Binding Structures as an Abstract Data Type

Wilmer Ricciotti

IRIT – Institut de Recherche en Informatique de Toulouse
Université de Toulouse
Wilmer.Ricciotti@irit.fr

Abstract. A long line of research has been dealing with the representation, in a formal tool such as an interactive theorem prover, of languages with binding structures (e.g. the lambda calculus). Several concrete encodings of binding have been proposed, including de Bruijn dummies, the locally nameless representation, and others. Each of these encodings has its strong and weak points, with no clear winner emerging. One common drawback to such techniques is that reasoning on them discloses too much information about what we could call "implementation details": often, in a formal proof, an unbound index will appear out of nowhere, only to be substituted immediately after; such details are never seen in an informal proof. To hide this unnecessary complexity, we propose to represent binding structures by means of an abstract data type, equipped with high level operations allowing to manipulate terms with binding with a degree of abstraction comparable to that of informal proofs. We also prove that our abstract representation is sound by providing a de Bruijn model.

1 Introduction

The techniques for reasoning on languages with binders are a very popular topic in both programming and logic ([20,5]). Especially in logic, the choice of a representation of binding structures is one of the most significant issues when formalizing the metatheory of a programming language. Over the years, a number of different styles have been proposed to deal with binding, roughly divided in two different categories: first order encodings, also called *concrete* encodings, and higher-order encodings like higher-order abstract syntax (HOAS). In interactive theorem provers based on a strong type theory, like Coq, Matita, or Agda, trivial implementations of HOAS by means of inductive types are rejected because they do not satisfy the positivity checks required by those systems to ensure consistency and, more importantly, adequacy concerns related to the appearance of *exotic terms* arise; thus, concrete encodings are more usually employed (notable exceptions include two-level approaches [8] and weak HOAS [10]).

Concrete encodings include some of the best known styles, like the de Bruijn nameless encoding [16] (which represents variables using indices pointing to the binder that declares them), the locally nameless encoding (a variant of the de Bruijn encoding where only bound variables are represented by indices, whereas free variables still use names) and the canonically named encoding of

J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 762–786, 2015.
DOI: 10.1007/978-3-662-46669-8_31

Pollack and Sato [23], where a bound variable is represented by means of a name that is programmatically chosen depending on the structure of the term within scope. All of these styles are described as *canonical* because terms that are equal up to α-renaming are identified. We have studied these styles in [17,3] and drawn a comparison in [22].

Our experience tells us that every concrete encoding has its own disadvantages, but more importantly that all of them share one problem: they force the formalizer to deal with the intricacy of the inner representation of binding, something that in an informal proof is never seen. In a formal proof based on a concrete approach, it is only a matter of time before nameless dummies, lifting operations, or name choosing operations come to the surface.

We should ask ourselves whether this inconvenience is inherent to the concrete representation of binding. Our understanding is that very often (if not *always*) the internal representation of binding must be treated explicitly because of the lack of an infrastructure designed to keep it hidden. We have very good access to the *implementation* but, crucially, we lack an *abstract* view on binding.

This paper describes a project which aims at representing binding only by means of abstract operations (similar to the ones employed in a pencil-and-paper proof), keeping the implementation details hidden from the user. More precisely, we will represent the terms of the object language as an abstract data type, which can only be manipulated by means of the operations and logical properties provided by its module. Based on this, we will prove two kinds of results:

- soundness properties, showing the existence of an implementation, or model, of the abstract data type which validates all the stated properties;
- theorems about the object language (e.g.: subject reduction), whose are carried out within the abstract data type, without resorting to any property specific to the model.

While other proposals to treat binding structures axiomatically exist ([11,19]), in this paper we will address some topics that have been neglected, particularly the treatment of inductively defined predicates over binding structures. All of the proofs presented here have been proved valid in the Matita theorem prover.[1]

The paper is structured as follows: Section 2 presents an abstract data type representing the term language of the simply typed lambda calculus; in Section 3 we provide an implementation of the abstract data type in the form of a locally nameless model; after recalling the problem of induction principle strengthening in the context of typing rules (Section 4), we extend our abstract data type to the level of type systems (Section 5) and beta reduction (Section 6), showing that the technique is sufficiently powerful to carry out common proofs like weakening and subject reduction; finally Section 7 concludes.

[1] The Matita formalization can be found at
`http://www.irit.fr/~Wilmer.Ricciotti/publications.html` .

2 An Abstract View of Binding

We present in this section a collection of abstract data types describing a simple language with binding: the simply typed lambda calculus (or, for brevity, $\boldsymbol{\lambda}_{\to}$). Similarly to the axiomatization in [11], the operations working on our data type include a set of opaque constants acting as "constructors" for the terms of the language and a principle allowing to define functions by structural recursion on the terms. However, instead of a primitive substitution function, we provide facilities to form contexts (terms with holes) and apply them to variables, which we regard as more basic. An operation to retrieve the list of the free variables in a term or context is also given. In addition, properties asserting the computational behavior of the aforementioned operations are provided.

Signature. Our module defines the abstract data types of $\boldsymbol{\lambda}_{\to}$ as follows:

$$
\begin{array}{ll}
tp & : \mathbf{Type} \\
\mathsf{Atom} & : tp \\
\mathsf{Arr} & : tp \Rightarrow tp \Rightarrow tp
\end{array}
$$

$$
\begin{array}{ll}
(\Lambda_i)_{i\in\mathbb{N}} & : \mathbf{Type} \\
\mathsf{Par} & : \mathbb{A} \Rightarrow \Lambda_0 \\
\mathsf{App} & : \Lambda_0 \Rightarrow \Lambda_0 \Rightarrow \Lambda_0 \\
\mathsf{Lam} & : \mathbb{A} \Rightarrow tp \Rightarrow \Lambda_0 \Rightarrow \Lambda_0
\end{array}
$$

$$
\begin{array}{ll}
\nu & : \mathbb{A} \Rightarrow \Lambda_i \Rightarrow \Lambda_{i+1} \\
-\lceil - \rceil & : \Lambda_{i+1} \Rightarrow \mathbb{A} \Rightarrow \Lambda_i
\end{array}
$$

$$
\mathrm{FV} \quad : (\Lambda_i)_{i\in\mathbb{N}} \Rightarrow \mathit{list}\ \mathbb{A}
$$

$$
\begin{array}{ll}
\mathcal{R}_{\Lambda_0} & : \forall T : \Lambda_0 \Rightarrow \mathbf{Type}, C : \mathit{list}\ \mathbb{A}. \\
& \quad (\forall x : \mathbb{A}. T\ (\mathsf{Par}\ x)) \Rightarrow \\
& \quad (\forall u, v : \Lambda_0. T\ u \Rightarrow T\ v \Rightarrow T\ (\mathsf{App}\ u\ v)) \Rightarrow \\
& \quad \left(\begin{array}{l} \forall x : \mathbb{A}, \sigma : tp, v : \Lambda_1. x \notin \mathrm{FV}(v), C \Rightarrow T\ (v\lceil x \rceil) \\ \quad \Rightarrow T\ (\mathsf{Lam}\ x\ \sigma\ (v\lceil x \rceil)) \end{array} \right) \Rightarrow \\
& \quad \forall u : \Lambda_0. T\ u
\end{array}
$$

We call this presentation of binding *ostensibly named* because at the external level we always manipulate terms as entities containing names, including bound variables: we never see bound variables represented as nameless dummies, or pointers to their binder. The concrete implementation of binding structures may or may not use names, but this is hidden from the user.

The set of types tp of the simply typed lambda calculus is of no particular interest and is here provided for reference only: it is the free algebra obtained from the zeroary constructor of the atomic type Atom and the binary constructor of arrow (function) types Arr. Types of the simply typed lambda calculus will be denoted by σ,τ,...

Λ_i will represent the type of terms with i holes, or i-ary contexts. Zeroary contexts are taken as the terms of the calculus. We will denote terms and contexts alike by $u, v, \ldots$. The type of names $\mathbb{A}$ is an arbitrary infinite type with decidable equality. We assume the existence of an operation $\varphi : \mathit{list}\ \mathbb{A} \Rightarrow \mathbb{A}$ allowing us to choose a name which is *fresh* with respect to any given finite list (i.e., $\varphi(C) \notin C$ holds for all finite lists of names C).

The constructors of terms include Par, encapsulating a name to represent a free variable or parameter, applications App, and lambda abstractions Lam. Just as in informal syntax, lambda abstractions bear a type and bind a name inside a subterm. For example, the identity function $\lambda x : \mathsf{Atom}.x$ is expressed as

$$\mathsf{Lam}\ x\ \mathsf{Atom}\ (\mathsf{Par}\ x)$$

provided that x is a name in $\mathbb{A}$.

Crucially, to put our representation to some use we need to be able to talk about *contexts*. Two operations ν and $-\lceil - \rceil$ (respectively *variable closing* or *context formation* and *variable opening* or *context application*) are provided to build and apply contexts: $\nu x.u$ substitutes a hole for all (free) occurrences of $\mathsf{Par}\ x$ in u, thus increasing its arity, whereas $u\lceil x \rceil$ replaces the last created hole in u with $\mathsf{Par}\ x$, decreasing its arity. It is worth noting that, since it cancels out a free variable, ν acts like a binder (the notation was chosen in analogy to the "new channel" operator of the π-calculus). Closing and opening can be combined (in this order) to rename a variable: we will use the following special notation for *variable renaming*:

$$u\,\langle {}^{y}/_{x} \rangle \triangleq (\nu x.u)\lceil y \rceil$$

If $p = (x, y)$ is a pair of variable names, we will write $u\,\langle p \rangle$ for $u\,\langle {}^{y}/_{x} \rangle$. This notation is further extended to vectors: if $\overrightarrow{p} = [p_1, \ldots, p_n]$ is a vector of pairs, we will write $u\,\langle \overrightarrow{p} \rangle$ for $u\,\langle p_1 \rangle \cdots \langle p_n \rangle$.

An abstract operation FV takes as input a term or a context and returns the list of free names used in that term or context. Lastly, $\mathcal{R}_{\Lambda_0}$ is a primitive recursion principle over terms. Recursion principles on inductive types have a well defined shape, which is followed by $\mathcal{R}_{\Lambda_0}$, except for the Lam case, which provides a special treatment for the bound variable. We will make this clear in the following paragraphs.

Properties of terms and contexts. The following properties of terms and context forming operations are assumed:

$$\begin{aligned}
x\,\langle {}^{y}/_{x} \rangle &= y \\
z\,\langle {}^{y}/_{x} \rangle &= z && \text{if } z \neq x \\
(\mathsf{App}\ u\ v)\,\langle {}^{y}/_{x} \rangle &= \mathsf{App}\ (u\,\langle {}^{y}/_{x} \rangle)\ (v\,\langle {}^{y}/_{x} \rangle) \\
(\mathsf{Lam}\ z\ \sigma\ u)\,\langle {}^{y}/_{x} \rangle &= \mathsf{Lam}\ z\ \sigma\ (u\,\langle {}^{y}/_{x} \rangle) && \text{if } z \neq x, y
\end{aligned}$$

$$\begin{aligned}
u\,\langle {}^{x}/_{x} \rangle &= u && (*) \\
\nu x.(u\lceil x \rceil) &= u && \text{if } x \notin \mathrm{FV}(u) \\
\mathsf{Lam}\ x\ \sigma\ (u\lceil x \rceil) &= \mathsf{Lam}\ y\ \sigma\ (u\lceil y \rceil) && \text{if } x, y \notin \mathrm{FV}(u)
\end{aligned}$$

The first four lines fall logically into the same group, they describe the computational behaviour of renaming. The last line is also remarkable, since it expresses the fact that Λ_0 is canonical, i.e. α-convertible terms are provably equal. The second-to-last property expresses a sort of an "η-equivalence" on contexts: opening a context and then closing it with respect to the same variable yields the original context, provided that the variable involved is fresh.

The property marked with (*) has a special status since, assuming a suitable induction principle on Λ_0, it could be proved from the other properties whenever u is a term; we will provide such an induction principle on terms, but if we want (*) to be valid not just for terms, but for proper contexts as well, we will still have to assume it as part of the abstract data type.

Recursion. We employ contexts to express a recursion principle $\mathcal{R}_{\Lambda_0}$ for Λ_0, allowing us to define functions over terms by structural recursion.

The lines 2–5 of the type of $\mathcal{R}_{\Lambda_0}$ express the types of the arguments of the principle which will provide its behaviour in the Par, App, and Lam case. To better understand how $\mathcal{R}_{\Lambda_0}$ works, we use it to define the usual operation of substitution of terms for free variables. Informally, substitution is often defined as follows:

$$u\,[v/x] \triangleq \begin{cases} (\mathsf{Par}\ x)\,[v/x] & = v \\ (\mathsf{Par}\ y)\,[v/x] & = \mathsf{Par}\ y \qquad \text{if } x \neq y \\ (\mathsf{App}\ u_1\ u_2)\,[v/x] & = \mathsf{App}\ (u_1\,[v/x])\ (u_2\,[v/x]) \\ (\mathsf{Lam}\ y\ \sigma\ u_1)\,[v/x] & = \mathsf{Lam}\ y\ \sigma\ (u_1\,[v/x]) \qquad \text{if } y \notin \{x\} \cup \mathrm{FV}(v) \end{cases}$$

This is not a regular pattern matching over an inductive type: while the Par and App cases do not look special (and the same can be said about the types of the associated clauses in $\mathcal{R}_{\Lambda_0}$) the Lam case hides an implicit α-conversion in order to make the bound variable different from both x and any free variable occurring in v, to prevent variable capture. More generally, an effective recursion principle over lambda abstractions should allow us to retrieve, for a bound variable, a name that is fresh with respect to an arbitrary list: for this reason, we add a "freshness context" C to the principle $\mathcal{R}_{\Lambda_0}$ (similarly to what is done in Nominal Isabelle [25]).

Thus, we can express the substitution operation as a structurally recursive function over ostensibly named terms as follows.

Definition 1 (substitution). *For all terms u, v and parameter names x, the function subst is defined as follows:*

$$\begin{aligned} \mathit{subst}\ u\ x\ v \triangleq \mathcal{R}_{\Lambda_0}\ & (\lambda\_.\Lambda_0)\ (x, \mathrm{FV}(v)) \\ & (\lambda y.\mathit{if}\ (x = y)\ \mathit{then}\ v\ \mathit{else}\ (\mathsf{Par}\ y)) \\ & (\lambda u_1, u_2, r_1, r_2.\mathsf{App}\ r_1\ r_2) \\ & (\lambda y, \sigma, u^*, \_, r^*.\mathsf{Lam}\ y\ \sigma\ r^*)\ u \end{aligned}$$

We will use the notation $u\,[v/x]$ as a short form for $\mathit{subst}\ u\ x\ v$.

In this definition, variables r_1,r_2,r^* are used to represent the result of recursion on the subterms u_1,u_2,$u^*\lceil y \rceil$ respectively. The abstraction operation is special: the recursion principle unpacks it as $\mathsf{Lam}\ y\ \sigma\ (u^*\lceil y \rceil)$, where u^* is a unary context and y is taken to be fresh with respect to the list $x, \mathrm{FV}(v)$ we provided as an argument and also with respect to $\mathrm{FV}(u^*)$ (a proof that $y \notin x, \mathrm{FV}(v), \mathrm{FV}(u^*)$ is also provided as the underscore "_" argument, that is irrelevant to the definition of the substitution, but may be employed in a proof of correctness).

Properties of FV. For FV, we assume that the following properties hold:

$$\mathrm{FV}(\mathsf{Par}\ x) = [x]$$
$$\mathrm{FV}(\mathsf{App}\ u\ v) = \mathrm{FV}(u) \cup \mathrm{FV}(v)$$
$$\mathrm{FV}(\mathsf{Lam}\ x\ \sigma\ u) = \mathrm{FV}(\nu x.u)$$

$$x \in \mathrm{FV}(\nu y.u) \iff (x \neq y \wedge x \in \mathrm{FV}(u))$$

$$\mathrm{FV}(u\lceil x \rceil) \subseteq \{x\} \cup \mathrm{FV}(u)$$
$$\mathrm{FV}(u) \subseteq \mathrm{FV}(u\lceil x \rceil)$$

Properties of Recursion. Since the recursion principle is part of our ostensibly named interface, it is opaque. This means its algorithmic behaviour must be expressed explicitly. Let *Rec* be short for $\mathcal{R}_{\Lambda_0}\ T\ C\ f_{\mathsf{Par}}\ f_{\mathsf{App}}\ f_{\mathsf{Lam}}$ (where T, C, f_{Par}, f_{App}, f_{Lam} have a suitable type). We will assume that the following properties hold:

$$\mathit{Rec}\ (\mathsf{Par}\ x) = f_{\mathsf{Par}}\ x$$

$$\mathit{Rec}\ (\mathsf{App}\ u\ v) = f_{\mathsf{App}}\ u\ v\ (\mathit{Rec}\ u)\ (\mathit{Rec}\ v)$$

$$\begin{aligned}&\forall U : (\forall u : \Lambda_0.T\ u \Rightarrow \mathbf{Type}).x \notin \mathrm{FV}(u) \Rightarrow\\ &\quad (\forall y, H_y.U\ (u\lceil y \rceil)\ (f_{\mathsf{Lam}}\ C\ y\ \sigma\ u\ H_y\ (\mathit{Rec}\ (u\lceil y \rceil)))) \Rightarrow\\ &\quad U\ (\mathsf{Lam}\ x\ \sigma\ (u\lceil x \rceil))\ (\mathit{Rec}\ (\mathsf{Lam}\ x\ \sigma\ (u\lceil x \rceil)))\end{aligned}$$

The first two lines are equations stating that on parameters and applications, $\mathcal{R}_{\Lambda_0}$ behaves as a normal recursion operator on an inductive type: it can be rewritten as an application of the appropriate branch (f_{Par} or f_{App}) to the arguments of the constructor and, in the case of App, to the result of recursion on its subterms. The last line expresses the behaviour in the Lam case, which is more complicated: if we allowed the same scheme as with Par and App we could take $f_{\mathsf{Lam}} = \lambda y, \_, \_, \_, \_.y$ and use $\mathcal{R}_{\Lambda_0}$ to expose the variable bound by Lam as follows:

$$\begin{aligned}&\mathit{Rec}\ (\mathsf{Lam}\ x\ \sigma\ (u\lceil x \rceil))\\ &\quad = (\lambda y, \_, \_, \_, \_.y)\ x\ \sigma\ u\ H\ (\mathit{Rec}\ (u\lceil x \rceil))\\ &\quad = x\end{aligned}$$

(where H is any proof that $x \notin \mathrm{FV}(u), C$). As it turns out, this equation would make it possible to look into the name bound by Lam: this would in turn enable

us to discriminate abstractions in terms of their bound variables, which is clearly inconsistent with the α-equivalence hypothesis.

The fact that we should not be able to extract naming information from binders prevents us from expressing the computational behaviour of the recursion principle explicitly in the Lam case. The property we stated is a "constrained rewriting principle" which does not allow, in general, to compute the result of a structurally recursive function in the $\mathsf{Lam}\ x\ \sigma\ (u\lceil x\rceil)$ case. However, if we employ it in a proof whose goal involves such a function, we will get a new variable y, together with a proof H_y that $y \notin C, \mathrm{FV}(u)$ (both universally quantified in the property) and the goal will be rewritten in such a way that we have the illusion that $\mathsf{Lam}\ x\ \sigma\ (u\lceil x\rceil)$ has been renamed to $\mathsf{Lam}\ y\ \sigma\ (u\lceil y\rceil)$ and a computation step on the recursive definition has occurred. Notice the difference with Nominal Isabelle, which does not allow one to define functions exposing bound variables: in contrast, not only can we write a function that given a term $\mathsf{Lam}\ x\ \sigma\ (u\lceil x\rceil)$ (with $x \notin \mathrm{FV}(u)$) will return the tuple $\langle y, \sigma, u\lceil y\rceil\rangle$ (for some $y \notin \mathrm{FV}(u)$), but we can also prove that putting together those items we obtain the original term, i.e. $\mathsf{Lam}\ x\ \sigma\ (u\lceil x\rceil) = \mathsf{Lam}\ y\ \sigma\ (u\lceil y\rceil)$.

2.1 Some Derived Properties

Since $\mathcal{R}_{\Lambda_0}$ provides case analysis, it can be used to prove many of the properties that we expect from Par, App and Lam as constructors. Among these, an important one concerns injectivity and discrimination:

Lemma 2. *The following properties hold:*

- *if* $\mathsf{Par}\ x = \mathsf{Par}\ y$, *then* $x = y$;
- *if* $\mathsf{App}\ u_1\ u_2 = \mathsf{App}\ v_1\ v_2$, *then* $u_1 = v_1$ *and* $u_2 = v_2$;
- *if* $\mathsf{Lam}\ x\ \sigma\ u = \mathsf{Lam}\ x\ \tau\ v$, *then* $\sigma = \tau$ *and* $u = v$;
- *different constructors always yield different terms:* $\mathsf{Par}\ x \neq \mathsf{App}\ u\ v$, $\mathsf{App}\ u_1\ u_2 \neq \mathsf{Lam}\ x\ \sigma\ v$, $\mathsf{Par}\ x \neq \mathsf{Lam}\ y\ \sigma\ v$.

Proof (sketch). McBride's generic proof for inductive types ([13]) only requires pattern matching and reasoning by cases: it is thus easy to adapt it to our abstract data type.

One thing to notice is that the injectivity property for Lam requires the bound variable to be the same in both abstractions. If this is not the case, the two should be made equal by α-equivalence before applying injectivity.

Other properties cannot be proved as easily. One reason for this is that the constrained rewriting approach for the recursion principle is not completely satisfying: we are giving up the possibility to compute directly the result of functions defined by means of $\mathcal{R}_{\Lambda_0}$ because some of those functions (like those that try to expose bound variables) are ill-behaved. But other functions, like substitution, are well-behaved: we expect to know that whenever the bound variable x is not in $y, \mathrm{FV}(v)$, then the following equality holds:

$$(\mathsf{Lam}\ x\ \sigma\ u)\,[v/y] = \mathsf{Lam}\ x\ \sigma\ (u\,[v/y])$$

As a matter of fact, the equality holds in the context of our abstract data type. To prove it, we need the following induction principle:

Theorem 3. *The following induction principle $\mathcal{E}_{\Lambda_0}$ is provable:*

$$\begin{aligned}\mathcal{E}_{\Lambda_0} : {} & \forall P : \Lambda_0 \Rightarrow \mathbf{Prop}.\\ & (\forall x.P\ (\mathsf{Par}\ x)) \Rightarrow\\ & (\forall u_1, u_2.P\ u_1 \Rightarrow P\ u_2 \Rightarrow P\ (\mathsf{App}\ u_1\ u_2)) \Rightarrow\\ & (\forall x, \sigma, v.x \notin \mathrm{FV}(v) \Rightarrow\\ & \quad (\forall y.y \notin \mathrm{FV}(v) \Rightarrow P\ (v\lceil y\rceil)) \Rightarrow P\ (\mathsf{Lam}\ x\ \sigma\ (v\lceil x\rceil))) \Rightarrow\\ & \forall u.P\ u\end{aligned}$$

Proof (sketch). We assume the branches of $\mathcal{E}_{\Lambda_0}$ as hypotheses and we subsequently prove $\forall \overrightarrow{p}.P\ (u\,\langle\overrightarrow{p}\rangle)$ using the recursion principle $\mathcal{R}_{\Lambda_0}$ on u. This implies $P\ u$ by instantiating $\overrightarrow{p}$ with the empty list $[]$.

That $\mathcal{E}_{\Lambda_0}$ can be proved using $\mathcal{R}_{\Lambda_0}$ should not be surprising, as in type theory recursion and induction are intimately related. Actually, when we ignore the computational content of $\mathcal{R}_{\Lambda_0}$ and only consider its type, we see that its form is very similar to that of an induction principle (where the result of recursion on a subterm corresponds to the induction hypothesis). Besides the fact that $\mathcal{E}_{\Lambda_0}$ returns a proof of a proposition rather than an object of a given type[2], the biggest difference between $\mathcal{R}_{\Lambda_0}$ and $\mathcal{E}_{\Lambda_0}$ is that the latter, in the Lam case, provides a different induction hypothesis for all possible choices of the bound variable, while $\mathcal{R}_{\Lambda_0}$ only considers a single variable (which one is not under our control: at most, we can require that it should be sufficiently fresh): this is usually enough to define a recursive function, but not in proofs by induction like the following ones. We will come back to the topic of universally quantified induction hypotheses in Sections 4 and 5.

Lemma 4. *For all u, x, v, $\mathrm{FV}(u\,[v/x]) \subseteq \mathrm{FV}(u) \cup \mathrm{FV}(v)$.*

Lemma 5. *For all u, x, v, if $\overrightarrow{p}$ is a list of pairs of variable names not in $x, \mathrm{FV}(v)$, then*

$$u\,[v/x]\,\langle\overrightarrow{p}\rangle = (u\,\langle\overrightarrow{p}\rangle)\,[v/x]$$

Both proofs are by induction on u, with Lemma 5 using Lemma 4 in the Lam case. These two properties are what we need to prove the commutation property for subst in the Lam case.

Fact 6 *If $x \notin y, \mathrm{FV}(v)$, then $(\mathsf{Lam}\ x\ \sigma\ u)\,[v/y] = \mathsf{Lam}\ x\ \sigma\ (u\,[v/y])$*

Proof. Notice that $u = u\,\langle x/x\rangle = (\nu x.u)\lceil x\rceil$. Thus we can apply the constrained rewriting property: this leaves us with the goal

$$\mathsf{Lam}\ z\ \sigma\ ((u\,\langle z/x\rangle)\,[v/y]) = \mathsf{Lam}\ x\ \sigma\ (u\,[v/y])$$

[2] Induction principles are usually given for **Prop**; however we could as well derive a similar principle for **Type**, at no additional formalization cost.

where z is not free in $\nu x.u$ or $y, \mathrm{FV}(v)$. The goal can be proved by rewriting the left-hand side as follows:

$$\begin{aligned}
&\mathsf{Lam}\ z\ \sigma\ ((u\,\langle z/x\rangle)\,[v/y]) \\
=\ &\mathsf{Lam}\ z\ \sigma\ (u\,[v/y]\,\langle z/x\rangle) && \text{(by Lemma 5)} \\
=\ &\mathsf{Lam}\ z\ \sigma\ ((\nu x.u\,[v/y])\lceil z\rceil) && \text{(by def. of renaming)} \\
=\ &\mathsf{Lam}\ x\ \sigma\ ((\nu x.u\,[v/y])\lceil x\rceil) && \text{(by } \alpha\text{-equivalence)} \\
=\ &\mathsf{Lam}\ x\ \sigma\ (u\,[v/y]\,\langle x/x\rangle) && \text{(by def. of renaming)} \\
=\ &\mathsf{Lam}\ x\ \sigma\ (u\,[v/y]) && \text{(by axiom)}
\end{aligned}$$

where the α-equivalence holds because $z \notin \nu x.u\,[v/y]$, which is easily proved using Lemma 4.

It is worth noting that the recursion principle in Gordon and Melham's axiomatization ([11]) handles abstractions differently, allowing direct computation of functions employing it. This, however, comes at the price of requiring explicit treatment of variable renaming in the function definition. As a consequence, if we expressed substitution in that style, the following computation property would trivially hold in the Lam case:

$$(\mathsf{Lam}\ x\ \sigma\ u)\,[v/y] = \mathsf{Lam}\ \varphi(y, \mathrm{FV}(v))\ \sigma\ ((u\,\langle \varphi(y,\mathrm{FV}(v))/x\rangle)\,[v/y])$$

However, this is a weaker property than Fact 6 (which remains provable, with a similar argument as ours).

3 A Locally Nameless Model

A locally nameless representation [9] of a language with binders is a variant of de Bruijn's nameless representation where names are allowed to represent free parameters, but indices are always used to express bound variables. A locally nameless representation of the simply typed lambda calculus can be given as the following *pretm* inductive type of *pre-terms*:

$$\begin{aligned}
&\text{inductive } \mathit{pretm} : \mathbf{Type} \triangleq \\
&\quad \mathsf{var} : \mathbb{N} \Rightarrow \mathit{pretm} \\
&\quad \mathsf{par} : \mathbb{A} \Rightarrow \mathit{pretm} \\
&\quad \mathsf{app} : \mathit{pretm} \Rightarrow \mathit{pretm} \Rightarrow \mathit{pretm} \\
&\quad \mathsf{abs} : \mathit{tp} \Rightarrow \mathit{pretm} \Rightarrow \mathit{pretm}.
\end{aligned}$$

Notice we use lowercase identifiers to distinguish the constructors of *pretm* from the similar operations discussed in the previous section. The constructor var is used to construct indices and par for named parameters; abs is a nameless abstraction that is used as the counterpart of Lam abstractions, binding an index rather than a named variable: by convention, our indices are zero-based, so that index var k is considered to be bound to the $(k+1)$-th outer abstraction.

In such a representation, indices whose values are too high and thus do not point to any binder are said to be *dangling*: a dangling index is neither a bound

variable nor a free, named parameter, thus it is often an unwanted situation. Most formalizations employing this style adopt a validity predicate on pre-terms that is verified only for real terms, i.e. those that do not contain dangling indices (also called *locally closed*).

However, in our case the type of pre-terms will have a much more substantial value as the interpretation of both terms and n-ary contexts. We regard dangling indices as holes implicitly bound at the outermost level, waiting for a context application to fill a free variable in them.

The following function checks whether a pre-term can be the interpretation of a k-ary context by verifying that the value of all dangling indices is less than k:

$$\text{check } u\ k \triangleq \begin{cases} \text{check } (\mathsf{var}\ n)\ k = \begin{cases} \text{true if } n < k \\ \text{false else} \end{cases} \\ \text{check } (\mathsf{par}\ x)\ k = \text{true} \\ \text{check } (\mathsf{app}\ u_1\ u_2)\ k = (\text{check } u_1\ k) \wedge (\text{check } u_2\ k) \\ \text{check } (\mathsf{abs}\ \sigma\ u_1)\ k = \text{check } u_1\ (k+1) \end{cases}$$

We thus define the interpretation of i-ary ostensibly named contexts in the locally nameless model as the dependent pair associating a pre-term u to the proof that check $u\ i = \text{true}$:

$$[\![\Lambda_i]\!] = ctx\ i \triangleq \Sigma u : pretm.\text{check } u\ i = \text{true}$$

We define algorithmically in the model two contextual operations that are a counterpart to the similar operations of the ostensibly named presentation. They employ a parameter k that is used, in recursive calls, to keep track of the number of abstractions crossed.

$$\nu_k x.u \triangleq \begin{cases} \nu_k x.\mathsf{var}\ n = \begin{cases} \mathsf{var}\ n & \text{if } n < k \\ \mathsf{var}\ (n+1) & \text{else} \end{cases} \\ \nu_k x.\mathsf{par}\ y = \begin{cases} \mathsf{var}\ k & \text{if } x = y \\ \mathsf{par}\ y & \text{else} \end{cases} \\ \nu_k x.\mathsf{app}\ u_1\ u_2 = \mathsf{app}\ (\nu_k x.u_1)\ (\nu_k x.u_2) \\ \nu_k x.\mathsf{abs}\ \sigma\ u_1 = \mathsf{abs}\ \sigma\ (\nu_{k+1} x.u_1) \end{cases}$$

$$u\lceil x \rceil_k \triangleq \begin{cases} (\mathsf{var}\ n)\lceil x \rceil_k = \begin{cases} \mathsf{par}\ x & \text{if } n = k \\ \mathsf{var}\ n & \text{if } n < k \\ \mathsf{var}\ (n-1) & \text{if } n > k \end{cases} \\ (\mathsf{par}\ y)\lceil x \rceil_k = \mathsf{par}\ y \\ (\mathsf{app}\ u_1\ u_2)\lceil x \rceil_k = \mathsf{app}\ (u_1\lceil x \rceil_k)\ (u_2\lceil x \rceil_k) \\ (\mathsf{abs}\ \sigma\ u_1)\lceil x \rceil_k = \mathsf{abs}\ \sigma\ (u_1\lceil x \rceil_{k+1}) \end{cases}$$

The definition of $[\![\Lambda_i]\!]$ as a dependent pair implies that, in the model, every term or context is composed of a structural part – a pre-term – together with a proof object asserting that the pre-term has the expected arity. This is reflected in the interpretation:

$$\begin{aligned}
[\![\mathsf{Par}\ x]\!] &= (\mathsf{par}\ x, \ldots) \\
[\![\mathsf{App}\ u_1\ u_2]\!] &= (\mathsf{app}\ \pi_1([\![u_1]\!])\ \pi_1([\![u_2]\!]), \ldots) \\
[\![\mathsf{Lam}\ x\ \sigma\ u_1]\!] &= (\mathsf{abs}\ \sigma\ (\nu_0 x.\pi_1([\![u_1]\!])), \ldots)
\end{aligned}$$

$$\begin{aligned}
[\![\nu x.u]\!] &\triangleq (\nu_0 x.\pi_1([\![u]\!]), \ldots) \\
[\![u\lceil x \rceil]\!] &\triangleq (\pi_1([\![u]\!])\lceil x \rceil_0, \ldots)
\end{aligned}$$

where π_1 is the left projection of a dependent pair (here used to extract a pre-term from the interpretation of a term) and the ellipses "..." must be filled with appropriate proof objects. When we limit ourselves to the structural part, most of the interpretations are straightforward, but that of Lam is worth looking into: the name-carrying lambda is transformed by interpreting its body u_1 first, then turning all the occurrences of the parameter x into a dangling index that is immediately bound by a nameless abstraction.

We have formalized the existence of proof objects such that the interpretation of terms and contexts satisfies the following lemma, stating its soundness.

Lemma 7.

1. $[\![\mathsf{Par}\ x]\!] : ctx\ 0$
2. *if* $[\![u]\!] : ctx\ 0$ *and* $[\![v]\!] : ctx\ 0$, *then* $[\![\mathsf{App}\ u\ v]\!] : ctx\ 0$
3. *if* $[\![u]\!] : ctx\ 0$, *then* $[\![\mathsf{Lam}\ x\ \sigma\ u]\!] : ctx\ 0$
4. *if* $[\![u]\!] : ctx\ i$, *then* $[\![\nu x.u]\!] : ctx\ (i+1)$
5. *if* $[\![u]\!] : ctx\ (i+1)$, *then* $[\![u\lceil x \rceil]\!] : ctx\ i$

We omit the trivial interpretation of the FV operation and state some of the remaing properties we proved to ensure the validity of the model.

Lemma 8.

1. $[\![x\ \langle y/x \rangle]\!] = [\![y]\!]$
2. *if* $x \neq y$, *then* $[\![x\ \langle z/y \rangle]\!] = [\![x]\!]$
3. $[\![(\mathsf{App}\ u\ v)\ \langle y/x \rangle]\!] = [\![\mathsf{App}\ (u\ \langle y/x \rangle)\ (v\ \langle y/x \rangle)]\!]$
4. *if* $z \neq x, y$, *then* $[\![(\mathsf{Lam}\ z\ \sigma\ u)\ \langle y/x \rangle]\!] = [\![\mathsf{Lam}\ z\ \sigma\ (u\ \langle y/x \rangle)]\!]$

Lemma 9. *(α-conversion)*
If $x, y \notin \mathrm{FV}(u)$, *then* $[\![\mathsf{Lam}\ x\ \sigma\ (u\lceil x \rceil)]\!] = [\![\mathsf{Lam}\ y\ \sigma\ (u\lceil y \rceil)]\!]$

Lemma 10.

1. $[\![(\nu x.u)\lceil x \rceil]\!] = [\![u]\!]$
2. *if* $x \notin \mathrm{FV}(u)$, *then* $[\![\nu x.(u\lceil x \rceil)]\!] = [\![u]\!]$

A final piece is missing to complete the model: an intepretation of the recursion principle $\mathcal{R}_{\Lambda_0}$, and the proof that its equational properties are valid. We provide such an interpretation as a recursive function *pretm_rec* on pre-terms, which is later lifted to proper terms.

The function *pretm_rec* receives similar arguments to the abstract $\mathcal{R}_{\Lambda_0}$, plus an additional f_{var} for dangling indices (which are missing from the ostensibly named presentation) that is not of particular interest here.

When dealing with a term of the form $\mathsf{abs}\ \sigma\ u$, we generate a new fresh name $x = \varphi(C, \mathrm{FV}(u))$ and open u with respect to that name; we then perform the recursive call on the opened $u\lceil x \rceil_0$. The full *pretm_rec* is defined by recursion on the *height* of the syntax tree of a pre-term, rather than structural recursion on the pre-term, because not all the recursive calls are on a pre-term which is structurally smaller than the one received in input (something that is beyond the capabilities of the termination heuristics found in Matita):

$$
\begin{array}{l}
\text{let rec } \mathit{pretm\_rec\_aux}\ (P : \mathit{pretm} \Rightarrow \mathbf{Type}) \\
\quad (C : \mathit{list}\ \mathbb{A})\ (f_{\mathsf{par}} : \forall x.P\ (\mathsf{par}\ x))\ (f_{\mathsf{var}} : \forall n.P\ (\mathsf{var}\ n)) \\
\quad (f_{\mathsf{app}} : \forall v1, v2.P\ v1 \Rightarrow P\ v2 \Rightarrow P\ (\mathsf{app}\ v1\ v2)) \\
\quad (f_{\mathsf{abs}} : \forall x, s, v.x \notin \mathrm{FV}\ \ v \Rightarrow x \notin C \Rightarrow P\ (v\lceil x \rceil) \Rightarrow P\ (\mathsf{abs}\ s\ (\nu x.(v\lceil x \rceil)))) \\
\quad (h : \mathbb{N})\ u\ \mathit{on}\ h : (\mathrm{height}(u) < h \Rightarrow P\ u) \triangleq \text{match } h \text{ with} \\
\quad [0 \Rightarrow \ldots \mathit{(* absurd: height is always > 0 *)} \\
\quad |S\ h_0 \Rightarrow \text{let } \mathit{rcall}\ \triangleq \mathit{pretm\_rec\_aux}\ P\ C\ f_{\mathsf{par}}\ f_{\mathsf{var}}\ f_{\mathsf{app}}\ f_{\mathsf{abs}}\ h_0 \text{ in} \\
\quad\quad \text{match } u \text{ with} \\
\quad\quad [\mathsf{par}\ x \Rightarrow \lambda\_.f_{\mathsf{par}}\ x \\
\quad\quad |\mathsf{var}\ n \Rightarrow \lambda\_.f_{\mathsf{var}}\ n \\
\quad\quad |\mathsf{app}\ v_1\ v_2 \Rightarrow \lambda p.f_{\mathsf{app}} \ldots (\mathit{rcall}\ v_1 \ldots)\ (\mathit{rcall}\ v_2 \ldots) \\
\quad\quad |\mathsf{abs}\ \sigma\ v \Rightarrow \text{let } x \triangleq \varphi(C, \mathrm{FV}(v)) \text{ in} \\
\quad\quad\quad f_{\mathsf{abs}} \ldots (\mathit{rcall}\ (v\lceil x \rceil_0) \ldots)]] \\
\\
\mathit{pretm\_rec}\ P\ C\ f_{\mathsf{par}}\ f_{\mathsf{var}}\ f_{\mathsf{app}}\ f_{\mathsf{abs}}\ u \triangleq \\
\quad \mathit{pretm\_rec\_aux}\ P\ C\ f_{\mathsf{par}}\ f_{\mathsf{var}}\ f_{\mathsf{app}}\ f_{\mathsf{abs}}\ (S\ \mathrm{height}(u))\ u \ldots
\end{array}
$$

The ellipses in *pretm_rec* and in the app and abs cases of *pretm_rec_aux* must be filled with proofs that the value provided for h is an upper bound to the height of the term on which we are performing recursion (in our formalization, those proofs were filled in interactively).

Since proper terms are a subset of pre-terms, expressing $\mathcal{R}_{\Lambda_0}$ in terms of *pretm_rec* is conceptually simple, although in practice the related proofs are technical, due to the handling of dependent types. The interested reader can check the details of the proof in the formalization, within the module `model.ma`.

4 Intermezzo: Formalizing Typing Rules

We now turn our attention to the formalization of more complex structures: typing judgments and their derivations by means of inductive rules. We chose the simply typed lambda-calculus as our setting, because even in its simplicity some of the issues of the representation of binding are already quite visible.

Its formalization in the most common representations of binding is well understood. Most locally nameless formalizations employ the following concrete introduction rule for lambda abstractions:

$$\frac{x \notin FV(u) \qquad \langle x, \sigma \rangle, \Gamma \vdash u\,\{\mathsf{var}\ 0 \mapsto \mathsf{par}\ x\} : \tau}{\Gamma \vdash \mathsf{abs}\ \sigma\ u : \sigma \to \tau}\ (\text{LN-T-Abs})$$

where $u\{v \mapsto v'\}$ is a generalized substitution operator, replacing a subterm v in u with v', preserving scopes. Following [23], we call rules in this style "backward", as they are most easily read from the bottom upwards: if the term which we intend to type can be deconstructed as $\mathsf{abs}\ \sigma\ u$, then we should first get a typing derivation for u in an extended typing environment. However, since unboxing an abstraction yields a term where the index $\mathsf{var}\ 0$ is possibly dangling, we are supposed to substitute a fresh name x for it, which must also be used in the extended context.

An alternative "forward" representation of the abstraction rule has a more familiar look:

$$\frac{\langle x, \sigma\rangle, \Gamma \vdash u : \tau}{\Gamma \vdash \mathsf{Lam}\ x\ \sigma\ u : \sigma \to \tau}\ (\text{LN-T-Lam})$$

In this case, the substitution is hidden inside the Lam operator: $\mathsf{Lam}\ x\ \sigma\ u$ is syntactic sugar for $\mathsf{abs}\ \sigma\ u\{\mathsf{par}\ x \mapsto \mathsf{var}\ 0\}$. Although this rule is more pleasant to read, in practice it is seldom used in formalizations because the associated induction principle is more difficult to use, due to the fact that Lam is not a real constructor: on the contrary, the algorithmic interpretation of the backward rule is immediate, as we argued some lines above.

As it turns out, even if we formalize a type system by means of backward rules, we get an induction principle which is weaker than what a formalizer expects. For example, suppose that we write a type checker for the simply typed calculus: we can verify its soundness with respect to the formalized type system (type-checking does not succeed for ill-typed terms) quite easily by induction; however verifying completeness (all well-typed terms typecheck successfully) turns out to be challenging for a naive formalizer.

As originally noted by McKinna and Pollack [15], the reason behind this difficulty lies in the fact that the LN-T-Abs rule is quite liberal: x can be any sufficiently fresh parameter. Given the typing judgment associated to an abstraction, we get a different derivation for every choice of a suitable x. All the derivations are isomorphic, but contain, so to say, a "hardcoded" parameter name: in other words, when we view typing derivations as data structures, they are not canonical.

The problem with typing derivations being not canonical is that, in a proof by induction, the hardcoded fresh parameter x makes its return as part of the induction hypothesis associated with the abstraction case:

$$\begin{array}{l}
\forall P. \\
\ldots \\
\left(\begin{array}{l}
\forall \Gamma, x, \sigma, u, \tau. \\
\quad x \notin \mathrm{FV}(u) \Rightarrow \\
\quad \langle x, \sigma\rangle, \Gamma \vdash u\{\mathsf{var}\ 0 \mapsto \mathsf{par}\ x\} : \tau \Rightarrow \\
\quad P(\langle x, \sigma\rangle, \Gamma, u\{\mathsf{var}\ 0 \mapsto \mathsf{par}\ x\}, \tau) \Rightarrow \\
\quad P(\Gamma, \mathsf{abs}\ \sigma\ u, \sigma \to \tau)
\end{array}\right) \Rightarrow \\
\ldots \\
\forall \Gamma, u, \sigma. \Gamma \vdash u : \sigma \Rightarrow P(\Gamma, u, \sigma)
\end{array}$$

However on many occasions we will need our induction hypothesis to refer to an *arbitrary* $y \notin \mathrm{dom}(\Gamma)$ (or even *all* such ys).

We can force typing derivations to be canonical (independent of arbitrary choices of parameter names) by means of a universally quantified premise:

$$\frac{\begin{pmatrix}\forall x.x \notin \mathrm{dom}(\Gamma), \mathrm{FV}(u) \Rightarrow \\ \quad \langle x, \sigma\rangle, \Gamma \vdash u\,\{\mathsf{var}\ 0 \mapsto \mathsf{par}\ x\} : \tau\end{pmatrix}}{\Gamma \vdash \mathsf{abs}\ \sigma\ u : \sigma \to \tau}\ (\text{LN-T-Abs'})$$

This yields a *strong* induction principle, where the induction hypothesis associated to the abstraction case is similarly quantified over all suitable xs. However, the rule LN-T-Abs' itself is actually weaker: to derive a typing judgment for abstractions, one now needs to prove an infinite number of judgments, one for every choice of x! This is not how typecheckers work and is thus usually not considered a good formalization of a typing rule.

Still, it must be noted that all the rules presented in this section are equivalent. In particular, it is possible to prove the "strong" induction principle for a formalization using LN-T-Abs by showing that the typing judgment is *equivariant*, i.e. stable under arbitrary finite permutations of names π:

$$\Gamma \vdash u : \sigma \iff \forall \pi.\pi \cdot \Gamma \vdash \pi \cdot u : \sigma$$

5 Ostensibly Named Representation of Typing

We employ the ostensibly named style presented in Section 2 to express the type system of the simply typed lambda calculus.

$$\frac{\langle x, \sigma\rangle \in \Gamma \qquad \mathrm{dom}(\Gamma) \text{ is duplicate-free}}{\Gamma \vdash_O \mathsf{Par}\ x : \sigma}\ (\text{ON-T-Par})$$

$$\frac{\langle x, \sigma\rangle, \Gamma \vdash_O u : \tau}{\Gamma \vdash_O \mathsf{Lam}\ x\ \sigma\ u : \sigma \to \tau}\ (\text{ON-T-Lam})$$

$$\frac{\Gamma \vdash_O u : \sigma \to \tau \qquad \Gamma \vdash_O v : \sigma}{\Gamma \vdash_O \mathsf{App}\ u\ v : \tau}\ (\text{ON-T-App})$$

Fig. 1. Typing rules for $\boldsymbol{\lambda}_{\to}$, ostensibly named style

The typing rules, shown in Figure 1, look quite unremarkable. The rule ON-T-Lam, in particular, looks the same as the rule LN-T-Lam of the previous section, although in this case Lam is opaque and, more importantly, the rules themselves must not be intended as the constructors of the concrete inductive

type of typing derivations, but as operations provided by the abstract data type of typing derivations. We postpone the discussion about the internal representation of typing to the next section.

$$
\begin{array}{l}
\forall P. \\
\quad (\forall \Gamma, x, \sigma. \langle x, \sigma\rangle \in \Gamma \Rightarrow P(\Gamma, \mathsf{Par}\ x, \sigma)) \Rightarrow \\
\quad \left(\begin{array}{l}
\quad \forall \Gamma, x, \sigma, u, \tau. x \notin \mathrm{dom}(\Gamma), \mathrm{FV}(u) \Rightarrow \\
\qquad (\forall y. y \notin \mathrm{dom}(\Gamma), \mathrm{FV}(u) \Rightarrow \langle y, \sigma\rangle, \Gamma \vdash_O u\lceil y\rceil : \tau) \Rightarrow \\
\qquad (\forall y. y \notin \mathrm{dom}(\Gamma), \mathrm{FV}(u) \Rightarrow P(\langle y, \sigma\rangle, \Gamma, u\lceil y\rceil, \tau)) \Rightarrow \\
\qquad P(\Gamma, \mathsf{Lam}\ x\ \sigma\ (u\lceil x\rceil), \sigma \to \tau)
\end{array}\right) \Rightarrow \\
\quad \left(\begin{array}{l}
\quad \forall \Gamma, u, \sigma, \tau. \\
\qquad \Gamma \vdash_O u : \sigma \to \tau \Rightarrow \Gamma \vdash_O v : \sigma \Rightarrow \\
\qquad P(\Gamma, u, \sigma \to \tau) \Rightarrow P(\Gamma, v, \sigma) \Rightarrow \\
\qquad P(\Gamma, \mathsf{App}\ u\ v, \tau)
\end{array}\right) \Rightarrow \\
\quad \forall \Gamma, u, \sigma. \Gamma \vdash_O u : \sigma \Rightarrow P(\Gamma, u, \sigma)
\end{array}
$$

Fig. 2. Rule induction for the $\boldsymbol{\lambda}_{\to}$ typing derivations, ostensibly named style

The ostensibly named induction principle we associate to these rules (Figure 2) is more interesting. The induction hypothesis of the lambda case (highlighted in the figure) is quantified over all suitable parameter names, as in a strong principle; however, we use the variable opening operation, both in the induction hypothesis and in the conclusion, to avoid exposing the internal structure of the terms. To prevent variable capture, the new names are chosen to be fresh with respect to the context u being opened.

Ostensibly Named Inversion. We can use the ostensibly named induction principle to derive an *inversion* principle in the style of McBride ([14]). Together with Lemma 2, inversion principles provide an effective tool to perform case analysis on the last rule used in a derivation tree. The inversion principle we obtain is *strong* (as in [6]) in the sense that, for instance, given a derivation of $\Gamma \vdash \mathsf{Lam}\ x\ \sigma\ (u\lceil x\rceil) : \sigma \to \tau$ with $x \notin \mathrm{FV}(u)$, we can deduce $\langle y, \sigma\rangle, \Gamma \vdash u\lceil y\rceil : \tau$ for all $y \notin \mathrm{FV}(u), \mathrm{dom}(\Gamma)$.

5.1 Internal Representation of Typing Rules

As we argued in Section 4, the weak or strong induction principle dilemma, in the context of typing, stems from the fact that the natural typing rules mentioning a specific variable in the binder case, yield a plurality of derivations for the same typing judgment; but to have a single derivation and thus a strong induction principle, one has to employ an infinitary typing rule.

In essence, names are the origin of the dilemma: so it is just natural to look at a de Bruijn formalization of the typing rules, shown in Figure 3. In the nameless encoding, typing environments are just lists of types: we denote them as $\gamma, \gamma', \ldots$.

$$\frac{\gamma(n) = \sigma}{\gamma \vdash_D \mathsf{var}\ n : \sigma}\ (\text{DB-T-VAR})$$

$$\frac{\sigma, \gamma \vdash_D u : \tau}{\gamma \vdash_D \mathsf{abs}\ \sigma\ u : \sigma \to \tau}\ (\text{DB-T-ABS})$$

$$\frac{\gamma \vdash_D u : \sigma \to \tau \qquad \gamma \vdash_D v : \sigma}{\gamma \vdash_D \mathsf{app}\ u\ v : \tau}\ (\text{DB-T-APP})$$

Fig. 3. Typing rules for $\boldsymbol{\lambda}_{\to}$, pure de Bruijn style

Since in this presentation no named parameter appears, context references are by position (rule DB-T-VAR, where $\gamma(n)$ returns the $n+1$-th type in γ). In the abstraction rule, unboxing an abstraction yields, in the premise, a new dangling index, whose type is referenced in an extended context.

The nice thing about going nameless is the following: the rule DB-T-ABS is finitary (in fact, unary), but at the same time it is also canonical! For every well-typed abstraction, there is exactly one derivation, because we do not have the freedom of choosing *any* fresh name: in fact, we choose *none*. This desirable situation comes from the fact that, in a nameless setting, not only abstractions, but also the typing environment γ of the judgments and even the rule DB-T-ABS are treated as binders.

These properties make the de Bruijn style rules, together with the associated induction principle (Figure 4), an ideal model for the abstract rules of the previous section.

$$\begin{array}{l}
\forall P. \\
\quad (\forall \gamma, n, \sigma. \gamma(n) = \sigma \Rightarrow P(\gamma, \mathsf{var}\ n, \sigma)) \Rightarrow \\
\quad \left(\begin{array}{l} \forall \gamma, \sigma, u, \tau. \\ \quad \sigma, \gamma \vdash_D u : \tau \Rightarrow \\ \quad P(\sigma, \gamma, u, \tau) \Rightarrow \\ \quad P(\gamma, \mathsf{abs}\ \sigma\ u, \sigma \to \tau) \end{array}\right) \Rightarrow \\
\quad \left(\begin{array}{l} \forall \gamma, u, \sigma, \tau. \\ \quad \gamma \vdash_D u : \sigma \to \tau \Rightarrow \gamma \vdash_D v : \sigma \Rightarrow \\ \quad P(\gamma, u, \sigma \to \tau) \Rightarrow P(\gamma, v, \sigma) \Rightarrow \\ \quad P(\gamma, \mathsf{App}\ u\ v, \tau) \end{array}\right) \Rightarrow \\
\quad \forall \gamma, u, \sigma. \gamma \vdash_D u : \sigma \Rightarrow P(\gamma, u, \sigma)
\end{array}$$

Fig. 4. Rule induction for the $\boldsymbol{\lambda}_{\to}$ typing derivations, de Bruijn style

To model the ostensibly named presentation of $\boldsymbol{\lambda}_{\to}$, we first need to interpret $\vdash_O$ in terms of $\vdash_D$. For this purpose, we give an interpretation of the ostensibly

named representations of types, typing environments, and terms into the corresponding concepts of the de Bruijn representation. As usual, the interpretation of types is the identity. For what concerns typing environments, all we need to do is to throw away the names, keeping the types in the same order: this is best done by projecting the second component of each pair in the list. Finally the interpretation of terms is given by taking the interpretation we used in Section 3 and subsequently closing the obtained locally-nameless term with respect to the names in its typing context: assuming all the names referenced in the term have an entry in the environment, the resulting interpretation is nameless. In symbols:

$$\begin{aligned} [\![\sigma]\!] &\triangleq \sigma \\ [\![\Gamma]\!] &\triangleq \mathrm{cod}(\Gamma) \\ [\![u]\!]_{\overrightarrow{x}} &\triangleq \nu_0 \overrightarrow{x}.[\![u]\!] \end{aligned}$$

$$[\![\Gamma \vdash_O u : \sigma]\!] \triangleq df(\mathrm{dom}(\Gamma)) \wedge [\![\Gamma]\!] \vdash_D [\![u]\!]_{\mathrm{dom}(\Gamma)} : [\![\sigma]\!]$$

where:

$$\begin{aligned} \mathrm{dom}([x_1 : \sigma_1; \ldots; x_n : \sigma_n]) &\triangleq [x_1; \ldots; x_n] \\ \mathrm{cod}([x_1 : \sigma_1; \ldots; x_n : \sigma_n]) &\triangleq [\sigma_1; \ldots; \sigma_n] \\ \nu_k[x_1; \ldots; x_n].u &\triangleq \nu_k x_1 \ldots \nu_k x_n.u \end{aligned}$$

The model of an ostensibly named judgment contains, in addition to its nameless counterpart, a proof that the domain of Γ is duplicate-free (a property which we expect to be able to prove, and which is not implied by the nameless judgment). We have used the predicate *df* to assert that a certain list of names is duplicate-free. The vector notation $\overrightarrow{x}$ is employed as a compact way of referring to lists, in this case to a list of names. Our second task is to model the rules of Figure 1 as instances of their nameless counterparts. This is expressed by the following lemma:

Lemma 11.

1. *If* $\langle x, \sigma \rangle \in \Gamma$ *and* $\mathrm{dom}(\Gamma)$ *is duplicate-free, then* $[\![\Gamma \vdash_O \mathsf{Par}\ x : \sigma]\!]$.
2. *If* $[\![\langle x, \sigma \rangle, \Gamma \vdash_O u : \tau]\!]$, *then* $[\![\Gamma \vdash_O \mathsf{Lam}\ x\ \sigma\ u : \sigma \to \tau]\!]$.
3. *If* $[\![\Gamma \vdash_O u : \sigma \to \tau]\!]$ *and* $[\![\Gamma \vdash_O v : \sigma]\!]$, *then* $[\![\Gamma \vdash_O \mathsf{App}\ u\ v : \tau]\!]$.

Finally, we provide an interpretation of the ostensibly named induction principle as follows:

Theorem 12. *Let* P *be a predicate over named typing environments, terms, and types. Assume the following properties:*

1. *for all* Γ,x,σ, $\langle x, \sigma \rangle \in \Gamma$ *implies* $P(\Gamma, \mathsf{Par}\ x, \sigma)$;
2. *for all* Γ,x,σ,u,τ *such that*
   - $x \notin \mathrm{dom}(\Gamma), \mathrm{FV}(u)$
   - $\forall y.y \notin \mathrm{dom}(\Gamma), \mathrm{FV}(u) \Rightarrow [\![\langle y, \sigma \rangle, \Gamma \vdash_O u\lceil y \rceil : \tau]\!]$
   - $\forall y.y \notin \mathrm{dom}(\Gamma), \mathrm{FV}(u) \Rightarrow P(\langle y, \sigma \rangle, \Gamma, u\lceil y \rceil, \tau)$

   then $P(\Gamma, \mathsf{Lam}\ x\ \sigma\ (u\lceil x \rceil), \sigma \to \tau)$ *holds;*

3. for all Γ,u,v,σ,τ such that
- $[\![\Gamma \vdash_O u : \sigma \to \tau]\!]$
- $[\![\Gamma \vdash_O v : \sigma]\!]$
- $P(\Gamma, u, \sigma \to \tau)$
- $P(\Gamma, v, \sigma)$

then $P(\Gamma, \mathsf{App}\ u\ v, \tau)$.

Then for all Γ,u,σ such that $[\![\Gamma \vdash_O u : \sigma]\!]$, $P(\Gamma, u, \sigma)$ holds.

Proof (sketch). Assume $[\![\Gamma \vdash_O u : \sigma]\!]$. By definition, we know that the domain of Γ is duplicate-free and that $[\![\Gamma]\!] \vdash_D [\![u]\!]_{\mathrm{dom}(\Gamma)} : \sigma$.

Let $\hat{P}$ be the augmented predicate:

$$\hat{P}(\gamma, u, \sigma) \triangleq \forall \overrightarrow{x_{|\gamma|}}.df(\overrightarrow{x_{|\gamma|}}) \Rightarrow P(\gamma\lceil\overrightarrow{x_{|\gamma|}}\rceil, u\lceil\overrightarrow{x_{|\gamma|0}}\rceil, \sigma)$$

where we have extended the definition of $-\lceil-\rceil$ as follows:

$$[\sigma_1; \ldots; \sigma_n]\lceil[x_1; \ldots; x_n]\rceil \triangleq [\langle x_1, \sigma_1\rangle; \ldots; \langle x_n, \sigma_n\rangle]$$
$$u\lceil\overrightarrow{x}\rceil_k \triangleq \begin{cases} u\lceil[]\rceil_k = u \\ u\lceil y, \overrightarrow{z}\rceil_k = u\lceil y\rceil_k\lceil\overrightarrow{z}\rceil_k \end{cases}$$

The notation $|\gamma|$ indicates the length of the list γ. The extended vector notation in the form $\overrightarrow{x_n}$ is used to express lists of length n. We now proceed by induction on $[\![\Gamma]\!] \vdash_D [\![u]\!]_{\mathrm{dom}(\Gamma)} : \sigma$ to prove $\hat{P}([\![\Gamma]\!], [\![u]\!]_{\mathrm{dom}(\Gamma)}, \sigma)$. By instantiating the augmented predicate over the list $\mathrm{dom}(\Gamma)$, we finally obtain $P(\Gamma, u, \sigma)$ (using lemma 10).

The most difficult part of the induction is the abstraction case: given $\gamma_0, \sigma_0, \tau_0, u_0$ and a duplicate free list of names $\overrightarrow{x_{|\gamma_0|}}$, we need to prove

$$P(\gamma_0\lceil\overrightarrow{x_{|\gamma_0|}}\rceil, (\mathsf{abs}\ \sigma_0\ u_0)\lceil\overrightarrow{x_{|\gamma_0|}}\rceil, \sigma_0 \to \tau_0)$$

under the hypotheses

$$\sigma_0, \gamma_0 \vdash_D u_0 : \tau_0$$
$$\hat{P}(\sigma_0, \gamma_0, u_0, \tau_0)$$

where the latter is the induction hypothesis. Then we take a fresh name y and rewrite in the thesis

$$\begin{aligned} &(\mathsf{abs}\ \sigma_0\ u_0)\lceil\overrightarrow{x_{|\gamma_0|}}\rceil \\ =\ &\mathsf{abs}\ \sigma_0\ (u_0\lceil\overrightarrow{x_{|\gamma_0|}}\rceil_1) \\ =\ &\mathsf{Lam}\ y\ \sigma_0\ (u_0\lceil\overrightarrow{x_{|\gamma_0|}}\rceil_1\lceil y\rceil) \end{aligned}$$

This allows us to apply the second lemma hypothesis (to fulfill the guards of the hypothesis, we exploit the equality $u_0\lceil\overrightarrow{x_{|\gamma_0|}}\rceil_1\lceil y\rceil = u_0\lceil y, \overrightarrow{x_{|\gamma_0|}}\rceil$).

6 Beta Reduction

The ostensibly named technique we used in the previous section to define typing judgments extends to other types of judgments. The trick is to make explicit the environment where all the free parameters appearing in the judgment are defined. Other than that, we axiomatize reduction rules that are close to informal syntax (Fig. 5). These introduction rules are completed by a strong induction principle, providing a universally quantified induction hypothesis for the case where reduction happens under a lambda (Fig. 6).

6.1 De Bruijn Model of Beta Reduction

A de Bruijn-style model of beta reduction is given in Figure 7. The main difference with the ostensibly named rules is that the list of free parameters is replaced by an integer stating the number of dangling indices possibly appearing in the judgment. Thus we define the interpretation of an ostensibly named reduction as:

$$[\![\overrightarrow{x_k} \vdash_O u \triangleright v]\!] \triangleq k \vdash_D [\![u]\!]_{\overrightarrow{x_k}} \triangleright [\![v]\!]_{\overrightarrow{x_k}}$$

All the preterms involved in the de Bruijn judgment must be contexts of a suitable arity containing no parameters. When this property is not implied by

$$\frac{\overrightarrow{x} \text{ is duplicate-free} \qquad \mathrm{FV}(\mathsf{App}\ (\mathsf{Lam}\ y\ \sigma\ u)\ v) \subseteq \overrightarrow{x}}{\overrightarrow{x} \vdash_O \mathsf{App}\ (\mathsf{Lam}\ y\ \sigma\ u)\ v \triangleright u\,[v/_y]}\ (\text{ON-B-Red})$$

$$\frac{\overrightarrow{x} \vdash_O u \triangleright u' \qquad \mathrm{FV}(v) \subseteq \overrightarrow{x}}{\overrightarrow{x} \vdash_O \mathsf{App}\ u\ v \triangleright \mathsf{App}\ u'\ v}\ (\text{ON-B-App1})$$

$$\frac{\overrightarrow{x} \vdash_O v \triangleright v' \qquad \mathrm{FV}(u) \subseteq \overrightarrow{x}}{\overrightarrow{x} \vdash_O \mathsf{App}\ u\ v \triangleright \mathsf{App}\ u\ v'}\ (\text{ON-B-App2})$$

$$\frac{y, \overrightarrow{x} \vdash_O u \triangleright u'}{\overrightarrow{x} \vdash_O \mathsf{Lam}\ y\ \sigma\ u \triangleright \mathsf{Lam}\ y\ \sigma\ u'}\ (\text{ON-B-Xi})$$

Fig. 5. Beta reduction: ostensibly named encoding

$$\begin{array}{l}
\forall P. \\
\left(\begin{array}{l}
\forall \overrightarrow{x}, y, \sigma, u, v. df(\overrightarrow{x}) \Rightarrow \\
\quad \mathrm{FV}(\mathsf{App}\ (\mathsf{Lam}\ y\ \sigma\ u)\ v) \subseteq \overrightarrow{x} \Rightarrow \\
\quad P(\overrightarrow{x}, \mathsf{App}\ (\mathsf{Lam}\ y\ \sigma\ u)\ v, u\,[v/_y])
\end{array}\right) \Rightarrow \\
\left(\begin{array}{l}
\forall \overrightarrow{x}, u, u', v. \mathrm{FV}(v) \subseteq \overrightarrow{x} \Rightarrow \\
\quad \overrightarrow{x} \vdash_O u \triangleright u' \Rightarrow P(\overrightarrow{x}, u, u') \Rightarrow \\
\quad P(\overrightarrow{x}, \mathsf{App}\ u\ v, \mathsf{App}\ u'\ v)
\end{array}\right) \Rightarrow \\
\left(\begin{array}{l}
\forall \overrightarrow{x}, u, v, v'. \mathrm{FV}(u) \subseteq \overrightarrow{x} \Rightarrow \\
\quad \overrightarrow{x} \vdash_O v \triangleright v' \Rightarrow P(\overrightarrow{x}, v, v') \Rightarrow \\
\quad P(\overrightarrow{x}, \mathsf{App}\ u\ v, \mathsf{App}\ u\ v')
\end{array}\right) \Rightarrow \\
\left(\begin{array}{l}
\forall \overrightarrow{x}, y, \sigma, u, u'. y \notin \overrightarrow{x}, \mathrm{FV}(u), \mathrm{FV}(u') \Rightarrow \\
\quad (\forall z. z \notin \overrightarrow{x}, \mathrm{FV}(u), \mathrm{FV}(u') \Rightarrow z, \overrightarrow{x} \vdash_O u\lceil z \rceil \triangleright u'\lceil z \rceil) \Rightarrow \\
\quad (\forall z. z \notin \overrightarrow{x}, \mathrm{FV}(u), \mathrm{FV}(u') \Rightarrow P(z, \overrightarrow{x}, u\lceil z \rceil, u'\lceil z \rceil)) \Rightarrow \\
\quad P(\overrightarrow{x}, \mathsf{Lam}\ y\ \sigma\ (u\lceil y \rceil), \mathsf{Lam}\ y\ \sigma\ (u'\lceil y \rceil))
\end{array}\right) \Rightarrow \\
\forall \overrightarrow{x}, u, u'. \overrightarrow{x} \vdash_O u \triangleright u' \Rightarrow P(\overrightarrow{x}, u, u')
\end{array}$$

Fig. 6. Induction principle for beta reduction: ostensibly named encoding

a recursive premise of a rule, we have to specify it as an extra premise: for this purpose, we use the notation

$$k \vdash_D u \ \mathbf{ok} \triangleq \text{check\_tm}\ u\ k = \mathit{true} \wedge \mathrm{FV}(u) = \emptyset$$

$$\frac{y, \overrightarrow{x_k} \text{ is duplicate-free} \qquad k \vdash_D \mathsf{app}\ (\mathsf{abs}\ \sigma\ u)\ v\ \mathbf{ok}}{k \vdash_D \mathsf{app}\ (\mathsf{abs}\ \sigma\ u)\ v \triangleright [\![u\lceil y, \overrightarrow{x_k}\rceil\ [^{v\lceil \overrightarrow{x_k}\rceil}/_y]]\!]_{\overrightarrow{x_k}}} \text{(DB-B-Red)}$$

$$\frac{k \vdash_D u \triangleright u' \qquad k \vdash_D v\ \mathbf{ok}}{k \vdash_D \mathsf{app}\ u\ v \triangleright \mathsf{app}\ u'\ v} \text{(DB-B-App1)}$$

$$\frac{k \vdash_D v \triangleright v' \qquad k \vdash_D u\ \mathbf{ok}}{k \vdash_D \mathsf{app}\ u\ v \triangleright \mathsf{app}\ u\ v'} \text{(DB-B-App2)}$$

$$\frac{k+1 \vdash_D u \triangleright u'}{k \vdash_D \mathsf{abs}\ \sigma\ u \triangleright \mathsf{abs}\ \sigma\ u'} \text{(DB-B-Xi)}$$

Fig. 7. Beta reduction: de Bruijn encoding

While most other adaptations are trivial, rule DB-B-Red is slightly upsetting: de Bruijn terms u and v are opened in an arbitrary environment to become ostensibly named terms; then we use ostensibly named substitution and finally convert the result back to a de Bruijn term. This round-trip is entirely unnecessary if we define a substitution operation on de Bruijn terms; however for our purpose – justifying the ostensibly named rules and induction principle – this is not required: thus we decided not to bother dealing with two notions of substitution and their equivalence.

The following properties show that the de Bruijn rules are a model of the ostensibly named rules. Their proofs are similar to those relative to the typing judgment.

Lemma 13.

1. *If* $\overrightarrow{x}$ *is duplicate-free and* $\mathrm{FV}(\mathsf{App}\ (\mathsf{Lam}\ y\ \sigma\ u)\ v) \subseteq \overrightarrow{x}$ *then* $[\![\overrightarrow{x} \vdash_O \mathsf{App}\ (\mathsf{Lam}\ y\ \sigma\ u)\ v \triangleright u\,[^v/_y]]\!]$.
2. *If* $[\![\overrightarrow{x} \vdash_O u \triangleright u']\!]$ *and* $\mathrm{FV}(v) \subseteq \overrightarrow{x}$ *then* $[\![\overrightarrow{x} \vdash_O \mathsf{App}\ u\ v \triangleright \mathsf{App}\ u'\ v]\!]$.
3. *If* $[\![\overrightarrow{x} \vdash_O v \triangleright v']\!]$ *and* $\mathrm{FV}(u) \subseteq \overrightarrow{x}$ *then* $[\![\overrightarrow{x} \vdash_O \mathsf{App}\ u\ v \triangleright \mathsf{App}\ u\ v']\!]$.
4. *If* $[\![y, \overrightarrow{x} \vdash_O u \triangleright u']\!]$ *then* $[\![\overrightarrow{x} \vdash_O \mathsf{Lam}\ y\ \sigma\ u \triangleright \mathsf{Lam}\ y\ \sigma\ u')]\!]$.

Theorem 14. *Let* P *be a predicate over named lists of variable names and pairs of terms. Assume the following properties:*

1. *for all* $\overrightarrow{x}$,y,σ,u,v *such that* $\overrightarrow{x}$ *is duplicate-free,* $\mathrm{FV}(\mathsf{App}\ (\mathsf{Lam}\ y\ \sigma\ u)\ v) \subseteq \overrightarrow{x}$ *implies* $P(\overrightarrow{x}, \mathsf{App}\ (\mathsf{Lam}\ y\ \sigma\ u)\ v, u\,[v/y])$;
2. *for all* $\overrightarrow{x}$,u,u',v *such that*
   - $\mathrm{FV}(v) \subseteq \overrightarrow{x}$
   - $[\![\overrightarrow{x} \vdash u \triangleright u']\!]$
   - $P(\overrightarrow{x}, u, u')$

   then $P(\overrightarrow{x}, \mathsf{App}\ u\ v, \mathsf{App}\ u'\ v)$ *holds;*
3. *for all* $\overrightarrow{x}$,u,v,v' *such that*
   - $\mathrm{FV}(u) \subseteq \overrightarrow{x}$
   - $[\![\overrightarrow{x} \vdash v \triangleright v']\!]$
   - $P(\overrightarrow{x}, v, v')$

   then $P(\overrightarrow{x}, \mathsf{App}\ u\ v, \mathsf{App}\ u\ v')$ *holds;*
4. *for all* $\overrightarrow{x}$,y,σ,u,u' *such that*
   - $y \notin \overrightarrow{x}, \mathrm{FV}(u), \mathrm{FV}(u')$;
   - $\forall z.z \notin \overrightarrow{x}, \mathrm{FV}(u), \mathrm{FV}(u') \Rightarrow [\![y, \overrightarrow{x} \vdash_O u\lceil z\rceil \triangleright u'\lceil z\rceil]\!]$
   - $\forall z.z \notin \overrightarrow{x}, \mathrm{FV}(u), \mathrm{FV}(u') \Rightarrow P(y, \overrightarrow{x}, u\lceil z\rceil, u'\lceil z\rceil)$

   then $P(\overrightarrow{x}, \mathsf{Lam}\ y\ \sigma\ (u\lceil y\rceil), \mathsf{Lam}\ y\ \sigma, u'\lceil y\rceil)$ *holds;*

Then for all $\overrightarrow{x}$,u,u' *such that* $[\![\overrightarrow{x} \vdash_O u \triangleright u']\!]$, $P(\overrightarrow{x}, u, u')$ *holds.*

6.2 Some Formalized Results

The machinery we have presented in the previous sections is all we need to prove metatheoretical properties of $\boldsymbol{\lambda}_{\rightarrow}$ such as weakening of typing judgments and subject reduction.

Theorem 15 (weakening of typing). *If* $\Gamma \vdash_O u : \sigma$ *and* $\Gamma \subseteq \Delta$, *then* $\Delta \vdash_O u : \sigma$.

Proof. Routine induction on the derivation of $\Gamma \vdash_O u : \sigma$, closely resembling the corresponding proof in a locally nameless setting. This is remarkable when considering that the underlying implementation of our typing judgments uses a pure nameless approach: normally, a proof of weakening in a nameless setting requires relatively complex arguments about lifting and permutations of indices and typing environments. Such unnecessary technicalities are completely hidden in our proof because the ostensibly named approach allows for a more adequate degree of abstraction.

Lemma 16 (substitutivity of typing). *If* $\Delta, \langle x, \sigma\rangle, \Gamma \vdash_O u : \tau$ *and* $\Gamma \vdash_O v : \sigma$, *then* $\Delta, \Gamma \vdash_O u\,[v/x] : \tau$.

Theorem 17 (preservation of typing). *If* $\Gamma \vdash_O u : \sigma$ *and* $\mathrm{dom}(\Gamma) \vdash_O u \triangleright u'$, *then* $\Gamma \vdash_O u' : \sigma$.

Proof. We proceed by induction on the derivation of $\Gamma \vdash u : \sigma$, followed, for each case, by an inversion on the reduction judgment. These are the interesting cases:

- ON-T-App and ON-B-Red: we have $u = \mathsf{App}\ (\mathsf{Lam}\ x\ \sigma_0\ u_0)\ v_0$, $u' = u_0\,[^{v_0}/_x]$ and $\sigma = \sigma_0 \to \tau_0$, and we also know that $\Gamma \vdash_O \mathsf{Lam}\ x\ \sigma_0\ u_0 : \sigma_0 \to \tau_0$ and $\Gamma \vdash_O v_0 : \sigma_0$; we must prove $\Gamma \vdash_O u_0\,[^{v_0}/_x]$. By inversion we obtain z,u_1 such that $z \notin \mathrm{FV}(u_1), \mathrm{dom}(\Gamma)$, $\mathsf{Lam}\ \sigma_0\ x\ u_0 = \mathsf{Lam}\ z\ \sigma_0\ (u_1\lceil z\rceil)$ and $\langle z : \sigma_0\rangle, \Gamma \vdash u_1\lceil z\rceil : \tau_0$ (this implies $u_1\lceil z\rceil = u_0\,\langle {}^z/_x\rangle$); thus, by Lemma 16 we get $\Gamma \vdash_O u_1\lceil z\rceil\,[^v/_z]$; it is then easy to prove $u_1\lceil z\rceil\,[^v/_z] = u_0\,\langle {}^z/_x\rangle\,[^v/_z] = u_0\,[^v/_x]$, as needed.
- ON-T-Lam and ON-B-Xi: we have $u = \mathsf{Lam}\ x\ \sigma_0\ (u_0\lceil x\rceil)$, $u' = \mathsf{Lam}\ y\ \sigma_0\ (u_0'\lceil y\rceil)$ and $\sigma = \sigma_0 \to \tau_0$, where $x \notin \mathrm{dom}(\Gamma), \mathrm{FV}(u_0)$ and $y \notin \mathrm{dom}(\Gamma), \mathrm{FV}(u_0), \mathrm{FV}(u_0')$, and we also know that $\mathrm{dom}(\langle y, \sigma_0\rangle, \Gamma) \vdash_O u_0\lceil y\rceil \triangleright u_0'\lceil y\rceil$. By induction hypothesis, for all $z \notin \mathrm{dom}(\Gamma), \mathrm{FV}(u_0)$, for all u'' such that $\mathrm{dom}(\langle z, \sigma\rangle, \Gamma) \vdash u_0\lceil z\rceil \triangleright u''$, we have $\langle z, \sigma_0\rangle, \Gamma \vdash_O u'' : \tau_0$; thus, by taking $z = y$ and $u'' = u_0'\lceil y\rceil$, we have $\langle y, \sigma_0\rangle, \Gamma \vdash_O u_0'\lceil y\rceil : \tau_0$. From this we immediately derive $\Gamma \vdash_O \mathsf{Lam}\ y\ \sigma_0\ (u_0'\lceil y\rceil) : \sigma_0 \to \tau_0$, as needed.

Theorem 18 (progress). *If $\vdash_O u : \sigma$ and u is not a value (i.e. it is not in the form $\mathsf{Lam}\ x\ \tau\ v$), then there exists u' such that $\vdash_O u \triangleright u'$.*

Proof. By induction on u. By hypothesis, u is not a Lam abstraction and, since it is well typed in the empty environment, it is not a Par either. Therefore, we have $u = \mathsf{App}\ u_1\ u_2$. By inversion of typing, we also know that $\vdash_O u_1 : \tau \to \sigma$ and $\vdash_O u_2 : \tau$, for some type τ. Then:

- if u_1 is in the form $\mathsf{Lam}\ y\ \tau\ u_1'$, we have a redex: we can take $u' = u_1'\,[^{u_2}/_y]$ and obtain the thesis by rule ON-B-Red;
- otherwise u_1 is not a value and by induction hypothesis there exists a u_1' such that $\vdash_O u_1 \triangleright u_1'$: then we can take $u' = \mathsf{App}\ u_1'\ u_2$ and obtain the thesis by rule ON-B-App1 (the side condition about the free variables of u_2 is easily closed knowing that u_2 is well typed in the empty environment).

7 Conclusions

In this paper we have presented an ostensibly named abstract data type for the formalization of languages with binding, which enables the user of an interactive theorem prover to only deal with familiar concepts like named binders and terms with holes. Our work can be likened to other axiomatic or abstract approaches ([11,19,21] just to list a few). While other authors have focused especially on the representation of terms and recursively defined functions, our technique extends to inductively defined judgments. In the representation of judgments, an important role is played by our ability to express contexts (terms with holes).

To show the soundness of our axiomatization, we provided and fully formalized a constructive model employing de Bruijn indices. The term language of the model is locally nameless, with non-locally closed terms used to represent contexts. Judgments, instead, are represented in a pure de Bruijn fashion. Since the model is formalized, we retain the possibility of extracting code from all the definitions and proofs based on the ostensibly named ADT.

Even though our internal representation of binding structures employs nameless dummies, other models are possible as long as they are canonical, the most obvious alternatives being the canonical locally named representation [17] and nested datatypes [7]. However, users do not need to worry about this, since they only deal with an abstract data type that does not expose such inner details.

In the long run, every representation of binding should be expected to scale up to dependently-typed object languages with generalized binders (i.e. binders declaring multiple variables simultaneously) and possibly other complex operations. The system $\boldsymbol{\lambda}_{\rightarrow}$ that we formalized does not include dependent types; however multiple binders *are* part of our formalization, if only in the constrained form of typing judgments. Other recent efforts to accomodate generalized binding have been made in the context of Nominal Isabelle [12,26]. Among the biggest challenges in the formalization of binding, we include languages combining generalized binding with dependent types and hereditary substitution ([1]). In practice formalizations of such rich languages are attempted rarely and require non-trivial adaptations; however, it is with complex languages that abstract approaches like ours give the most ample benefits. For this reason, we are working on an extension of ostensibly named syntax to languages with signatures expressed in a generic way, in the style of [2].

In perspective, the ostensibly named approach seems to enjoy very desirable properties that would recommend its adoption as an alternative to more established techniques. These properties, however, come at a cost: without the help of automated tools, the burden of providing two formalization levels (concrete nameless and abstract ostensibly named) together with the associated proofs, will scare away most formalizers. Secondly, abstract recursion principles whose computational behaviour is expressed by an equational theory are not as convenient as the concrete ones available for inductive types. Lastly, defining recursive functions as instances of a recursion principle is quite unusual and can be tricky, although syntactic sugar can be used to make definitions more readable.

Such drawbacks could be greatly mitigated, if not completely eliminated, by means of specialized tool support. For this purpose, we plan to investigate in the future whether it is feasible to produce the overhead to an ostensibly named formalization programmatically from a declarative specification of a language with binding (similarly to what tools like DBgen [18] and LNgen [4] provide for pure de Bruijn and locally nameless formalizations). Taking advantage of recent works on specialized automation tactics for concrete encodings of binding (e.g. Autosubst [24]), we will also study the design of tactics and syntactic constructs to allow interactive theorem provers to present ostensibly named interfaces almost as if they were inductive types, automating computation of recursively defined functions and allowing definitions by pattern matching.

Acknowledgements. We would like to thank Benedikt Ahrens and Ralph Matthes for their valuable remarks. We also thank Randy Pollack for his useful comments about a preliminary version of this work.

References

1. Adams, R.: A Modular Hierarchy of Logical Frameworks. Ph.D. thesis, University of Manchester (2004)
2. Ahrens, B., Zsido, J.: Initial semantics for higher-order typed syntax in Coq. Journal of Formalized Reasoning 4(1) (2011), http://jfr.unibo.it/article/view/2066
3. Asperti, A., et al.: Formal metatheory of programming languages in the Matita interactive theorem prover. Journal of Automated Reasoning: Special Issue on the Poplmark Challenge 49(3), 427–451 (2012)
4. Aydemir, B., Weirich, S.: LNgen: Tool support for locally nameless representations. Tech. Rep. MS-CIS-10-24, University of Pennsylvania, Department of Computer and Information Science (2010)
5. Aydemir, B.E., et al.: Mechanized metatheory for the masses: The POPLMARK challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)
6. Berghofer, S., Urban, C.: Nominal inversion principles. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 71–85. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-71067-7_10
7. Bird, R.S., Paterson, R.: De Bruijn notation as a nested datatype. Journal of Functional Programming (1999)
8. Capretta, V., Felty, A.: Higher-order abstract syntax in type theory. In: Cooper, S.B., Geuvers, H., Pillay, A., Väänänen, J. (eds.) Logic Colloquium 2006. Lecture Notes in Logic, vol. 32, pp. 65–90. Cambridge University Press (2009)
9. Charguraud, A.: The locally nameless representation. Journal of Automated Reasoning 49(3), 363–408 (2012), http://dx.doi.org/10.1007/s10817-011-9225-2
10. Ciaffaglione, A., Scagnetto, I.: A weak HOAS approach to the POPLmark challenge. In: Kesner, D., Viana, P. (eds.) Proceedings Seventh Workshop on Logical and Semantic Frameworks, with Applications, Rio de Janeiro, Brazil, September 29-30. Electronic Proceedings in Theoretical Computer Science, vol. 113, pp. 109–124. Open Publishing Association (2013)
11. Gordon, A.D., Melham, T.: Five axioms of alpha-conversion. In: von Wright, J., Harrison, J., Grundy, J. (eds.) TPHOLs 1996. LNCS, vol. 1125, pp. 173–190. Springer, Heidelberg (1996), http://dx.doi.org/10.1007/BFb0105404
12. Huffman, B., Urban, C.: A new foundation for nominal isabelle. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 35–50. Springer, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-14052-5_5
13. McBride, C.: Dependently Typed Functional Programs and their Proofs. Ph.D. thesis, University of Edinburgh (1999)
14. McBride, C.: Elimination with a motive. In: Callaghan, P., et al. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 197–216. Springer, Heidelberg (2002)
15. McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. Journal of Automated Reasoning 23(3), 373–409 (1999), http://dx.doi.org/10.1023/A
16. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae 34, 381–392 (1972)
17. Pollack, R., Sato, M., Ricciotti, W.: A canonical locally named representation of binding. Journal of Automated Reasoning 49(2), 185–207 (2012), http://dx.doi.org/10.1007/s10817-011-9229-y

18. Polonowski, E.: Automatically generated infrastructure for de bruijn syntaxes. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 402–417. Springer, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-39634-2_29
19. Popescu, A., Gunter, E.L.: Recursion principles for syntax with bindings and substitution. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. pp. 346–358. ICFP 2011 ACM, New York (2011), http://doi.acm.org/10.1145/2034773.2034819
20. Pottier, F.: An overview of Cαml. Electronic Notes in Theoretical Computer Science 148(2), 27–52 (2006), Proceedings of the ACM-SIGPLAN Workshop on ML (ML 2005) ACM-SIGPLAN Workshop on ML 2005 (2005)
21. Pouillard, N.: Nameless, painless. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, pp. 320–332. ACM, New York (2011), http://doi.acm.org/10.1145/2034773.2034817
22. Ricciotti, W.: Theoretical and Implementation Aspects in the Mechanization of the Metatheory of Programming Languages. Ph.D. thesis, Università di Bologna (2011)
23. Sato, M., Pollack, R.: External and internal syntax of the lambda-calculus. J. Symb. Comput. 45(5), 598–616 (2010), http://dx.doi.org/10.1016/j.jsc.2010.01.010
24. Schäfer, S., Tebbi, T.: Autosubst: Automation for de Bruijn substitutions. In: 6th Coq Workshop (July 2014)
25. Urban, C.: Nominal techniques in Isabelle/HOL. Journal of Automated Reasoning 40(4), 327–356 (2008), http://dx.doi.org/10.1007/s10817-008-9097-2
26. Urban, C., Kaliszyk, C.: General bindings and alpha-equivalence in nominal isabelle. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 480–500. Springer, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-19718-5_25