** $\to$ **term** $\to$ **bool** *is defined inductively by the following rules:*

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

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

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

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

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

3.2 The Church-Rosser theorem

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

Theorem 9. $\to^*$ *is confluent.*

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

Dealing with multiple steps in the proof is possible, but the reasoning becomes intricate. A more elegant solution, due to William Tait, proceeds along the following lines [5]:

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

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

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

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

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

It is not difficult to prove (by standard rule induction) that $X \rightarrow Y$ implies $X \Rightarrow Y$ and that $X \Rightarrow Y$ implies $X \rightarrow^* Y$, which ensure that $\Rightarrow^* = \rightarrow^*$. 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 [34]. Let us assume that $X \Rightarrow Y_1$ and $X \Rightarrow Y_2$, which means that both Y_1 and Y_2 have been obtained from X by the parallel reduction of a number of redexes—it is the choice of which redexes have been reduced and which have been ignored (via the (Refl) rule) that constitutes the difference between Y_1 and Y_2. Hence, if Z is the term obtained from X by a *complete* parallel reduction (with no redexes ignored)—which we write as $Z = \mathsf{cpred}\ X$—then Z would be a valid join for Y_1 and Y_2. Indeed, Z would be obtained from both Y_1 and Y_2 by reducing the redexes that had been ignored during the reductions of X to Y_1 and Y_2.

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

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

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

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

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

Prop 11. *There exists a unique function* $\mathsf{cpred} : \mathbf{term} \rightarrow \mathbf{term}$ *satisfying all the above clauses (for the term constructors as well as the freshness and swapping operators).*

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

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

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

– $\mathsf{FRESH}\ x\ X'\ X\ =\ \mathsf{fresh}\ x\ X$

– $\mathsf{SWAP}\ X'\ X\ z_1\ z_2\ =\ X[z_1\ \wedge\ z_2]$

Verifying the FSw model clauses for the above is completely routine (follows by Isabelle's "auto" proof method, which applies the natural simplification rules for term constructors and operators). Note that Prop. 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 cpred in place, it remains to prove the following:

Lemma 12 *$X \Rightarrow X'$ implies $X' \Rightarrow \mathsf{cpred}\ X$*

The informal proof of this lemma would go by induction on X, applying the Barendregt convention in the Lm-case, i.e., when X has the form $\mathsf{Lm}\ y\ Y$, to ensure that the bound variable y is fresh for X'. One might expect that the structural fresh induction principle (Prop. 3) is ideal for formalizing this task. However, the problem is that cpred analyzes X more than one-level deep—when testing if X is a β-redex, i.e., has the form $\mathsf{App}\ (\mathsf{Lm}\ x_1\ X_1)\ X_2$. This means that we need to go a bit beyond structural induction. We therefore use induction on the depth of X,[6] and take advantage of the Barendregt convention by means of the fresh case distinction principle (Prop. 4) instead.

3.3 The standardization theorem

The relation $\rightarrow$ makes a completely nondeterministic choice of the redex it reduces. The standardization theorem [28] refers to enforcing, without loss of expressiveness, a "standard" reduction strategy, which prioritizes leftmost redexes.

Def 13. *The* one-step left reduction *relation* $\not\rightarrow$: **term** $\rightarrow$ **term** $\rightarrow$ **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 14. *The* standard reduction (s.r.) sequence *predicate* srs : **term list** $\rightarrow$ **bool** *is defined inductively by the following rules:*

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

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

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

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

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

Theorem 15. *$X \to^* X'$ 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 3.2 that they are equal. To have better control over $\Rightarrow$ (and over $\Rightarrow^*$), we need to be able to count the number of redexes that are being reduced in a step $X \Rightarrow Y$. In his informal proof, Plotkin defines this number by a recursive traversal of the derivation tree for $X \Rightarrow Y$. Since we defined the relation $\Rightarrow$ inductively, i.e., as a least fixed point, we do not have direct access to the derivation trees. Instead, we introduce this number in a labeled variation of $\Rightarrow$, defined inductively as follows:

Def 16. *The* labeled one-step parallel reduction *relation* $\Rightarrow\_ : \mathbf{term} \to \mathbf{term} \to \mathbf{nat} \to \mathbf{bool}$ *is defined inductively by the following rules:*

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

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

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

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

- 1 for the top redex (which is being explicitly reduced in the rule)
- m for the redexes being reduced in X to obtain X'
- $n*\mathrm{no}\ X'\ y$ 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'

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

Def 17. $\mathsf{no} : \mathbf{term} \to (\mathbf{var} \to \mathbf{nat})$ *is the unique function satisfying the following properties:*

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

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

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

To justify the above definition, we construct an FSb model obtained from the above clauses, where $\mathbf{D} = \mathbf{var} \to \mathbf{nat}$ and $\mathsf{SUBST} : \mathbf{term} \to \mathbf{D} \to \mathbf{term} \to \mathbf{D} \to \mathbf{var} \to \mathbf{D}$ is

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

Verifying Prop. 7's conditions is again routine—some simple arithmetics that again has been discharged by the "auto" proof method. Note again how we included not only the recursive clauses for the constructors, but also those for the interaction with freshness and substitution. On the one hand, the freshness and substitution clauses are needed to establish the correctness of the definition; on the other hand, they are useful theorems that are produced (and proved) at definition time together with the recursive clauses for the constructors.

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

Prop 18. *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 19. *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. 19 easily implies Prop. 18 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. 19. The proof requires a quite elaborate induction, namely lexicographic induction on three measures: the length of Xs, the number (of X-to-X' reduction steps) m and the depth of X. Inside the induction proof, there is a case distinction on the form of X.

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

Lemma 20 *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. 17). The second expresses commutation between (labeled) parallel reduction and left reduction:

Lemma 21 *If $X \Rightarrow_m Y$ and $Y \rightarrowtail Z$, then there exist Y' and n such that $X \rightarrowtail^* Y'$ and $Y' \Rightarrow_n Z$.*

It is proved by lexicographic induction on m and the depth of X. Back to the proof of Prop. 19, 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 22 *If* $\mathsf{Lm}\ y\ Y \Rightarrow_m X'$*, then one of the following holds:*

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

Proving the above is not straightforward, and relies on some properties of $\Rightarrow_m$ that are global, i.e., depend on the behavior of its rules different from (ξ). All we can get from the standard inversion rule (coming from the inductive definition of $\Rightarrow_m$) is, in the second case, the existence of z, Z and Z' such that $\mathsf{Lm}\ y\ Y = \mathsf{Lm}\ z\ Z$, $X' = \mathsf{Lm}\ z\ Z'$ and $Z \Rightarrow_m Z'$. Using the properties of equality between Lm-terms, we obtain that $Y = Z\,[y \wedge z]$. To complete the proof of Lemma 22, we further need the following:

Lemma 23 $\Rightarrow\_$ *is equivariant, i.e.,* $Z \Rightarrow_m Z'$ *implies* $Z\,[y \wedge z] \Rightarrow_m Z'\,[y \wedge z]$.

Lemma 24 $\Rightarrow\_$ *preserves freshness, i.e.,* $\mathsf{fresh}\ y\ Z$ *and* $Z \Rightarrow_m Z'$ *implies* $\mathsf{fresh}\ y\ Z'$.

Using these lemmas and the basic properties of freshness and swapping, we define Y' to be $Z'\,[y \wedge z]$ and obtain $\mathsf{Lm}\ y\ Y' = \mathsf{Lm}\ z\ Z'$ and $Y \Rightarrow_m Y'$; in particular, $X' = \mathsf{Lm}\ y\ Y'$ and $Y \Rightarrow_m Y'$, as desired. This concludes our outline of the proof of Prop 19 and overall of the standardization theorem.

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 25. *The* one-step CBV reduction *relation* $\rightarrow_v$: **term** $\rightarrow$ **term** $\rightarrow$ **bool** *is defined inductively by rules similar to those of Def. 8, namely by the rules (AppL) and (AppR) from there (of course, with* $\rightarrow_v$ *replacing* $\rightarrow$*), together with:*

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

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

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 26. *The* one-step parallel CBV reduction *relation* $\Rightarrow_v$: **term** $\to$ **term** $\to$ **bool** *is defined inductively by rules similar to those of Def. 10, namely by the rules (App) and (Refl) from there (with* $\Rightarrow_v$ *replacing* $\Rightarrow$*), together with:*

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

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

Def 27. *The* one-step left CBV reduction *relation* $\rightarrow\!\!\!\!\cdot_v$: **term** $\to$ **term** $\to$ **term** *is defined inductively by rules similar to those of Def. 13, namely by the rule (AppL) from there (with* $\rightarrow\!\!\!\!\cdot_v$ *replacing* $\rightarrow\!\!\!\!\cdot$*), together with:*

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

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

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

Moreover, the statements and proofs of the Church-Rosser and standardization theorems are essentially identical, *mutatis mutandis*. Like Plotkin has suggested in his informal development [28], the formal proofs could be easy adapted from CBN to CBV, obtaining:

Theorem 28. *Theorem 9 and Theorem 15 hold with the same statements, after replacing the CBN notions with their CBV counterparts.*

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

Def 29. *The* CBV complete parallel reduction operator *of a term X (written* $\mathsf{cpred}_{\mathbf{term}}\ X$*) and of a value V (written* $\mathsf{cpred}_{\mathbf{val}}\ V$*) are the unique pair of functions satisfying:*

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

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

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

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

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

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

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

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 and Lessons Learned

Our development is based on a general theory of syntax with bindings, which was presented elsewhere [14]. 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 Church-Rosser and Standardization 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 5000 LOC (2000 for CBN and 3000 for CBV) and took us one person-month.

An exception to the above general phenomenon (of being able to focus on the high-level proof ideas) was the need to engage in the low-level task of proving custom constructor-

directed inversion rules for our reduction relations—illustrated and motivated in the discussion leading to Lemma 22. 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 [41]. They seems to be generally useful in proof developments that involve inductively defined reductions but require structural induction over terms. The literature on formal reasoning seems to have overlooked the general usefulness of these rules; and state of the art definitional packages such as Isabelle Nominal do not attempt to infer them automatically.

Finally, our case study illustrates another interesting and apparently not uncommon phenomenon: that fresh structural induction on terms may be too weak in proofs, whereas depth-based induction in conjunction with fresh cases may do the job *while still enabling the use of Barendregt's convention*—as illustrated in our proof of Lemma 12.

6 Conclusions and Related Work

In a development that has become part of the Isabelle standard library, Nipkow and Berghofer [3, 23] 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 [36, 39] is a popular alternative to de Bruijn techniques. It hosted many developments concerning (variants of) λ-calculus [1], including the CBN Church-Rosser and Standardization [3, 22], the second fixed point theorem [18] and the meta-theory of Edinburgh's LF [38].

The Church-Rosser and Standardization theorems have also been formalized in other provers: the Church-Rosser theorem in Abella [2], Coq [17], HOL [16], LEGO [20], PVS [33] and Twelf [25] and the Standardization theorem in Coq [9] and LEGO [6, 21]. All of the above developments consider the CBN variant of λ-calculus (or of a more complex calculus)—which means our work provides the first formalization of these results for the CBV calculus. However, the CBV calculus has been formalized in other contexts, e.g., recently as a model of computation in Coq [12].

Apart from the novelty concerning CBV, a main motivation for performing our own formalization of these fundamental theorems is that they represent good benchmarks for our framework—and indeed they offered us the possibility to test essentially all our framework's features, from built-in substitution to induction and recursion principles to many-sortedness. While our structural induction principle (Prop. 3) is essentially the same as the nominal logic one (as implemented in Coq [4] and Isabelle [40]), our recursion principles (Prop. 7) differ from the nominal logic one in two essential ways: First, our FSw-model-based principle factors in freshness and swapping as primitives on the target domain, without assuming that the former is defined from the latter—this has similarities to a principle that has been formalized by Michael Norrish in HOL4 for the syntax of λ-calculus [24]. 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. For a more through discussion of the distinguishing features of our general framework, we refer the reader to [14].

We believe the features of our framework (notably a rich built-in theory of substitution and operator-aware recursion) have enabled us to produce a fully formal yet pedagogical presentation of these fundamental results, which highlighted some new ideas concerning the relationship between informal and formal reasoning about bindings.

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 Nominal Methods group (2018), https://nms.kcl.ac.uk/christian.urban/Nominal/
2. Accattoli, B.: Proof pearl: Abella formalization of λ-calculus cube property. In: CPP. pp. 173–187 (2012)
3. 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`
4. Aydemir, B.E., Bohannon, A., Weirich, S.: Nominal reasoning techniques in coq: (extended abstract). Electr. Notes Theor. Comput. Sci. 174(5), 69–77 (2007)
5. Barendregt, H.P.: The Lambda Calculus. North-Holland (1984)
6. van Benthem Jutting, L.S., McKinna, J., Pollack, R.: Checking algorithms for pure type systems. In: TYPES. pp. 19–61 (1993)
7. 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)
8. Charguéraud, A.: The locally nameless representation. J. Autom. Reasoning 49(3), 363–408 (2012)
9. Coquand, C.: Combinator shared reduction and infinite objects in type theory (04 1996)
10. Felty, A.P., Pientka, B.: Reasoning with higher-order abstract syntax and contexts: A comparison. In: ITP. pp. 227–242 (2010)
11. Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding (extended abstract). In: LICS 1999. pp. 193–202 (1999)
12. 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)
13. Gheri, L., Popescu, A.: This paper's homepage. `http://andreipopescu.uk/papers/Lambda_CR_Std.html`
14. Gheri, L., Popescu, A.: A formalized general theory of syntax with bindings. In: ITP. pp. 241–261 (2017), Extended version submitted to the Journal of Automated Reasoning – draft available at `http://andreipopescu.uk/pdf/theoryOfBindings.pdf`
15. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. In: LICS 1987. pp. 194–204. IEEE, Computer Society Press (1987)
16. 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)
17. Huet, G.: Residual theory in lambda-calculus : a formal development. Research Report RR-2009, INRIA (1993), `https://hal.inria.fr/inria-00074663`
18. 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)
19. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales—a sectioning concept for Isabelle. In: TPHOLs. pp. 149–166 (1999)
20. McKinna, J., Pollack, R.: Pure type systems formalized. In: TLCA (1993)
21. McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. Journal of Automated Reasoning 23(3), 373–409 (Nov 1999)
22. 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)
23. Nipkow, T.: More church-rosser proofs (in isabelle/hol). In: CADE. pp. 733–747 (1996)
24. Norrish, M.: Recursive function definition for types with binders. In: TPHOLs. pp. 241–256 (2004)
25. Pfenning, F.: A proof of the Church-Rosser theorem and its representation in a Logical Framework. Tech. rep., Pittsburgh, PA, USA (1992)
26. Pitts, A.M.: Nominal logic: A first order theory of names and binding. In: TACS. pp. 219–242 (2001)
27. Pitts, A.M.: Alpha-structural recursion and induction. J. ACM 53(3) (2006)
28. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. 1(2), 125–159 (1975)

29. Pollack, R., Sato, M., Ricciotti, W.: A canonical locally named representation of binding. J. Autom. Reasoning 49(2), 185–207 (2012)
30. Popescu, A.: Contributions to the theory of syntax with bindings and to process algebra, phD thesis, Univ. of Illinois, 2010. Available at `andreipopescu.uk/thesis.pdf`
31. Popescu, A., Gunter, E.L.: Recursion principles for syntax with bindings and substitution. In: ICFP. pp. 346–358 (2011)
32. Schropp, A., Popescu, A.: Nonfree datatypes in isabelle/hol - animating a many-sorted metatheory. In: CPP. pp. 114–130 (2013)
33. Shankar, N.: A mechanical proof of the Church-Rosser theorem. J. ACM 35(3), 475–522 (1988)
34. Takahashi, M.: Parallel reductions in lambda-calculus. Inf. Comput. 118(1), 120–127 (1995)
35. Urban, C.: Nominal techniques in Isabelle/HOL. J. Autom. Reason. 40(4), 327–356 (2008)
36. Urban, C.: Nominal techniques in Isabelle/HOL. J. Autom. Reasoning 40(4), 327–356 (2008)
37. Urban, C., Berghofer, S., Norrish, M.: Barendregt's variable convention in rule inductions. In: CADE. pp. 35–50 (2007)
38. Urban, C., Cheney, J., Berghofer, S.: Mechanizing the metatheory of lf. In: LICS 2008. pp. 45–56 (2008)
39. Urban, C., Kaliszyk, C.: General bindings and alpha-equivalence in Nominal Isabelle. In: ESOP. pp. 480–500 (2011)
40. Urban, C., Tasson, C.: Nominal techniques in Isabelle/HOL. In: CADE. pp. 38–53 (2005)
41. Wenzel, M.: The Isabelle/Isar reference manual (2018), available at `http://isabelle.in.tum.de/doc/isar-ref.pdf`