Recursion principles for syntax with bindings and substitution

Andrei Popescu
Technical University Munich
University of Illinois at Urbana-Champaign
Institute of Mathematics Simion Stoilow of the Romanian Academy
uuomul@yahoo.com

Elsa L. Gunter
University of Illinois at Urbana-Champaign
egunter@uiuc.edu

Abstract

We characterize the data type of terms with bindings, freshness and substitution, as an initial model in a suitable Horn theory. This characterization yields a convenient recursive definition principle, which we have formalized in Isabelle/HOL and employed in a series of case studies taken from the λ-calculus literature.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

General Terms Languages, Verification

Keywords Syntax with bindings, Recursion, Lambda calculus, Substitution

1. Introduction

Bindings and substitution, i.e., the notions of *binding a variable in a term* and *substituting a term or value for a variable in a term*, respectively, are pervasive in logic and programming languages. Formal/rigorous reasoning about syntax with bindings is notoriously hard, as observed by many authors (e.g., [8, 11, 14, 19, 22, 28, 32, 39, 45]). In particular, basic lemmas about *composing and commuting substitutions* turn out, in practice, to be problematic.

In this paper, we introduce a recursor for terms with bindings with *built-in substitution*, which ensures, in exchange for checking a few conditions on the target domain, both the correctness of the definition and compositionality w.r.t. freshness and substitution. Our contribution is formalized in the Isabelle theorem prover [26], and is in general guided by formalization goals.

Apart from this introduction and Section 8 which draws conclusions and discusses related work, this paper has two main parts. The first part, consisting of Sections 2, 3, 4, and 5, motivates, presents and illustrates our mathematical contribution. In Section 2, we discuss two standard situations occurring in formal reasoning: defining a semantic domain interpretation map and a Higher-Order Abstract Syntax (HOAS) representation map. In order to go through, these definitions traditionally require a tedious treatment of bindings and α-equivalence. Moreover, later development usually requires that the involved interpretation/representation maps be shown to *preserve freshness* and be *compositional w.r.t. to substitution*. These "basic", "low-level" facts require serious effort from the developer, which contrasts with their highly intuitive character; and this discrepancy is of course even more dramatic if one is interested in a formal development (in a theorem prover). All the above motivate our approach, described in Section 3. The main idea is to integrate freshness and substitution in the definition (of the desired map from terms to another set), resolving "at definition time", both the issue of well-definedness (i.e., compatibility with α-equivalence) and that of substitution compositionality. Mathematically, this is based on a characterization of the term model as initial in a suitable Horn theory in the first-order language of the term-constructor, freshness and substitution operators. We show how our approach handles the aforementioned two situations – e.g., for HOAS, both the definition of the map and its so-called *adequacy* are established during the definition. Our thesis is that many situations from formal reasoning about syntax with bindings fall under the scope of this substitution-based definition principle. In Section 4, we bring evidence for this thesis by means of other examples taken from the λ-calculus literature. We also discuss cases where our principle does not apply. Finally, in Section 5, we explore the notion of inferring more facts about the maps defined by our methodology. It turns out that this is indeed possible (and, as witnessed by examples, useful). In particular, we obtain an internal characterization of the term model, expressed solely in terms of the basic operators: syntactic constructs, freshness and substitution.

The second part, consisting of Sections 6 and 7, deals with formalization aspects. Indeed, the main goal of our theoretical contribution is facilitating rigorous, and indeed *formal*, definitions of substitution-compositional maps on terms with bindings. Therefore, formalizing our results and applying them in actual formal developments is an important test for their relevance. In Section 6, we discuss the formal resolution in Isabelle/HOL [26] of one of our motivating problems, namely, adequate HOAS representation. In Section 7, we employ our characterization from Section 5 to relate by isomorphisms our formalization with other Isabelle formalizations of λ-calculus terms: Nominal [42], Hybrid [8] and *locally named* [38]. Indeed, our characterizations of terms as an abstract data type (ADT) is up to isomorphism – any model satisfying our properties yields an isomorphism to our own model (and to any correct model). We propose this methodology (of establishing such ADT isomorphisms) as a basis for certifying formalizations and transporting results across different formalizations.

The appendix contains more details on the Isabelle formalization of this paper's results and examples (made available at [4] for the interested reader). An extended version of the material presented in this paper, including technical details and explanations, can be found in [36] (as the largely self-contained chapter 2 from there).

Conventions and notations. We employ the standard symbols for functional and logical connectives and quantifiers (λ for functional

ICFP'11, September 19–21, 2011, Tokyo, Japan.

abstraction, $\wedge$ for conjunction, $\Longrightarrow$ for implication, $\forall$ for the universal quantifier), only in the meta-language of this paper, and *not* in the various formal languages that we discuss.

We fix a countable set **var** of variables, ranged over by x, y, z, and consider the set **term**, of terms, ranged over by X, Y, Z, to be generated by the following grammar:

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

where, within $\mathsf{Lm}\, x\, X$, any occurrence of x in X is assumed bound. Thus, we use lower cases for variables and upper cases for terms. $\mathsf{Var}\, x$ is the variable x regarded as a term, $\mathsf{App}\, X\, Y$ is an application, "X applied to Y", and $\mathsf{Lm}\, x\, X$ is a λ-abstraction, of x in X. We usually omit spelling the variable-injection operator Var, writing x instead of $\mathsf{Var}\, x$. Terms are identified modulo the notion of α-equivalence standardly induced by the above bindings. The freshness and substitution operators,
- fresh : **var** $\rightarrow$ **term** $\rightarrow$ **bool** and
- $\_[\_/\_]$: **term** $\rightarrow$ **term** $\rightarrow$ **var** $\rightarrow$ **term**,
are the standard ones, namely: $\mathsf{fresh}\, x\, X$ holds iff the variable x is fresh for (i.e., does not appear free in) the term X; $X[Y/y]$ is the term obtained from X by substituting, in a capture-free manner, all occurrences of y with Y.

We refer to the above type, **term** with the syntactic constructors (C), freshness (F) and substitution (S), as $\mathbf{term}_{C,F,S}$. Besides syntactic constructs and substitution, we occasionally consider the swapping operator, $\_[\_ \wedge \_]$: **term** $\rightarrow$ **var** $\rightarrow$ **var** $\rightarrow$ **term**, where $X[x \wedge y]$ is obtained from X by swapping (interchanging) all occurrences of x and y.

2. Two problems of rigorous/formal reasoning

We next recall two typical situations arising in formal reasoning on syntax with bindings – semantic domain interpretation and Higher Order Abstract Syntax (HOAS) representation – as motivation for the desire to have convenient (conceptual and formal) tools for recursive definitions sensible to the mechanisms of binding *and substitution*.

2.1 Problem I – Interpretation of syntax in semantic domains

Suppose we want to interpret terms in a semantic domain D (ranged over by d) endowed with operators $\mathsf{APP} : D \rightarrow D \rightarrow D$ and $\mathsf{LM} : (D \rightarrow D) \rightarrow D$, by matching the syntactic App and Lm with the semantic APP and LM, mapping syntactic binders to functional binders. For this, we need the collection of valuations (of variables into the domain), $\mathbf{val} = (\mathbf{var} \rightarrow D)$, ranged over by ρ. Then, the definition of the interpretation, $[\_] : \mathbf{term} \rightarrow \mathbf{val} \rightarrow D$, needs to proceed recursively on the structure of terms:
(1) $[x]\ \rho = \rho\ x$, **(2)** $[\mathsf{App}\ X\ Y]\ \rho = \mathsf{APP}\ ([X]\ \rho)\ ([Y]\ \rho)$,
(3) $[\mathsf{Lm}\ x\ X]\ \rho = \mathsf{LM}\ (\lambda d : D.\ [X]\ (\rho[x \leftarrow d]))$,
where $\rho[x \leftarrow d]$ is the valuation ρ updated at x with d. Moreover, we usually wish to prove the following:
- Obliviousness of the interpretation to the variables fresh for the represented term: **(4)** $\mathsf{fresh}\ x\ X\ \wedge\ \rho =_x \rho' \Longrightarrow [X]\ \rho = [X]\ \rho'$, where $\rho =_x \rho'$, read "ρ equals ρ' everywhere but in x", is defined to be $\forall y \neq x.\ \rho\ y = \rho'\ y$.
- Compositionality of substitution w.r.t. environment update, a.k.a. the *substitution lemma*: **(5)** $[Y[X/x]]\ \rho = [Y]\ (\rho[x \leftarrow [X]\ \rho])$.
Parenthesis. The above recursive definition and facts to be proved follow a canonical pattern of interpreting syntax in denotational semantics and formal logic. To give another well-known example, the semantics of FOL formulas φ is typically given as a relation $M \models_\rho \varphi$, read "M satisfies φ for the valuation $\rho : \mathbf{var} \rightarrow M$"; and the clause for the universal quantifier, which we denote by All, is the following: $M \models_\rho \mathsf{All}\ x\ \varphi$ iff $\forall m \in M.\ M \models_{\rho[x \leftarrow m]} \varphi$. Despite the different notation, this clause is of the same nature as that for Lm, in that the syntactic variable binding is captured in both cases by a semantic abstraction (functional in one case, universal in the other) in conjunction with environment update. Moreover, here (in FOL) we also wish to prove that the interpretation does not depend on the fresh variables and that the substitution lemma holds. In fact, most of the basic properties of FOL, including those necessary for the proof of the completeness theorem, rely on these two. The solution discussed below also applies to this FOL example and to many others in the same category. **End of parenthesis.**

The problem with the above definition of $[\_]$ is that, at the Lm-clause, we have to show the choice of the representatives x and X in $\mathsf{Lm}\, x\, X$ immaterial. In other words, this clause proceeds as if Lm were a *free* construct on terms, while it is not (since there exist x, X, x', X' such that $(x, X) \neq (x', X')$ and $\mathsf{Lm}\, x\, X = \mathsf{Lm}\, x'\, X'$). Here is another way to express the problem. Let us call "quasi-terms" the absolutely free data type generated by the constructors Var, App and Lm, without factoring to α-equivalence. What we called "terms" are thus quasi-terms factored by α, i.e., α-classes of quasi-terms. Now, the above is a valid definition if regarded on quasi-terms; however, to lift this to an operator on terms, we need to prove that it preserves α-equivalence. This problem is typically by-passed in informal texts, relying on "folklore", but needs to be faced and solved within a formalization. This folklore problem is more challenging than it may look at first. We urge the reader to try to solve it before reading our solution. (A proof of compatibility with α does not go through by induction directly; other auxiliary facts need to be discovered and proved in parallel; and care is required for the choice of these facts.)

2.2 Problem II – Compositional HOAS representation

HOAS [22, 31] prescribes the representation of formal systems of interest, referred to as the *object systems*, into a fixed, but customizable *logical framework*, a.k.a. the *meta system*, by capturing object-level features such as binding mechanisms directly as corresponding meta-level features. While logical frameworks in current use are typically more complex (LF [22], intuitionistic HOL [30], etc.), here we restrict ourselves to the λ-calculus augmented with constants as the logical framework (as in [7, 8, 25]). But our discussion also applies to the above more complex frameworks.

To augment the λ-calculus with constants, we fix **const**, ranged over by c, a set of *constants*. We extend the syntax of λ-calculus with these constants as usual, so that the new terms, which we call **const**-terms and whose set we denote by **term**(**const**), are now: either (injections of) variables x, or applications $\mathsf{App}\, X\, Y$, or λ-abstractions $\mathsf{Lm}\, x\, X$ (as before), or injections of constants, $\mathsf{Ct}\, c$. Constants do not participate in bindings and are unaffected by substitution. Just like for variables, we usually omit the constant-injection Ct, writing c instead of $\mathsf{Ct}\ c$.

We take **term**(**const**) as our logical framework. (A complete logical framework, of course, also has judgment mechanisms, typically given by a type system in combination with a reduction relation; for instance, in our case, reduction may be β or $\beta\eta$-reduction. Here, we ignore the largely orthogonal judgment aspect of HOAS and consider the syntax representation aspect only.) Say our object system is the λ-calculus itself (without constants). Its natural HOAS representation into **term**(**const**) proceeds as follows:
- Instantiate **const** to a two-element set, $\{\mathsf{APP}, \mathsf{LM}\}$, to match the two syntactic constructors App and Lm. (Note that one does not represent object-level variables and their injection into terms explicitly, as they will be captured directly by the meta-level items.)
- Define the representation map $\mathsf{rep} : \mathbf{term} \rightarrow \mathbf{term}(\{\mathsf{APP}, \mathsf{LM}\})$ recursively on the structure of terms: **(1)** $\mathsf{rep}\ x = x$;
(2) $\mathsf{rep}(\mathsf{App}\ X\ Y) = \mathsf{App}\ (\mathsf{App}\ \mathsf{APP}\ (\mathsf{rep}\ X))\ (\mathsf{rep}\ Y)$;
(3) $\mathsf{rep}(\mathsf{Lm}\ x\ X) = \mathsf{App}\ \mathsf{LM}\ (\mathsf{Lm}\ x\ (\mathsf{rep}\ X))$.

Much of what makes HOAS encodings so convenient is *substitution compositionality*: **(4)** $\mathsf{rep}(Y[X/x]) = (\mathsf{rep}\ Y)[(\mathsf{rep}\ X)/x]$, and indeed this is the main fact that is typically proved informally, by "pen-and-paper", about a HOAS representation. Preservation of freshness is another fact that turns out to be useful (see, e.g., [21], page 657), even though this is seldom acknowledged explicitly: **(5)** $\mathsf{fresh}\ x\ X \Longrightarrow \mathsf{fresh}\ x\ (\mathsf{rep}\ X)$. As before, the problem with the definition of rep and with the (customary) informal proof of compositionality is the non-rigorous treatment of the Lm-case.

3. Horn-based recursion with built-in substitution

In this section, we show that we can regard $\mathbf{term}_{C,F,S}$ as an "ordinary" data type, of the same kind with, say, finite sets or bags; namely, we can regard it as the initial model of a suitable Horn theory. But in order to explain the intuition behind this view, we first recall how "ordinary" data type recursion is standardly obtained and how the initial model of a Horn theory is constructed.

3.1 Ordinary data types

To define a function H (primitively) recursively from an inductive data type, say, $I\!N$, to another set A, $H : I\!N \to A$, one needs to express the behavior of H w.r.t. the number constructors, 0 and Suc, i.e., write equations of the forms **(1)** $H\,0 = 0_A$ and **(2)** $H\,(\mathsf{Suc}\ n) = \mathsf{Suc}_A\ (H\,n)$, where $0_A \in A$ and $\mathsf{Suc}_A : A \to A$. In other words, one needs to endow A with "number-like" operations, 0_A and Suc_A, i.e., organize A as a *model* for the signature having a constant symbol, 0, and a unary operation symbol, Suc.

The above simple recipe is valid for defining functions from an *absolutely free* data type such as $I\!N$. When the constructors are not free, simply defining on the target domain operations matching these constructors may not suffice for properly defining a function. For example, consider the data type $\mathbf{P}_f(U)$ of finite sets with elements in a fixed universe U (ranged over by u), with the singleton and the union constructors. To define a function H from $\mathbf{P}_f(U)$ to a set A, it does not suffice to endow A with operations $\{\_\}^A : U \to A$ and $\cup^A : A \to A \to A$, and state **(3)** $H\,\{x\} = \{x\}^A$ and **(4)** $H\,(S \cup T) = (H\,S) \cup^A (H\,T)$. Indeed, an H satisfying (3) and (4) may not exist. However, we know that $\mathbf{P}_f(U)$ is a data type *specifiable by equations*: if we start with formal expressions E given by the grammar $E\ ::=\ \{u\} \mid E_1 \cup E_2$ (where $\{\_\}$ and $\cup$ do not have any set-theoretic meaning, but are just free constructors) and then factor these expressions by all equalities deducible from associativity, commutativity and idempotence (ACI), we obtain a data type isomorphic to $\mathbf{P}_f(U)$. A useful consequence of this is that the clauses (3) and (4) do yield a valid definition of a function H provided the ACI properties hold for $\{\_\}^A$. (In other words, $\mathbf{P}_f(U)$ is the *initial ACI model*.) In this sense, $\mathbf{P}_f(U)$ is an "ordinary" data type, i.e., one freely generated by some constructors modulo some equations.

3.2 Terms as an ordinary data type

So is $\mathbf{term}_{C,F,S}$ also "ordinary"? To answer this question, we first remark that, due to the freshness operator, which is better regarded as a relation, we need to slightly relax our notion of "ordinary" data type, accepting not only operations, but also relations, and not only equations, but more generally *Horn clauses* – fortunately, this relaxation is unproblematic (w.r.t. our goal of obtaining a recursion principle), since Horn theories also admit initial models.

Next, we recall the way in which the initial model of a given Horn theory $\mathcal{H}$ is constructed: start, as before, with formal expressions built using the involved operation symbols, and then proceed to define the equality relation (by which one will need to factor to obtain the end-product type) as well as the other relations (corresponding to the relation symbols) *minimally*, as containing just what can be inferred from $\mathcal{H}$; therefore, the construction of the initial Horn model proceeds as if defining equality and all the relations *mutually recursively*. By contrast, term equality, i.e., α-equivalence, and substitution (main components of $\mathbf{term}_{C,F,S}$) are traditionally defined in a rather cumbersome manner: first substitution is defined on quasi-terms (i.e., "raw" terms, not yet factored to α) via an auxiliary "pick fresh" operator, then α is defined using substitution (or sometimes swapping) on quasi-terms, and finally substitution is shown to not depend on the choice made by "pick fresh" and to be compatible with α. Such traditional definitions are hardly exciting as a basis for a useful Horn specification (hence a useful recursion principle), due to the "immaterial" presence of "pick fresh".

Therefore, our original question becomes: can we have simple (yet complete) mutually recursive definitions of α-equivalence, freshness and substitution? It turns out the answer is positive,[1] if we renounce the ambition of having substitution *fully specified on quasi-terms*. Namely, in the Lm-case, when variable capture occurs, instead of eagerly "picking fresh" in order for substitution to go under Lm, as in $(\mathsf{Lm}\ x\ X)[Z/z] = \mathsf{Lm}\ x'\ (X[x'/x][Z/z])$ where $x' = \mathsf{pickFresh}\ \{z, Z\}$ (as is customarily done), we choose to lazily "define" substitution only when interaction with Lm is smooth, namely, $x \neq z\ \wedge\ \mathsf{fresh}\ x\ Z \Longrightarrow (\mathsf{Lm}\ x\ X)[Z/z] = \mathsf{Lm}\ x\ (X[Z/z])$; for the rest of the cases, an α-renaming clause, $x \neq y \wedge \mathsf{fresh}\,y\,X \Longrightarrow \mathsf{Lm}\,y\,(X[y/x]) = \mathsf{Lm}\,x\,X$, will eventually "make way" for substitution under Lm. (The latter two clauses are reflected in the definition of FSb-models below, as S4 and A.)

These observations lead to a simple Horn characterization of $\mathbf{term}_{C,F,S}$ and to a corresponding recursion principle; the latter states that, given a model of this Horn theory, there exists a (unique) constructor-, freshness- and substitution- compositional map from the term model to it. We think of such models as containing "generalized" terms, i.e., items with term-like operations on them (which we later refer to as "generalized" constructor, freshness and substitution operators) satisfying term-like properties (Horn clauses) – hence the prefix "g" in the notations below.

A *fresh-substitution-model* (*FSb-model*) consists of a set A (ranged over by gX, gY, gZ) together with operations and relation $\mathsf{gVar} : \mathbf{var} \to A$, $\mathsf{gApp} : A \to A \to A$, $\mathsf{gLm} : \mathbf{var} \to A \to A$, $\_[\_/\_]^g : A \to A \to \mathbf{var} \to A$, $\mathsf{gFresh} : \mathbf{var} \to A \to \mathbf{bool}$, satisfying the following properties:
F1: $x \neq z \Longrightarrow \mathsf{gFresh}\ z\ (\mathsf{gVar}\ x)$,
F2: $\mathsf{gFresh}\ z\ gX\ \wedge\ \mathsf{gFresh}\ z\ gY \Longrightarrow \mathsf{gFresh}\ z\ (\mathsf{gApp}\ gX\ gY)$,
F3: $\mathsf{gFresh}\ z\ (\mathsf{gLm}\ z\ gX)$,
F4: $\mathsf{gFresh}\ z\ gX \Longrightarrow \mathsf{gFresh}\ z\ (\mathsf{gLm}\ x\ gX)$,
S1: $(\mathsf{gVar}\ z)[gZ/z]^g = gZ$,
S2: $x \neq z \Longrightarrow (\mathsf{gVar}\ x)[gZ/z]^g = \mathsf{gVar}\ x$,
S3: $(\mathsf{gApp}\ gX\ gY)[gZ/z]^g = \mathsf{gApp}\ (gX[gZ/z]^g)\ (gY[gZ/z]^g)$,
S4: $x \neq z \wedge \mathsf{gFresh}\ x\ gZ \Longrightarrow (\mathsf{gLm}\ x\ gX)[gZ/z]^g = \mathsf{gLm}\ x\ (gX[gZ/z]^g)$,
A: $x \neq y \wedge \mathsf{gFresh}\ y\ gX \Longrightarrow \mathsf{gLm}\ y\ (gX[(\mathsf{gVar}\ y)/x]^g) = \mathsf{gLm}\ x\ gX$.

Note that, indeed, the above properties, F1-F4, S1-S4 and A, form an (infinitary) Horn theory $\mathcal{H}$ over the first-order language having:
- a binary operation symbol for application;
- for each variable x:
— a constant symbol for the x-injection as a term,
— a unary operation symbol for λx-abstraction,
— a binary operation symbol for substitution for x,
— a binary relation symbol for freshness of x.
(Note that negative conditions such as $x \neq z$ are *not* part of the $\mathcal{H}$ clauses, but are meta-level conditions – thus, e.g., F1 represents an infinite collection of Horn clauses, one for each pair of variables (x, z) with $x \neq z$.)

[1] Interestingly, the answer becomes negative if we remove any of the three – this is further evidence for the suitability of taking freshness and substitution as "built-ins" in the syntax.

Terms form of course a model of $\mathcal{H}$, since the latter consists of well-known properties of terms;[2] and the next theorem says this model is *initial* in $\mathcal{H}$:

THEOREM 1. *Let A be an FSb-model. Then there exists a unique map H :* **term** $\rightarrow$ *A commuting with the constructors, i.e.:*
(a) $H\ x = \mathsf{gVar}\ x$, **(b)** $H\ (\mathsf{App}\ X\ Y) = \mathsf{gApp}\ (H\ X)\ (H\ Y)$,
(c) $H\ (\mathsf{Lm}\ x\ X) = \mathsf{gLm}\ x\ (H\ X)$.
Additionally, H commutes with substitution and preserves freshness, i.e.: **(d)** $H\ (Y[X/x]) = (H\ Y)[(H\ X)/x]^g$,
(e) $\mathsf{fresh}\ x\ X \Longrightarrow \mathsf{gFresh}\ x\ (H\ X)$.

Proof sketch. **Existence:** First, note that, if H : **term** $\rightarrow A$ is an $(\mathsf{Var}, \mathsf{App}, \mathsf{Lm})$-morphism, i.e., if it commutes with the indicated three operators, then it also commutes with substitution and preserves freshness. This can be proved by induction on terms, using F1-F4, S1-S4, and the fact that **term** itself satisfies the corresponding clauses. This fact allows us to focus on the simpler task of finding a $(\mathsf{Var}, \mathsf{App}, \mathsf{Lm})$-morphism between **term** and A.

Next, recall that **term** consists of "terms up to α" i.e., of α-equivalence classes. We call *quasi-terms* the items not yet factored by α and let qVar, qApp, qLm, qFresh and $\_[\_ \wedge \_]^q$ be the corresponding operarors (variable injection, application, abstraction, and swapping) on quasi-terms. Let F : **qTerm** $\rightarrow A$ be the unique $(\mathsf{Var}, \mathsf{App}, \mathsf{Lm})$-morphism given by the absolute initiality of **qTerm**. I.e., F is defined by standard recursion:
- $F(\mathsf{qVar}\ x) = \mathsf{gVar}\ x$;
- $F(\mathsf{qApp}\ P\ Q) = \mathsf{gApp}\ (F\ P)\ (F\ Q)$;
- $F(\mathsf{qLm}\ x\ P) = \mathsf{gLm}\ x\ (F\ P)$.

Using F1-F4, one can show by induction on quasi-terms that F preserves freshness, namely:
(1) $\mathsf{qFresh}\ x\ P \Longrightarrow \mathsf{gFresh}\ x\ (F\ P)$.

Next, using (1), S1-S4, F1-F4 and A, one can show by induction on quasi-terms that, under an appropriate freshness assumption, F commutes with swapping versus substitution, namely:
(2) $\mathsf{qFresh}\ y_1\ P \Longrightarrow F\ (P[y_1 \wedge y]^q) = (F\ P)[(\mathsf{gVar}\ y_1)/y]^g$.
(Above, considering swapping on the left-hand side of the equality is much more convenient than considering variable-for-variable substitution, as the latter is not well-behaved on quasi-terms.)

Next, using (1), (2), S1-S4, F1-F4 and A, one can show by induction on quasi-terms that F respects α-equivalence, i.e.,
(3) $P \simeq_\alpha Q$ implies $F\ P = F\ Q$.

Finally, (3), together with the definitional clauses of F and the fact that α is a congruence on quasi-terms imply standardly (by the universal property of quotient FOL models) the existence of a $(\mathsf{Var}, \mathsf{App}, \mathsf{Lm})$-morphism H : **term** $\rightarrow A$, as desired.
Uniqueness: By an easy induction on terms. □

We call a map H as in Thm. 1 an *FSb-morphism*. Thus, Thm. 1 provides the following recursion principle: if one defines an FSb-model on the desired target domain (and thus discharges the resulted proof obligations, namely, F1-F4, S-S4 and A), then one obtains a compositional map between terms and the target domain, i.e., an FSb-morphism. Note that the particular "implementation" of this FSb-morphism is completely irrelevant (and is "hidden" in the proof of Thm. 1), since its abstract properties (a)-(c) characterize it uniquely, and, together with (d) and (e), form a convenient basis for working with it. We are now ready to present the promised solutions to the problems from Section 2.

3.3 Solution to Problem I

We let A, the set of *semantic values*, ranged over by s, t, be **val** $\rightarrow D$. We organize A as an FSb-model. The desired recursive equations from Section 2 leave us no choice about the generalized constructor operators on A:
- $\mathsf{gVar}\ x = \lambda\rho.\ \rho\ x$;
- $\mathsf{gApp}\ s\ t = \lambda\rho.\ \mathsf{APP}\ (s\ \rho)\ (t\ \rho)$;
- $\mathsf{gLm}\ x\ s = \lambda\rho.\ \mathsf{LM}\ (\lambda d.\ s\ (\rho[x \leftarrow d]))$.

Moreover, the desired freshness obliviousness property imposes that generalized freshness on A be at least as strong as the following relation, and we choose to define it as precisely this relation: $\mathsf{gFresh}\,x\ s = (\forall \rho, \rho'.\ \rho =_x \rho' \Longrightarrow s\ \rho = s\ \rho')$.

Finally, the desired substitution compositionality leaves us no choice about the definition of generalized substitution on A: $s[t/x]^g = \lambda\rho.\ s\ (\rho[x \leftarrow t\ \rho])$.

Thus, saying that the clauses and desired facts listed in Section 2 hold is the same as saying that the map $[\_]$ is an FSb-morphism. For instance, substitution compositionality, i.e.,
$\forall x\ Y\ X\ \rho.\ [Y[X/x]]\ \rho = [Y]\ (\rho[x \leftarrow [X]\ \rho])$
means $\forall x\ Y\ X.\ [Y[X/x]] = \lambda\ \rho.\ [Y]\ (\rho[x \leftarrow [X]\ \rho])$,
which means $\forall x\ Y\ X.\ [Y[X/x]] = [Y]\ [[X]/x]^g$,
which means: $[\_]$ commutes with substitution.

According to Thm. 1, what remains to do in order to have Problem I resolved is checking that A with the above structure is indeed an FSb-model, i.e., satisfies the clauses F1-F4, S1-S4 and A. It turns out that here, *as well as in most of the examples we consider later*, checking these facts is trivial (and usually automatically discharged in Isabelle – see Section 6). For instance, the most complex of them, A, here requires that $\forall\ \rho_1\ \rho_2.\ \rho_1 =_y \rho_2 \Longrightarrow s\ \rho_1 = s\ \rho_2$ implies $s(\rho[y \leftarrow d][x \leftarrow d]) = s(\rho[x \leftarrow d])$, which can be discharged immediately.

3.4 Solution to Problem II

We take A to be **term**$(\{\mathsf{APP}, \mathsf{LM}\})$ (the set of λ-terms with two constants). We organize A as an FSb-model.

Again, the generalized constructor operators, the weakest notion of freshness, and the generalized substitution on A are all predetermined by the desired recursive equations:
(1) $\mathsf{gVar}\,x = x$; **(2)** $\mathsf{gApp}\,X\ Y = \mathsf{App}\ (\mathsf{App}\ \mathsf{APP}\ X)\ Y$;
(3) $\mathsf{gLm}\ x\ X = \mathsf{App}\ \mathsf{LM}\ (\mathsf{Lm}\ x\ X)$; **(4)** $\_[\_/\_]^g = \_[\_/\_]$;
(5) $\mathsf{gFresh} = \mathsf{fresh}$.

Notice that gApp and gLm are different from App and Lm, while, to the contrary, gFresh and $\_[\_/\_]^g$ are just regular freshness and substitution – indeed, a main point of HOAS is the reuse of substitution. Again, it is immediate to check that A is an FSb-model, yielding, by Thm. 1, the existence of a map rep : **term** $\rightarrow$ **term**$(\{\mathsf{APP}, \mathsf{LM}\})$ satisfying the desired recursive equations and in addition preserving freshness and substitution.

3.5 Bottom line to our solutions

There are two advantages in employing the described recursors instead of directly attempting a representative-independent definition (perhaps proved to be so, dynamically, at definition-time): **(I)** clear a priori picture of what needs to be proved; **(II)** extra built-in compositionality results.

4. More examples

Our Horn-based principle is surprisingly general. We could not find in the literature any example of a syntactic map supposed to feature compositionality (of some kind) w.r.t. substitution and not falling in the scope of this principle. In this section we give more examples.

As seen in the solutions for Problems I and II, listing all the desired clauses – those for the term constructors, the *weakest* one for freshness, and the one for substitution – determines a model. Therefore, when discussing the next examples, we only spell out the clauses, leaving implicit the construction of the corresponding model.

[2] Note also that clauses S1-S4 coincide with the usual Nominal Logic definition of substitution, which also exploits the aforementioned idea of "lazy" clauses, however *not* with the purpose of obtaining a substitution-compositional recursor as we do here – see also Section 8.

4.1 The number of free occurrences of a variable in a term

Given a variable z, the intended definitional clauses of the map $\mathsf{no}^z : \textbf{term} \to I\!N$, taking any term X to the number of free occurrences of z in X, are the following:
- $\mathsf{no}^z\ x = \text{if } x = z \text{ then } 1 \text{ else } 0$;
- $\mathsf{no}^z(\mathsf{App}\ X\ Y) = (\mathsf{no}^z\ X) + (\mathsf{no}^z\ Y)$;
- $\mathsf{no}^z(\mathsf{Lm}\ x\ X) = \text{if } x = z \text{ then } 0 \text{ else } \mathsf{no}^z\ X$.

To make them into a rigorous definition, we need to ask: what is the desired/necessary relationship between no^z and freshness on one hand and no^z and substitution on the other. (In fact, an employment of this map in larger developments would typically need to ask these questions anyway.)

The answer to the the first question is the simpler one:
- $\mathsf{fresh}\ z\ X \implies \mathsf{no}^z\ X = 0$.

To the substitution question however, we cannot answer for a fixed z, but need to consider all no^z's at the same time, obtaining:
- $\mathsf{no}^z(X[Y/y]) = \begin{cases} (\mathsf{no}^y\ X) * (\mathsf{no}^z\ Y), & \text{if } y = z \\ (\mathsf{no}^z\ X) + (\mathsf{no}^y\ X) * (\mathsf{no}^z\ Y), & \text{otherwise} \end{cases}$

This of course suggests considering the simultaneous version of the map, $\mathsf{no} : \textbf{term} \to (\textbf{var} \to I\!N)$, where $\mathsf{no}\,X\,z$ now denotes what used to be $\mathsf{no}^z\ X$. We obtain:
- $\mathsf{no}\ x\ z = \text{if } x = z \text{ then } 1 \text{ else } 0$;
- $\mathsf{no}\,(\mathsf{App}\ X\ Y)\,z = (\mathsf{no}\ X\ z) + (\mathsf{no}\ Y\ z)$;
- $\mathsf{no}\,(\mathsf{Lm}\ x\ X)\,z = \text{if } x = z \text{ then } 0 \text{ else } \mathsf{no}\ X\ z$.
- $\mathsf{fresh}\ z\ X \implies \mathsf{no}\ X\ z = 0$;
- $\mathsf{no}\,(X[Y/y])\,z = \begin{cases} (\mathsf{no}\ X\ y) * (\mathsf{no}\ Y\ z), & \text{if } y = z \\ (\mathsf{no}\ X\ z) + (\mathsf{no}\ X\ y) * (\mathsf{no}\ Y\ z), & \text{otherwise,} \end{cases}$

which do work as a definition of no by our method, in that our method yields the existence of a (unique) function no such that the above 5 facts hold. (Discharging the resulted proof goals is trivial arithmetic.)

4.2 CPS transformation

Consider the task of defining the call-by-value to call-by-name continuation passing-style (CPS) transformation discussed in [34] in the context of λ-calculus. The transformation goes recursively by the following clauses on the syntax of terms:
- $\mathsf{cps}\ x = \mathsf{Lm}\ k\ (\mathsf{App}\ k\ x)$, for some $k \neq x$;
- $\mathsf{cps}(\mathsf{Lm}\,x\,X) = \mathsf{Lm}\,k\,(\mathsf{App}\,k\,(\mathsf{Lm}\,x\,(\mathsf{cps}\ X)))$, for fresh $k \in \textbf{var}$;
- $\mathsf{cps}\ (\mathsf{App}\ X\ Y) =$

$\mathsf{Lm}\ k\,(\mathsf{App}\ \ (\mathsf{cps}\ X)$
$\qquad\qquad (\mathsf{Lm}\ x\ (\mathsf{App}\ (\mathsf{cps}\ Y)$
$\qquad\qquad\qquad\qquad (\mathsf{Lm}\ y\ (\mathsf{App}\ (\mathsf{App}\ x\ y)\ k)))))$,

for fresh distinct variables k, x, y.

(Above, the most specific clause is the one for App, which follows the usual sequential interpretation of CPS. Namely, let X' and Y' be $\mathsf{cps}\ X$ and $\mathsf{cps}\ Y$, i.e., the CPS-transformed versions of X and Y. Then, given any continuation k, $\mathsf{cps}\ (\mathsf{App}\ X\ Y)$ is evaluated as follows:
- X' is evaluated and the result is passed, as x, to the continuation starting with "$\mathsf{Lm}\ x$",
- which in turn evaluates Y' and passes the result, as y, to the continuation starting with "$\mathsf{Lm}\ y$",
- which in turn evaluates $\mathsf{App}\ x\ y$ and finally passes the result to the original continuation, k.

Consequently, call-by-value behavior is achieved with call-by-name mechanisms.)

As before, the problem is making the above definition rigorous. Now, trying to approach this directly using our Horn machinery for the **term** syntax does not work for the following reason: the desired transformation does not commute with arbitrary substitution, but only with substitution of *values* for variables, where a value is a term which is either a variable or a Lm-abstraction. I.e., as shown in [34] (Lemma 1 on page 149), $\mathsf{cps}\ (X[\mathit{Yvl}/y]) = (\mathsf{cps}\ X)\ [(\mathsf{cps}\ \mathit{Yvl})/y]$ does hold for all values Yvl, but not for arbitrary terms. Interestingly, the solution lays here in recognizing the proper granularity of the syntax suggested above. Indeed, the aforementioned restricted substitution compositionality, *as well as all the rest of the call-by-value theory developed in [34]*, requires that, under call-by-value, the syntactic categories emphasize value terms. Namely, we are better off if we work with the following variation of **term**, split in two syntactic categories: $\textbf{term}_{\textbf{\textit{fl}}}$, of *full terms*, ranged over by X, Y, and $\textbf{term}_{\textbf{\textit{vl}}}$, of *value terms*, (or, simply, *values*), ranged over by *Xvl*, *Yvl*, defined mutually recursively by:

$X\ ::=\ \mathsf{InVl}\ \mathit{Xvl} \mid \mathsf{App}\ X\ Y \qquad\qquad \mathit{Xvl}\ ::=\ x \mid \mathsf{Lm}\ x\ X$

Thus, $(\textbf{term}_{\textbf{\textit{fl}}}, \textbf{term}_{\textbf{\textit{vl}}})$ is **term** with values singled out as a separate category (InVl being the injection of values into full terms). Here, only substitution of values for variables makes sense, "institutionalizing" the following semantic remark from [34] on page 135: "free variables should be thought of as ranging over values and not arbitrary terms". In this two-sorted context, the corresponding version of our recursion principle does the job of defining the mutually recursive maps $\mathsf{cps} : \textbf{term}_{\textbf{\textit{fl}}} \to \textbf{term}$ and $\mathsf{cps}_{\mathsf{vl}} : \textbf{term}_{\textbf{\textit{vl}}} \to \textbf{term}$, together with the (proved) statements of their freshness and substitution preservation, by the clauses:

(1) $\mathsf{cps}_{\mathsf{vl}}\ x = x$; **(2)** $\mathsf{cps}_{\mathsf{vl}}\ (\mathsf{Lm}\ x\ X) = \mathsf{Lm}\ x\ (\mathsf{cps}\ X)$;
(3) $\mathsf{cps}(\mathsf{InVl}\ \mathit{Xvl}) = \mathsf{Lm}\,k\,(\mathsf{App}\,k\,(\mathsf{cps}_{\mathsf{vl}}\mathit{Xvl}))$, for a fresh $k \in \textbf{var}$;
(4) $\mathsf{cps}(\mathsf{App}\ X\ Y) = \langle\text{same as before}\rangle$
(5) $\mathsf{fresh}\ y\ X \implies \mathsf{fresh}\ y\ (\mathsf{cps}\ X)$;
(6) $\mathsf{fresh}\ y\ \mathit{Xvl} \implies \mathsf{fresh}\ y\ (\mathsf{cps}_{\mathsf{vl}}\ \mathit{Xvl})$;
(7) $\mathsf{cps}_{\mathsf{vl}}\ (\mathit{Xvl}\,[\mathit{Yvl}/y]) = (\mathsf{cps}_{\mathsf{vl}}\ \mathit{Xvl})\ [(\mathsf{cps}_{\mathsf{vl}}\ \mathit{Yvl})/y]$;
(8) $\mathsf{cps}\ (X[\mathit{Yvl}/y]) = (\mathsf{cps}\ X)\ [(\mathsf{cps}\ \mathit{Yvl})/y]$.

Note that the originally intended behavior for variables and Lm-abstractions now follows by inlining $\mathsf{cps}_{\mathsf{vl}}$, for InVl-wrapped values: **(a)** $\mathsf{cps}\,(\mathsf{InVl}\ x) = \mathsf{Lm}\,k\,(\mathsf{App}\,k\,x)$; **(b)** $\mathsf{cps}\ (\mathsf{InVl}\ (\mathsf{Lm}\ x\ X)) = \mathsf{Lm}\ k\ (\mathsf{App}\ k\ (\mathsf{Lm}\ x\ (\mathsf{cps}\ X)))$.

Checking the clauses necessary for the definition to work (i.e., the clauses defining FSb-models for the considered two-sorted syntax $(\textbf{term}_{\textbf{\textit{fl}}}, \textbf{term}_{\textbf{\textit{vl}}})$) is again routine, provided several basic properties of freshness and substitution are available. There is of course the question whether it is worth "storing" call-by-value λ-calculus in a different data type from standard λ-calculus, or should one rather define values inside standard λ-calculus and work with these. This is an engineering, not a mathematical question. But if one opts for a different data type, then the isomorphism between $\textbf{term}_{\textbf{\textit{fl}}}$ and **term** is yet another (trivial) application of our recursion principle.

4.3 Other examples

These include the classic de Bruijn interpretation map from [12], the CPS transformation in the opposite direction (from call-by-name to call-by-value) from [34] and the translation of the LF syntax into untyped λ-calculus (meant to be subsequently Curry-style typed) employed in the proof of strong normalization for the LF reduction [22]. The reader is invited to try our method on his/her own examples pertaining to syntax with bindings – again, we believe it is very likely to work provided a substitution lemma of some kind is in sight.

4.4 Non-examples

Of course, not everything one wants to define is compositional w.r.t. substitution. Such cases fall out of the scope of our definitional principle. These include, e.g., the depth operator – an immediate symptom showing why our principle cannot handle this example is the fact that we cannot compute the depth of $X[Y/y]$ based on the depths of X and Y; in other words, we cannot answer the substitution question. So this example is problematic because it is, in some sense, syntactically under-specified, as it "ignores" substitution. An example problematic because of a somewhat opposite reason is the so-called *complete development*, $\mathsf{cdev} : \textbf{term} \to \textbf{term}$, introduced in [41] as a means to simplify the proof of the Church-Rosser the-

orem (and of other results):
(1) $\mathsf{cdev}\ x = x$; **(2)** $\mathsf{cdev}\ (\mathsf{Lm}\ x\ X) = \mathsf{Lm}\ x\ (\mathsf{cdev}\ X)$;
(3) $\mathsf{cdev}\ (\mathsf{App}\ X\ Y) = \mathsf{App}\ (\mathsf{cdev}\ X)\ (\mathsf{cdev}\ Y)$, if $\mathsf{App}\ X\ Y$ is not a β-redex (i.e., if X does not have the form $\mathsf{Lm}\ y\ Z$);
(3') $\mathsf{cdev}\ (\mathsf{App}\ (\mathsf{Lm}\ y\ X)\ Y) = (\mathsf{cdev}\ X)[(\mathsf{cdev}\ Y)/y]$.

cdev reduces all the (arbitrarily-nested) β-redexes in a term. It does not commute with substitution, mainly because it is not a "purely syntactic" map, but includes some "operational semantics". Note however that, even if the codomain consists of terms, our method does not require that substitution on the domain of terms be mapped to *standard* substitution on the codomain, but to any well-behaved notion of "generalized substitution". We therefore could define a non-standard substitution (on standard terms) that will eventually yield preservation of substitution, but this would be unnecessarily complicated *and artificial*. Our principle is not intended to have its users work hard to have their definitions go through; rather, it is aimed at situations where the issues of preservation of freshness and substitution appear naturally as further desired properties. In our formalization, we have incorporated a workaround that handles the cdev operator by a variation of our approach that renounces the substitution compositionality goal, replacing it with the less ambitious one of swapping compositionality (see the Appendix).

5. Pushing the Horn approach even further

Besides having the "up to α" definitions go through, our Horn approach infers some extra information about the defined maps: compositionality w.r.t. freshness and substitution. Can we infer even more? The answer is: yes, if we prove more facts about the constructed model. Next we state criteria for three commonly encountered properties: freshness reflection, injectiveness, surjectiveness.

An FSb-model is said to be:
- *fresh-reversing*, if the following facts (converse to F1-F4 from Section 3), hold for it: **(a)** $\mathsf{gFresh}\ z\ (\mathsf{gVar}\ x) \implies x \neq z$, **(b)** $\mathsf{gFresh}\ z\ (\mathsf{gApp}\ gX\ gY) \implies (\mathsf{gFresh}\ z\ gX\ \wedge\ \mathsf{gFresh}\ z\ gY)$, **(c)** $\mathsf{gFresh}\ z\ (\mathsf{gLm}\ x\ gX) \implies (z = x\ \vee\ \mathsf{gFresh}\ z\ gX)$;
- *constructor-injective*, if its constructor operators are mutually injective, i.e.: **(a)** $\mathsf{gVar}\ x$, $\mathsf{gApp}\ X\ Y$ and $\mathsf{gLm}\ z\ Z$ are mutually distinct; **(b)** gVar, gApp (regarded as an uncurried binary operation) and $\mathsf{gLm}\ x$ are injective;
- *inductive*, if the following induction principle is true: for all predicates φ, $\forall gX.\ \varphi\ gX$ holds provided the following hold: **(a)** $\forall x.\ \varphi\ (\mathsf{gVar}\ x)$, **(b)** $\forall\, gX\ gY.\ \varphi\ gX \wedge \varphi\ gY \implies \varphi\ (\mathsf{gApp}\ gX\ gY)$, **(c)** $\forall\, x\ gX.\ \varphi\ gX \implies \varphi\ (\mathsf{gLm}\ x\ gX)$.

The next theorem follows easily by structural induction on terms:

THEOREM 2. *Let A be an FSb-model and let H be the map from Thm. 1. Then, if A is fresh-reversing (construct-injective, inductive), then H is, respectively, fresh-reflecting (injective, surjective), where "fresh-reflecting" means:* $\mathsf{fresh}\ x\ (H\ X) \implies \mathsf{fresh}\ x\ X$.

The above result integrates even more facts as "built-ins" of the recursive definition. E.g.: the HOAS operator from Section 2 is freshness-reflective and injective; the "number of free occurrences" operator from Section 4.1 is freshness-reflecting; the CPS operator from Section 4.2 is freshness-reflective and injective. Putting together these 3 criteria, we obtain the following characterization:

THEOREM 3. *If an FSb-model A is fresh-reversing, constructor-injective and inductive, then the map H from Thm. 1 is an isomorphism of FSb-models (where "isomorphism of FSb-models" is the usual notion from first-order model theory, namely: "bijection preserving the constructors and substitution and preserving and reflecting freshness").*

This characterization of the term model is *internal*, in that it does not rely on things from "outside" the model itself (such as morphisms and other models), as does the characterization as initial Horn model. Although such an internal characterization is not the first one from the literature (see, e.g., [19], [28]), it appears to be the first expressed in terms of essential ingredients only – syntactic constructors, freshness and substitution – and also the first to be formalized and put to use (see Section 7).

6. Isabelle formalization

Above, we have presented our results for the syntax of λ-calculus terms. However, one can see that similar results hold for any other (possibly multiple-sorted) syntax with bindings, such as, e.g., that of LF [22]. For the sake of generality and reusability, we chose to formalize these results parameterizing by an *arbitrary binding signature*, within a theory of syntax consisting of:
- a wide collection of basic facts on freshness, substitution and swapping;
- customized induction principles, including fresh induction in the style of [32, 45];
- customized recursion principles (the topic of this paper).

Moreover, we have formalized all the examples discussed in this paper, in their particular syntaxes instantiated from the aforementioned general theory. We have also employed these examples in larger development, notably the formalization of a large part of Plotkin's classic paper [34].

A detailed presentation of our formal development, available at [4], falls outside the scope of this paper (but the appendix offers a few details for the interested reader). Here, we illustrate our approach by presenting the Isabelle formalization of Problem II from Sections 2.2 and 3.4 (contained in theory HOAS from [4]).

In order to relate the formal scripts described below to the mathematical notations from Sections 2.1 and 3.4, we note the following:

-(1) Unlike in the paper, in the formalization we distinguish terms from *abstractions*, an abstraction being essentially a pair (variable,term) up to α. We write Abs for the abstraction constructor (taking a variable and a term and returning an abstraction). In this setting, a binding operator such as λ no longer takes a variable and a term, but takes an abstraction (and returns a term); we write Lam for this operator. (Thus, our operator Lm from the paper can be obtained as $\mathsf{Lm}\, x\, X = \mathsf{Lam}\, (\mathsf{Abs}\ x\ X)$; the latter is actually defined as an abbreviation in our scripts.) This setting is reflected in the notion of an FSb-model too, where we have both "generalized terms" and "generalized abstractions" (and corresponding "generalized operators").

-(2) In the formalization we use a more general notion of FSb-model than in the paper, supporting not merely iteration, but full (primitive) recursion. This is reflected in the presence of extra arguments for the model operations, which take not only "generalized terms" (corresponding to the recursively computed result), but also genuine syntactic terms (corresponding to the original input at recursion time).

-(3) In the formalization, the problem setting is itself slightly more general than that in the paper. Namely, let us write **term**(**const**) for the syntax of the λ-calculus over a set of constants **const**. Then the object syntax is **term**(**const**), and the meta syntax is **term**(**metaConst**), where the set **metaConst**, of *meta-constants*, consists of copies $\mathsf{Const}\, c$ of elements $c \in$ **const** and of the new constants APP and LM.

We use the prefixes "Obj" and "Meta" for operators pertaining to the object syntax and to the meta syntax, respectively. Moreover, we write $\_[\_/_o\,\_]$ and $\_[\_/_m\,\_]$ for the object level and meta level substitutions, respectively. The Isabelle definition of the desired FSb-model, called rep_MOD, is the following:

```
definition rep_MOD :: ('const, 'const metaConst term,
'const metaConst abs)model
where rep_MOD ==
  gVar_vlm = Meta_Var,
  gAbs_lm_lm = λ y X mX. Meta_Abs y mX,
  gCt = λ c. Meta.Ct (Const c),
  gApp = λ X mX Y mY.
         Meta.App (Meta.App (Meta.Ct APP) mX) mY,
  gLam = λ A mA. Meta.App (Meta.Ct LM) (Meta.Lam mA),
  gFresh_vlm_lm = λ y X mX. Meta_fresh y mX,
  gFresh_vlm_lm_lm = λ y A mA. Meta_freshAbs y mA,
  gSubst_vlm_lm = λ Y mY y X mX. mX [mY /m y],
  gSubst_vlm_lm_lm = λ Y mY y A mA. mA [mY /m y]
```

Above, the type of rep_MOD (on the first 2 lines) is the record type of "(FSb-)models for the **term**(**const**) syntax", hence the first parameter **const**; the other two parameters from this type represent the carriers of the model, which are the meta-terms and meta-abstractions. Then there are defined the record components, i.e., the "generalized operators" on the model – X, Y and A range over object terms and abstractions, and *mX*, *mY* and *mA* over meta terms and abstractions. The "lm" and "vlm" suffixes from the operators' names are a reminiscence of our many-sorted setting: they refer to the only variable sort (vlm) and term sort (lm) for the syntax **term**(**const**).

Modulo these explanations, the reader should now recognize the definition from our resolution of Problem II in Section 3.4. E.g., the above definition of gAbs_lm_lm corresponds to the HOAS-specific recursive clause repAbs (Obj.Abs x X) $=$ Meta.Abs x (rep X), where rep denotes, as in Sections 2.1 and 3.4, the representation map on terms, and repAbs is its abstraction counterpart.

For each of the Isabelle lemmas listed below, after the keyword "lemma" comes the name of the lemma, followed by a colon, followed by the stated fact in double quotes. We omit their formal proofs. (Each of the proofs has at most 2 lines and proceeds by merely unfolding definitions and calling the automatic simplifier tool.)

Once the model has been defined, we need to check that it is indeed a "well-structured" FSb-model (expressed as the Isabelle predicate wlsFSb), i.e., that it satisfies the involved clauses for freshness and substitution (F1-F4, S1-S4 and A) – in this case, these goals are discharged automatically:

```
lemma wlsFSb_rep_MOD: "wlsFSb rep_MOD"
```

This is all we needed to check to have our recursive definition go through. However, we wish to infer some further facts for our morphism, in this case freshness reflection and injectiveness. For these, the first 2 points of Thm. 2 tell us that it suffices that the reversed fresh clauses hold (expressed by the Isabelle predicate gFresh_cls_rev) and that the generalized constructors be injective (expressed by the Isabelle predicate gCons_inj). The latter facts can again be proved automatically:

```
lemma gFresh_cls_rev_rep_MOD:"gFresh_cls_rev rep_MOD"
```

```
lemma gCons_inj_rep_MOD: "gCons_inj rep_MOD"
```

This concludes the creative part of the HOAS development. The rest is "bureaucracy", and goes the same way for all recursive definitions, as it relies on our general theory.

Parenthesis: Our Isabelle formalization of the recursion theorem Thm. 1 takes advantage of the Hilbert choice operator, building the constant rec_lm out of the "exists unique" statement from Thm. 1 (and the corresponding constant rec_lm_lm for abstractions) – i.e., given any FSb-model M, rec_lm M and rec_lm_lm M correspond to the function H from Thm. 1. Then theorem "wlsFSb rec term_FSb_morph", which is the Isabelle version of Thm. 1, says that, if M is an FSb-model, then rec_lm M and rec_lm_lm M form an FSb-morphism from terms to M. (All these are presented and illustrated in detail in the commented theory L from [4].)

Back to the HOAS development, we let rep and repAbs denote the recursive functions associated to the particular model rep_MOD, thus mapping object terms and abstractions to meta terms and abstractions:

```
definitions rep where "rep X ≡ rec_lm rep_MOD X"
and repAbs where "repAbs A ≡ rec_lm_lm rep_MOD A"
```

We now apply the aforementioned recursion theorem, "wlsFSb rec term_FSb_morph", obtaining that (rep, repAbs) is an FSb-morphism:

```
lemma term_FSb_morph_rep:
"term_FSb_morph rep repAbs rep_MOD"
```

Unfolding definitions, we see the morphism property in familiar format (similar to the one from Section 2.2):

```
lemma rep_simps:
"rep (Obj_Var x) = Meta_Var x"
"rep (Obj.Ct c) = Meta.Ct (Const c)"
"rep (Obj.App X Y) =
 Meta.App (Meta.App (Meta.Ct APP) (rep X)) (rep Y)"
"rep (Obj_Lm y X) =
 Meta.App (Meta.Ct LM) (Meta_Lm y (rep X))"
```

```
lemma rep_subst:
"rep (X [Y /o y]) = (rep X) [(rep Y) /m y]"
```

```
lemma rep_preserves_fresh:
"Obj_fresh y X  ==>  Meta_fresh y (rep X)"
```

Thus, Lemmas rep_simps, rep_subst and rep_preserves_fresh correspond to clauses (1)-(3), (4), and (5), respectively, from Section 2.2.

Additionally, we use (the Isabelle formalization of) the first 2 points of Thm. 2, to infer that this morphism is additionally freshness reflecting and injective:

```
lemma refl_freshAll_rep:
"refl_freshAll rep repAbs rep_MOD"
```

```
lemma is_injAll_rep:
"is_injAll rep repAbs"
```

And again, by unfolding definitions, we obtain more readable versions of these results:

```
lemma rep_reflects_fresh:
"Meta_fresh y (rep X) ==> Obj_fresh y X"
```

```
lemma rep_inj:
"(rep X = rep Y) = (X = Y)"
```

This concludes the definition of the HOAS representation map *and the proof of its syntactic adequacy*. Notice the high degree of automation of the development, as the proof goals generated by the FSb-model requirement are automatically discharged – this is the case for most applications of Thm. 1 and Thm. 2 throughout our formal development.

7. Certifying and relating formalizations of terms

The literature on λ-calculus theory and theorem proving abounds in approaches to representing syntax with bindings, such as α-classes, de Bruijn levels [13], de Bruijn indexes [13], locally nameless [8, 9, 18, 19], locally named [35, 38], proper weak-HOAS functions [14, 20], partial functions [42] (to list only a few; see [29] for an overview of several of these and others).

These representations have various merits: of being more efficient, more insightful w.r.t. definition and proof principles, etc.

However, all these approaches are aimed at capturing *the same Platonic notion of syntax*. So what makes them correct? Typically, work proposing such a new implementation/representation justifies its correctness by showing it isomorphic to a more standard representation, or to one that has been used many times and has a high degree of trustability. E.g., in [42], the new partial-function based representation of λ-terms from the Nominal package is proved isomorphic to the α-class-based one; in [35], the *locally named* representation is proved isomorphic to the nominal representation.

Our Thm. 3 provides a common formal basis for establishing such isomorphisms, and, in general, for proving a novel representation to be correct, namely, by *showing that it forms a fresh-reversing, constructor-injective and inductive FSb-model* (as the latter notion is unique up to fresh [preserving and reflecting] and substitution preserving isomorphism). Following this methodology, we have established in Isabelle formal isomorphisms between our representation and each of the following:
-(1) the Nominal representation [42];
-(2) the locally nameless representation underlying the Hybrid system [8, 15, 24, 25];
-(3) the *locally named* representation [35, 38].

(1) The connection with Nominal is formalized in the theory C_Nominal from [4]. We connect the λ-terms from the theory Lam_funs (located in the Nominal directory in the Isabelle distribution) with our *pure* λ-terms. The latter are λ-terms without constants, defined in our theory Pure. (We use pure terms here, since the terms in the aforementioned Nominal theory happen to not include constants.) First, we establish a bijection toNA, read "to Nominal atom" between our variables and the Nominal atoms, called "names" here – this is immediate, since both collections are countable. Then we proceed to organize the Nominal terms as an FSb-model with the required additional properties, yielding an isomorphism toN, read "to Nominal", between our pure terms and their terms. Checking the facts necessary for obtaining this isomorphism was immediate, given that Lam_funs and the underlying Nominal package provide a rich pool of basic facts. To give an example of the potential usefulness of the above isomorphism, for both parties:
- our rich theory of substitution is now available for this Nominal theory;[3]
- advanced Nominal techniques pertaining to rule induction are now available in our λ-calculus theory.

(2) For the connection with the Hybrid terms, we imported the theory Expr from the Hybrid scripts (http://hybrid.dsi.unimi.it). On top of Expr, Hybrid has several other HOAS layers – however, we were only interested in this very basic implementation layer. The connection, developed in our theory C_Hybrid (from [4]), took a little more work, given that the Hybrid representation is HOAS-oriented, hence it lacks a first-order binding operator, which we had to add ourselves. Then the Hybrid terms were organized as an FSb-model, yielding an isomorphism between our terms with constants and their terms with constants.

(3) The locally named representation from [35] is based on a distinction between parameters (or global variables), and (local) variables, only the latter being allowed (and required) to be bound. As mentioned, this representation has already been proved isomorphic to the Nominal one discussed above, by defining a relation and showing it to be total and deterministic, yielding a function – note that a main motivation of our approach is to avoid the roundabout route of defining a function first as a relation. The connection is developed in our theory C_Sato_Pollack from [4] (named after the two authors of the approach), importing a couple of theories from the scripts associated to the paper [35], available from the first author's home page: http://homepages.inf.ed.ac.uk/rpollack. Similarly to before, we define a bijection toP between our variables and their parameters, and then organize their terms as an FSb-model, yielding an isomorphism between our pure terms and their terms.

8. Conclusions and related work

Next we discuss other approaches to the main contribution of this paper – convenient recursion principles for terms with bindings. (See Section 2.10 in [36] for much more details.)

Nominal Logic is originally a non-standard logic, parameterized by various categories of *atoms* (a.k.a. names, or variables); there are built-in notions of swapping (primitive) and freshness of an atom for an object (derived) and an underlying assumption of *finite support*, essentially saying that, for each object, all but a finite number of atoms are fresh for it; all the expressible predicates have a property called *equivariance* (invariance under swapping). This non-standard logic is discussed, e.g., in [17]. However, the nominal approach can also be developed in a standard logic such as classic HOL, as shown by the Isabelle/HOL Nominal Package [42, 43, 45].

The Nominal recursor [33, 42] on terms with bindings employs the syntactic constructors Var, App and Lm, as well as a notion of permutation swapping. All the operators involved in the recursive clauses in a presumptive definition with target domain A (assumed to have a permutation swapping operator on it), in our notation $\mathsf{gVar} : \mathbf{var} \rightarrow A$, $\mathsf{gApp} : \mathbf{term} \rightarrow A \rightarrow \mathbf{term} \rightarrow A \rightarrow A$ and $\mathsf{gLm} : \mathbf{var} \rightarrow \mathbf{term} \rightarrow A \rightarrow A$, are required to have *finite support*. Moreover, a technical condition, *Freshness Condition for Binders* (FCB) (a form of freshness preservation for gLm), is required to hold. Under the above conditions, one obtains an operator $H : \mathbf{term} \rightarrow A$ which is "almost a morphism" w.r.t. the syntactic constructors, in that it commutes with Var and App in the usual sense and commutes with Lm for fresh names only, namely: $H\ (\mathsf{Lm}\ x\ X) = \mathsf{gLm}\ x\ X\ (H\ X)$ provided $\mathsf{gFresh}\ x\ \mathsf{gVar}$, $\mathsf{gFresh}\ x\ \mathsf{gApp}$ and $\mathsf{gFresh}\ x\ \mathsf{gLm}$ hold.

Compared to our main substitution-based recursion combinator (henceforth abbreviated SRC), the above Nominal recursion combinator (henceforth abbreviated NRC) has the advantage of uniformity and economy of structure (only swapping is primitive, and everything is based on it). Moreover, the consideration of the aforementioned "almost morphisms" as targets of definitions, w.r.t. which freshness for parameters may be assumed, provides extra generality to the NRC. On the other hand, the conditions that SRC requires to be checked by the user are simpler than those required by NRC, the latter involving a certain quantifier complexity, notably when considering the support of functions (although, like ours, often these conditions can be checked automatically, as shown by the various examples considered by work using the Nominal Package). Moreover, there is some benefit in staying first-order as in our Horn approach, and not getting into the second-order issue of finiteness of support. E.g., as soon as one deals with more semantic situations, such as our semantic-domain interpretation in Section 2, the finite support conditions are no longer satisfied, but a complex combination of induction and recursion is needed to have a Nominal definition eventually go through, as shown in [33] (on page 492) for this very example. Another difference between NRC and SRC is of course the presence in SRC of built-in compositionality, which relates to the user as both an obligation and a reward.

Michael Norrish's work. [27] introduces a recursor for λ-calculus involving swapping and the free-variable operator FV, and also considers parameters à la Nominal. If we ignore the largely orthogonal extra generality of [27] w.r.t. parameters, and replace the

[3] Substitution is of course definable with the Nominal package, but its many relevant properties of interaction with the other operators, including, e.g., the very recursion principles proposed in this paper, do not come with this definition, but would need to be established individually.

use of $\mathsf{FV} : \mathbf{term} \to \mathbf{P}_f(\mathbf{var})$ with the (complementary) use of $\mathsf{fresh} : \mathbf{var} \to \mathbf{term} \to \mathbf{bool}$, we find that [27] did in fact propose a Horn-like recursive principle for syntax with bindings. Interestingly, [27] also contemplates considering substitution, but renounces in favor of swapping because "permutations move around terms much more readily than substitution" ([27], page 248). However, as we show in this paper, the (indeed harder) task of having substitution "move around terms" results in a recursion principle which is *not* harder to apply (from the user's perspective), while bringing the convenience of substitution compositionality.

Non-standard-model approaches to syntax, based on functor categories [7, 16, 23]. These approaches add more structure to terms, regarding them as *terms in contexts* – this is inspired by the method of *de Bruijn levels* [13], and consists conceptually in "turning the free-variable function $\mathsf{FV} : \mathbf{term} \to \mathbf{P}(\mathbf{var})$ into extra structure on terms" ([16], page 3). Here, there is no single set of terms, but rather terms form a family of sets indexed by the "contexts", with additional categorical structure (i.e., as presheaves). The flexibility of moving between different contexts allows one to overcome the typical problems with recursive definitions on syntax. Compared to the more elementary settings discussed previously (which include our own), the non-standard model approach has the advantage of "syntactic purity" (as only the term constructors are involved) and mathematical elegance, but does have a couple of disadvantages too. Thus, it is harder to formalize in a standard theorem prover, due to the fancy category theory involved (although this is more of a (solvable) problem for the implementor, not for the user). More importantly, it seems rather tedious to employ in order to define concrete operators. Indeed, the target domain would have to be organized each time as an algebra for a suitable functor on presheaves, which means that a lot of structure and verifications would need to be provided and performed by the user. On the other hand, a category theorist may argue that all this effort would not go in vain, as the extra structure will be helpful later in proofs – and this is much like our argument in favor of built-in substitution made in this paper. Concerning the latter, both [23] and [16] show how to define substitution using their iterator. Moreover, [16] goes further and integrates substitution in the algebraic structure, inferring a version of recursion with a built-in substitution lemma, which is essentially what we do in this paper in a more standard setting.

Formalizations and case studies. Norrish has formalized his swapping-based recursion principle for the syntax of untyped λ-calculus and has included it in the distribution of the HOL4 system [6].

The Nominal package [46] has been used in several formal developments, including proofs of Church-Rosser and Standardization for λ-calculus – the latter classic result, presented, e.g., in Barendregt, is different than the call-by-name and call-by-value standardization theorems from [34] that we have formalized, in that the latter deals with more programming-oriented strategies, not allowing rewrites under λ-abstractions. Recent work at the Nominal Package [44] allows syntax with multiple (concurrent) binders, while our notion of binding signature allows only one binder at a time.

HOAS versus FOAS. In this paper, we took a *first-order abstract syntax (FOAS)* view of syntax with bindings, where variables are bound in terms by first-order operators such as Lm. By contrast, HOAS [22, 31] employs higher-order types for the binding operators. HOAS-based theorem proving and programming is currently a topic of intensive research, e.g., [1–3, 10]. HOAS recursion principles can be found, e.g., in [14, 19, 37, 40]. (See Section 3.7 in [36] for details.)

While HOAS is often more elegant than FOAS, it is not always an alternative to FOAS. For instance, the very foundations of HOAS, consisting of the definition of the meta and object logics and the associated adequacy proof, rely on (some form of) FOAS. In spite of the extensive and growing interest in HOAS and its meta-theory, our *formal* HOAS adequacy proof discussed in Section 6 seems to be the first reported in the literature. Even though it is performed for a fairly simple (albeit standard) logical framework, λ-calculus with constants, the actual proof technique works for more complex frameworks such as LF too. An important part of the meta-theory of LF has been formalized using the Nominal package, but we are not aware of actual representations and adequacy proofs performed in this setting.

Acknowledgments. We are indebted to the referees for helpful comments and suggestions. This work was supported in part by the National Science Foundation (NSF) Award number 0917218. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NSF.

References

[1] Bedwyr. `http://slimmer.gforge.inria.fr/bedwyr`.

[2] Beluga. `http://www.cs.mcgill.ca/~complogic/beluga`.

[3] Delphin. `http://cs-www.cs.yale.edu/homes/carsten/delphin`.

[4] `http://www4.in.tum.de/~popescua/Scripts.zip`.

[5] The Twelf Project, 2009. `http://twelf.plparty.org`.

[6] The HOL4 Theorem prover, 2010. http://hol.sourceforge.net/.

[7] S. J. Ambler, Roy L. Crole, and Alberto Momigliano. A definitional approach to primitive recursion over Higher Order Abstract Syntax. In *MERLIN*, 2003.

[8] Simon Ambler, Roy L. Crole, and Alberto Momigliano. Combining Higher Order Abstract Syntax with tactical theorem proving and (co)induction. In *TPHOLs*, pages 13–30, 2002.

[9] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL*, pages 3–15, 2008.

[10] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated. In *APLAS*, pages 222–238, 2007.

[11] Adam J. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP*, pages 143–156, 2008.

[12] Pierre-Louis Curien. Categorical combinators. *Information and Control*, 69(1-3):188–254, 1986.

[13] N. de Bruijn. 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.

[14] Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *TLCA*, pages 124–138, 1995.

[15] Amy P. Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with Higher-Order Abstract Syntax. *CoRR*, abs/0811.4367, 2008.

[16] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding (extended abstract). In *LICS*, pages 193–202, 1999.

[17] M. J. Gabbay. A theory of inductive definitions with α-equivalence. Ph.D. thesis. University of Cambridge, 2001.

[18] Andrew D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *HUG*, pages 413–425, 1994. ISBN 3-540-57826-9.

[19] Andrew D. Gordon and Thomas F. Melham. Five axioms of alpha-conversion. In *TPHOLs*, pages 173–190, 1996.

[20] Elsa L. Gunter, Christopher J. Osborn, and Andrei Popescu. Theory support for weak Higher Order Abstract Syntax in Isabelle/HOL. In *LFMTP*, pages 12–20, 2009.

[21] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007.

[22] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *LICS*, pages 194–204. IEEE, Computer Society Press, 1987.

[23] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *LICS*, page 204, 1999.

[24] Alberto Momigliano and Simon Ambler. Multi-level meta-reasoning with higher-order abstract syntax. In *FoSSaCS*, pages 375–391, 2003.

[25] Alberto Momigliano, Alan J. Martin, and Amy P. Felty. Two-level Hybrid: A system for reasoning using Higher-Order Abstract Syntax. *Electron. Notes Theor. Comput. Sci.*, 196:85–93, 2008.

[26] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer, 2002.

[27] Michael Norrish. Recursive function definition for types with binders. In *TPHOLs*, pages 241–256, 2004.

[28] Michael Norrish. Mechanising lambda-calculus using a classical first order theory of terms with permutations. *Higher-Order and Symbolic Computation*, 19(2-3):169–195, 2006.

[29] Michael Norrish and René Vestergaard. Proof pearl: De Bruijn terms really do work. In *TPHOLs*, pages 207–222, 2007.

[30] L. C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3), 1989.

[31] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI*, pages 199–208, 1988.

[32] Andrew M. Pitts. Nominal logic: A first order theory of names and binding. In *TACS*, pages 219–242, 2001.

[33] Andrew M. Pitts. Alpha-structural recursion and induction. *J. ACM*, 53(3), 2006.

[34] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[35] Robert Pollack and Masahiko Sato. A canonical locally named representation of binding. To appear in Journal of Automated Reasoning.

[36] Andrei Popescu. Contributions to the theory of syntax with bindings and to process algebra. Ph.D. Thesis, Univ. of Illinois, 2010. Available at `http://hdl.handle.net/2142/18477`.

[37] Andrei Popescu, Elsa L. Gunter, and Christopher J. Osborn. Strong normalization of System F by HOAS on top of FOAS. In *LICS*, pages 31–40, 2010.

[38] Masahiko Sato and Robert Pollack. External and internal syntax of the lambda-calculus. *Journal of Symbolic Computation*, 45:598–616, 2010.

[39] Carsten Schurmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In *CADE*, pages 286–300, 1998.

[40] Carsten Schurmann, Joelle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theor. Comput. Sci.*, 266(1-2):1–57, 2001.

[41] Masako Takahashi. Parallel reductions in lambda-calculus. *Inf. Comput.*, 118(1):120–127, 1995.

[42] Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reason.*, 40(4):327–356, 2008.

[43] Christian Urban and Stefan Berghofer. A recursion combinator for nominal datatypes implemented in Isabelle/HOL. In *IJCAR*, pages 498–512, 2006.

[44] Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in Nominal Isabelle. In *ESOP*, pages 480–500, 2011.

[45] Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In *CADE*, pages 38–53, 2005.

[46] Christian Urban, James Cheney, and Stefan Berghofer. Mechanizing the metatheory of LF. In *LICS*, pages 45–56, 2008.

[47] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The isabelle framework. In *TPHOLs*, pages 33–38, 2008.

More examples of Horn-based recursion

All the examples listed explicitly in this paper have been formalized. Our first example, namely, semantic interpretation (Problem I) has been included as a "built-in" in the general theory. The rest of the examples are formalized in roughly the same syntax in which they were presented in this paper (usually in a slightly more general format, allowing an unspecified number of constants too besides variables, application and λ-abstraction). Many of the examples appear in the formal scripts as parts of larger developments, the largest one being a formalization of a significant part of Plotkin's classic paper [34]. In all such cases, we claim that the overall development benefits highly from the possibility to *define the map with the desired properties and move on*, provided by our recursion principles, as opposed to trying to make the definition work by ad hoc bases.[4]

Semantic-domain interpretation. Due to its importance and generality, the semantic interpretation example (or, rather, class of examples) discussed as motivational Problem I has also been included in the general development (for an arbitrary syntax) – this is the topic of theory Semantic_Domains. A "well-structured" semantic domain (predicate wlsSEM) consists essentially of a type **sTerm**, of semantic values (called "semantic terms" in the scripts' comments), and of an interpretation of the operation symbols from a signature, where abstractions are treated as functions (thus the operations have second-order arguments to match bindings in the syntax, as for LM versus Lm at Problem I). A compositional interpretation map (predicate compInt) is a map, via valuations, of syntax to the semantic domain which is compositional with the syntactic constructors and with substitution and oblivious to freshness, as is $[\_]$ in Problem I. The main theorem, semIntAll_compInt, states that, for any given semantic domain SEM, the pair (semInt SEM, semIntAbs SEM) is indeed such a compositional interpretation map.

Instances of the general theory for particular syntaxes. So far, we have considered the following two instances:
(I) The syntax of the untyped λ-calculus with constants, with terms $X, Y, Z \in$ **term** and abstractions $A, B \in$ **abs**:
$X \ ::= \ \mathsf{Var}\,x \mid \mathsf{Ct}\,c \mid \mathsf{App}\,X\,Y \mid \mathsf{Lam}\,A \qquad A \ ::= \ \mathsf{Abs}\,x\,X$
(II) The two-sorted value-based variation of the above (as at the CPS example in Section 4), with full terms $X, Y, Z \in$ **term**$_{\boldsymbol{fl}}$, value terms *Xvl*, *Yvl*, *Zvl* $\in$ **term**$_{\boldsymbol{vl}}$ and abstractions (of value variables in full terms) $A, B \in$ **abs**$_{\boldsymbol{(vl,fl)}}$: $\quad X \ ::= \ \mathsf{InVl}\ \mathit{Xvl} \mid \mathsf{App}\,X\,Y$
$\mathit{Xvl} \ ::= \ \mathsf{Var}\,x \mid \mathsf{Ct}\ c \mid \mathsf{Lam}\,A \qquad A \ ::= \ \mathsf{Abs}\,x\,X$
where InVl is the injection of value terms into full terms. For both syntaxes, we recover the notation from this paper's previous sections by letting $\mathsf{Lm}\,x\,X$ be an abbreviation for $\mathsf{Lam}\,(\mathsf{Abs}\,x\,X)$.

Theories L1 and L are performing the instantiation of the general theory to the above syntax (I) (and LV1 and LV do the same for (II)). The instantiation process is performed in a completely uniform manner and will eventually be automated, but currently it is done by hand. Theories L1 and L most of the facts one needs to know about the λ-calculus instance of the theory, including: **(1)** basic facts about freshness, substitution and swapping; **(2)** induction principles; **(3)** the Horn-based recursion principles. All these facts, especially those pertaining to (3) (present in L1), are documented and illustrated by comments inserted in between the formal scripts.

The theory of call-by-name λ-calculus. This is the content of theories CBN, CBN_CR and CBN_Std, following Section 5 of [34]. In theory CBN, we introduce all the necessary relations pertaining to $\beta\delta$-reduction (where δ refers to the presence of the so-called δ rules for reducing constants application.) – one-step left reduction, one-step reduction, parallel reduction, their reflexive-transitive clo-

[4] But see Section 8 from the main paper for a discussion of other means for handling such definitions uniformly.

sures, the associated CBN equational theory, etc. – as inductive definitions, and then prove several basic facts about them, such as fresh induction and inversion rules. The number-of-free-occurrences operator $\mathsf{no} : \mathbf{term} \to \mathbf{var} \to \mathbb{N}$ discussed in Section 4 plays an important role in the development, where a measure based on it needs to be assigned to parallel reduction (this measure roughly indicates the number of redexes available in parallel in one step). In CBN_CR, we prove the call-by-name Church-Rosser theorem (a prerequisite for [34]). In CBN_Std, we prove the call-by-name standardization theorem, (stated as Thm. 1 on page 146 in [34]) and its corollaries.

HOAS representation of λ-calculus into itself. This is the content of the theory HOAS. The first part of this theory deals with the representation of syntax, and has been described in Section 6 of the main paper.

As operational semantics for the object system we have chosen the call-by-name big-step reduction (as defined, e.g., in [34] on page 145, using the function Eval_N) – this is defined in our theory CBN as the relation $\mathsf{Bredn} : \mathbf{Obj.term} \to \mathbf{Obj.term} \to \mathbf{bool}$, with concrete syntax $\Longrightarrow_n$, where the index n is not a number, but a reminder of the "by name" style of reduction. (The choice was not important, we could have chosen any other reduction relation.) This relation is represented in the usual HOAS, LF-style fashion. The difference from LF though is that the simple logical framework considered here, **term**(**metaConst**), does not have judgement mechanisms of its own, and therefore we use the inductive mechanism of Isabelle, which is here the "meta-meta level". Thus, we represent Bredn by an inductively defined relation $\mathsf{MetaBredn} : \mathbf{Meta.term} \to \mathbf{Meta.term} \to \mathbf{bool}$, with concrete syntax $\Longrightarrow_{nM}$, where the index "nM" stands for "by name, Meta". E.g., the β-clause in the definition of Bredn, namely,

$$\frac{X \Longrightarrow_n \mathsf{Obj.Lm}\ y\ Z' \qquad Z'[Y/y] \Longrightarrow_n U''}{\mathsf{Obj.App}\ X\ Y \Longrightarrow_n U''}$$

is captured by the clause

$$\frac{\begin{array}{c} mX \Longrightarrow_{nM} \mathsf{Meta.App\ LM}\ mV' \\ \mathsf{Meta.nf}\ (\mathsf{Meta.App}\ mV'\ mY) \Longrightarrow_{nM} mU'' \end{array}}{\mathsf{Meta.App}\ (\mathsf{Meta.App\ APP}\ mX)\ mY \Longrightarrow_{nM} mU''}$$

where, as usual, we omitted spelling the meta-constant injection operator, Meta.Ct, and where $\mathsf{Meta.nf} : \mathbf{Meta.term} \to \mathbf{Meta.term}$ associates to each meta-level term its $\beta\delta$-normal form if such a normal form exists and is undefined otherwise.

Notice that the meta-level rule does not involve any bindings – this was the whole purpose, to capture the involved bindings implicitly, by the meta-level mechanisms. The object-level action of substituting in Z' a previously bound variable (as $Z'[Y/y]$) is matched by meta-level application $\mathsf{Meta.App}\ mV'\ mY$ in conjunction with normalization – since in this correspondence $\mathsf{Meta.App\ LM}\ mV'$ will be the representation of $\mathsf{Obj.Lm}\ y\ Z'$, mV' will itself be a (meta-level) Lm-abstraction; the purpose of normalizing $\mathsf{Meta.App}\ mV'\ mY$ is therefore taking care of the resulting meta-level β-redex.

For this sample representation, we have chosen to normalize "on the fly", since this matches most faithfully the practice of logical frameworks. (For instance, in (generic) Isabelle [47], $(\lambda\ x.\ X)\ Y$ is a volatile entity, being instantaneously replaced by $X[Y/y]$, the latter being what the user sees.) Another approach would be not normalizing (i.e., not including Meta.nf in the above rule), but then stating the adequacy referring to normal forms – the latter is more common in the LF literature on adequacy [5, 22]. Yet another, newer approach, surveyed in [21], is to normalize at substitution time, via a notion of *hereditary substitution*. These approaches are equivalent, and they reflect the semantic intuition that the β (or $\beta\delta$, $\beta\eta$, etc.) equational theory is acting as the meta-level equality.

The semantic adequacy of the representation is stated as two theorems:
- rep_preserves_Bredn: If $X \Longrightarrow_n X'$, then $\mathsf{rep}\ X \Longrightarrow_{nM} \mathsf{rep}\ X'$.
- rep_reflects_Bredn: If $\mathsf{rep}\ X \Longrightarrow_n mX'$, then there exists X' such that $X \Longrightarrow_n X'$ and $\mathsf{rep}\ X' = mX'$.

In particular, these theorems imply:
- corollary rep_Bredn_iff: $\mathsf{rep}\ \ X \Longrightarrow_{nM} \mathsf{rep}\ \ X'$ holds iff $X \Longrightarrow_n X'$ holds.

CPS transformation of call-by-value to call-by-name. This is the content of the theory CPS. It includes formalization of the discussion from Section 4.2. CPS imports the theory Embed, where the two-sorted value-based syntax is embedded in the (single-sorted) standard syntax. We defined two combinators, $\mathsf{combIC} : \mathbf{term} \to \mathbf{term}$, read "the identity-continuation combinator", and $\mathsf{combAC} : \mathbf{term} \to \mathbf{term} \to \mathbf{term}$, read "the application-continuation combinator", corresponding to the right-hand sides of clauses (3) and (4) from Section 4.2, by making choices of fresh variables (via the Isabelle Hilbert choice) and then showing that these choices are irrelevant. Thus, the aforementioned clauses appear in the scripts (in lemma cps_simps) as $\mathsf{cps}\,(\mathsf{InVl}\ Xvl) = \mathsf{combIC}\,(\mathsf{cps}_{\mathsf{vl}}\,Xvl))$ and $\mathsf{cps}\,(\mathsf{App}\,X\,Y) = \mathsf{combAC}\,(\mathsf{cps}\ X)\,(\mathsf{cps}\ Y)$. The "built-in" freshness reflection and injectiveness of the morphism are also formalized, as lemmas cps_reflects_fresh and cps_inj.

A more technically involved example – connection with the de Bruijn representation. This example has not been described in the main body of the paper, so we enter more details here.

We use the following references:
-[a] P.-L. Curien, Categorical combinators. Information and Control, 69(1-3):188–254, 1986.
-[b] H. Ohtsuka. A proof of the substitution lemma in de Bruijn's notation. Inf. Process. Lett., 46(2):63–66, 1993.

De Bruijn indexes are a standard way to represent λ-calculus for efficient manipulation / implementation purposes. The set **dB**, of *de Bruijn terms*, ranged over by K, L, M, is defined by the following grammar, where n ranges over $\mathbb{N}$:

$$K \ ::= \ \mathsf{VAR}\ n \mid \mathsf{APP}\ K\ L \mid \mathsf{LM}\ K$$

There are no variable bindings involved – rather, the index n of a de Bruijn variable $\mathsf{VAR}\ n$ indicates the "distance" between the given occurrence and its LM-binder (if any), i.e., the number of LM-operators interposed between that occurrence and its binder. Thus, e.g., the λ-term $\mathsf{Lm}\ x\ (\mathsf{Lm}\ y\ (\mathsf{App}\ y\ x))$ corresponds to the de Bruijn term $\mathsf{LM}\ (\mathsf{LM}\ (\mathsf{APP}\ (\mathsf{VAR}\ 0)\ (\mathsf{VAR}\ 1)))$.

As it stands, this correspondence is not perfect, since, on one hand, there are λ-terms with free variables, and, on the other, there are de Bruijn terms with dangling indexes, i.e., indexes that fail to point to a valid binder (e.g., $\mathsf{VAR}\ 0$, or $\mathsf{LM}\ (\mathsf{VAR}\ 1)$). A more serious discrepancy is that the processes of binding variables in terms are different: using Lm, one can choose any variable x to be bound, while using LM one binds the "default" de Bruijn variables waiting to be bound, namely, those with dangling level 0.

A closer mathematical connection, allowing one to actually relate de Bruijn versions of standard results with the "official" λ-versions (as is done, e.g., in [b]) can be achieved via parameterizing by variable orderings, as discussed in [a].

We model such orderings as injective valuations of variables into numbers, i.e., injective elements of $\mathbf{val} = (\mathbf{var} \to \mathbb{N})$. Let **sem**, the set of semantic values, ranged over by s, be $\{\rho \in \mathbf{val}.\ \mathsf{inj}\ \rho\} \to \mathbf{dB}$, where $\mathsf{inj} : \mathbf{val} \to \mathbf{bool}$ is the predicate stating that a valuation is injective. Then λ-terms are interpreted as de Bruijn terms by a map $\mathsf{toDB} : \mathbf{term} \to \mathbf{sem}$, read "to de Bruijn" as follows.

The clauses for variables and application are the obvious ones:
- (1) $\mathsf{toDB}\ x = \lambda\rho.\ \mathsf{VAR}(\rho\ x)$;
- (2) $\mathsf{toDB}\ X\ Y = \lambda\rho.\ \mathsf{APP}\ (\mathsf{toDB}\ X\ \rho)\ (\mathsf{toDB}\ Y\ \rho)$.

To interpret λ-abstraction, we first define $\mathsf{mkFst} : \mathbf{var} \to \mathbf{val} \to \mathbf{val}$, read "make first":
- $\mathsf{mkFst}\ x\ \rho = \lambda\ y.$ if $y = x$ then 0 else $(\rho\ y + 1)$.

(Thus, x is "made first" by assigning it the first position, 0, in ρ, and by making room for it through shifting the values of all the other variables.) Now, we set
- (3) $\mathsf{toDB}\ (\mathsf{Lm}\ x\ X) = \lambda\rho.\ \mathsf{LM}(\mathsf{toDB}\ X\ (\mathsf{mkFst}\ x\ \rho))$.

(Thus, in order to map $(\mathsf{Lm}\ x)$-abstractions to LM-abstractions, x is initially "made first", i.e., mapped to 0, the only value which LM "knows" how to bind.)

The above clauses are essentially Definition 2.3 on page 192 in [a] (under a slightly different notation). There are some similarities between these clauses and those from Problem I, but there are also some major differences, notably the interpretation of λ-abstractions and the restricted space of valuations here, which prevent us from reducing one problem to the other.

As usual, to make the definition rigorous, we ask the freshness and substitution questions.

What can we infer about x versus $\mathsf{toDB}\ X$ if we know that x is fresh for X? The answer to this is similar to that from Problem I: we can infer obliviousness to x of the interpretation:
- (4) $\mathsf{fresh}\ x\ X \Longrightarrow (\forall\rho\ \rho'.\ \rho =_x \rho' \Longrightarrow \mathsf{toDB}\ X\ \rho = \mathsf{toDB}\ X\ \rho')$.

As for the substitution question, the answer is given by a lemma stated in loc. cit.:
- (5) $\mathsf{toDB}\,(X[Y/y]) = \lambda\rho.\,(\mathsf{toDB}\,X)\,(\mathsf{mkFst}\,y\,\rho)\,[(\mathsf{toDB}\,Y\,\rho)/0]^b$,

where $\_[\_/\_]^b : \mathbf{dB} \to \mathbf{dB} \to \mathbb{N} \to \mathbf{dB}$ is de Bruijn substitution (i.e, plain FO-substitution):

— $(\mathsf{VAR}\ n)[M/m]^b =$ if $n = m$ then M else $\mathsf{VAR}\ n$,
— $(\mathsf{APP}\ K\ L)[M/m]^b = \mathsf{APP}\ (K[M/m]^b)\ (L[M/m]^b)$,
— $(\mathsf{LM}\ K)[M/m]^b = \mathsf{LM}\ (K[M/m]^b)$.

(Thus, the toDB treatment of substitution is similar to that of λ-abstraction: x is 0-indexed and then 0 is de Bruijn-substituted.)

Again, to have clauses (1)-(3) rigorously justified and clauses (4), (5) proved, one needs to check that the resulted model is an FSb-model. Unlike in the previous examples, here there are some complications, in that some of the FSb-model clauses simply do not hold on the whole space **sem**.

Fortunately however, a technical closure condition fixes this problem. Namely, if we take
$\mathbf{sem}' = \{s \in \mathbf{sem}.\ \forall\sigma : \mathbb{N} \to \mathbb{N},\ \rho \in \mathbf{val}.\ \mathsf{inj}\ \sigma\ \wedge\ \mathsf{inj}\ \rho \Longrightarrow s\ (\sigma \circ \rho) = G\ \sigma\ (\mathsf{toDB}\ X\ \rho)\}$,
where $G : (\mathbb{N} \to \mathbb{N}) \to \mathbf{dB} \to \mathbf{dB}$ is a map whose definition we give below, then $\mathbf{sem}'$, with the FSb-structure given by clauses (1)-(5), indeed turns out to be an FSb-model – checking this is relatively easy, but does rely on some de Bruijn arithmetic.

The definition of G (having little importance beyond its technical role in the invariant that helps defining toDB), goes as follows:
- $G\ \sigma\ K = \mathsf{vmapDB}\ (\mathsf{cut}\ \sigma)\ 0\ K$, where
- $\mathsf{vmapDB} : (\mathbb{N} \to \mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbf{dB} \to \mathbf{dB}$, read "variable map", is the natural de Bruijn analogue of, say, list map (also taking into account, like any "natural" de Bruijn function, the LM-depth, recursively):

— $\mathsf{vmapDB}\ \sigma\ n\ (\mathsf{VAR}\ i) = \mathsf{VAR}\ (\sigma\ n\ i)$,
— $\mathsf{vmapDB}\ \sigma\ n\ (\mathsf{APP}\ K\ L) = \mathsf{APP}\ (\mathsf{vmapDB}\,\sigma\,K)\ (\mathsf{vmapDB}\,\sigma\,L)$;
— $\mathsf{vmapDB}\ \sigma\ (\mathsf{LM}\ K) = \mathsf{LM}\ (\mathsf{vmapDB}\ \sigma\ (n+1)\ K)$;
- and $\mathsf{cut} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ is defined by
$\mathsf{cut}\ \sigma\ n\ i =$ if $i < n$ then i else $\sigma\ (i - n) + n$.

We conjecture that the difficulty we encountered with constructing an FSb-model for this example would be matched by a corresponding difficulty in making the definitions go through in an ad-hoc approach, where one would eventually have to discover a similar invariant (again, compared to our approach, with the disadvantage of having to do this discovery by repeated trial and error in the middle of inductive proofs, not knowing in advance what properties need to hold). Unfortunately, we have no way of testing this conjecture, as ours seems to be the first rigorous (let alone formal) treatment of this construction from the literature. In [a], one employs quasi-terms rather than terms, and α-equivalence is not even defined until after defining the above de Bruijn interpretation, and then its definition is non-standard: α-equivalence is taken to be the very kernel of this interpretation. Moreover, the substitution lemma is stated in [a] without a proof.

This example has been formalized as the theory C_deBruijn from [4]. The scripts follow quite closely the notations we used above. We import theory Lambda, the existing formalization of the de Bruijn representation from the Isabelle library. Then we develop some de Bruijn arithmetics facts, after which we proceed with the interpretation map toDB by means of the model toDB_MOD. (This naming convention, having the name of the model be the name of the desired map followed by "_MOD" is observed in all our formalized examples.) This time, since there are some nontrivial computations involved in the verification of the clauses, we prefer to first define the model components separately (operators ggWls, ggVar etc. in the scripts). Lemmas toDB_simps, toDB_preserves_fresh and toDB_subst are the end-products of the definition, i.e. correspond to clauses (1)-(5). (For these end-products, we also observe the same naming pattern, as seen here and in the HOAS case, for all our examples.)