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.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic**]: Lambda calculus and related systems

General Terms

Languages, Theory, Verification

Keywords

Proof assistants, Formalization, Representation of binding, Matita

1. INTRODUCTION

Arguably, issues related to the representation of binding structures are among the most significant choices 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: thus, concrete encodings are more usually employed.

Concrete encodings include some of the best known styles, like the de Bruijn nameless encoding [6] (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 Pollack and Sato [10], where a bound variable is represented by means of a name that is programmatically chosen depending on the structure of the term within the 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 [7, 1] and drawn a comparison in [9].

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 a very good access to the *implementation* but, crucially, we lack an *abstract* view on binding. We may say *if all you have is a hammer, everything looks like a nail.*

This paper describes a work 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. This is obtained by representing the terms of the object language as an abstract data type, that can only be manipulated by

means of the operations and logical properties provided by its module. An implementation, or model, of the abstract data type is provided separately and shown to validate all the expected properties. The details of this formalization 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; Section 4 extends the previous discussion to the level of type systems; finally Section 5 concludes.

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, $\boldsymbol{\lambda}_{\rightarrow}$). Similarly to Gordon and Melham's axiomatization in [4], 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.

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

Figure 1: Ostensibly named presentation of the simply typed lambda calculus.

Figure 1 shows the constructors of our presentation of the simply typed lambda calculus. We call this presentation *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,... The type of names $\mathbb{A}$ is an arbitrary infinite type with decidable equality, providing 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:

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

$\nu x.u$ substitutes a hole for any $\mathsf{Par}\ x$ occuring 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.

An abstract operation FV takes in input a term or a context and returns the list of free names used in that term or context. This allows us to formulate the following properties of context forming operations and Lam abstractions:

$$
\begin{array}{rl}
(\nu x.u)\lceil x \rceil = u & \\
\nu x.(u\lceil x \rceil) = u & \text{if } x \notin \mathrm{FV}(u) \\
\mathsf{Lam}\ x\ \sigma\ (u\lceil x \rceil) = \mathsf{Lam}\ y\ \sigma\ (u\lceil y \rceil) & \text{if } x, y \notin \mathrm{FV}(u)
\end{array}
$$

In particular the last property expresses the fact that Λ_0 is canonical, identifying α-convertible terms.

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.

$$
\begin{array}{lll}
\mathcal{R}_{\Lambda_0} : & \forall T : \textbf{Type}, C : \mathit{list}\ \mathbb{A}. & \\
 & (\mathbb{A} \Rightarrow T) \Rightarrow & \\
 & (\Lambda_0 \Rightarrow \Lambda_0 \Rightarrow T \Rightarrow T \Rightarrow T) \Rightarrow & \\
 & (\forall x : \mathbb{A}, \sigma : tp, v : \Lambda_1. & \\
 & \quad x \notin \mathrm{FV}(v) @ C \Rightarrow T \Rightarrow T) \Rightarrow & \\
 & \Lambda_0 \Rightarrow T &
\end{array}
$$

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

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

often defined as follows:

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

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 [11]).

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

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

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). Since the principle is part of our ostensibly named interface, it is opaque and we need to provide properties expressing its computational behaviour. For parameters and applications, this is reasonably simple:

$$\mathcal{R}_{\Lambda_0}\ T\ C\ f_{\mathsf{Par}}\ f_{\mathsf{App}}\ f_{\mathsf{Lam}}(\mathsf{Par}\ x) = f_{\mathsf{Par}}\ x$$

$$\begin{array}{l} \mathcal{R}_{\Lambda_0}\ T\ C\ f_{\mathsf{Par}}\ f_{\mathsf{App}}\ f_{\mathsf{Lam}}(\mathsf{App}\ u\ v) = \\ \quad f_{\mathsf{App}}\ u\ v\ (\mathcal{R}_{\Lambda_0}\ T\ C\ f_{\mathsf{Par}}\ f_{\mathsf{App}}\ f_{\mathsf{Lam}}\ u) \\ \quad\quad (\mathcal{R}_{\Lambda_0}\ T\ C\ f_{\mathsf{Par}}\ f_{\mathsf{App}}\ f_{\mathsf{Lam}}\ v) \end{array}$$

However a similar approach is not sound in the Lam case: we can convince ourselves of this by means of the following example:

$$\begin{array}{l} \mathcal{R}_{\Lambda_0}\ \mathbb{A}\ C\ f_{\mathsf{Par}}\ f_{\mathsf{App}}\ (\lambda y, \_, \_, \_, \_.y)\ (\mathsf{Lam}\ x\ \sigma\ (u\lceil x \rceil)) \\ \quad \stackrel{?}{=} f_{\mathsf{Lam}}\ x\ \sigma\ u\ H \\ \quad\quad (\mathcal{R}_{\Lambda_0}\ \mathbb{A}\ C\ f_{\mathsf{Par}}\ f_{\mathsf{App}}\ (\lambda y, \_, \_, \_, \_.y)\ (u\lceil x \rceil)) \\ \quad = x \end{array}$$

(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. For this reason, we need to express computation by means of a more involved property:

$$\begin{array}{l} \forall T : \mathbf{Type}.\forall U : T \Rightarrow \mathbf{Type}. \\ \quad \forall x, \sigma, u, C, H_{\mathsf{Par}}, H_{\mathsf{App}}, H_{\mathsf{Lam}}.x \notin \mathrm{FV}(u) \Rightarrow \\ \quad (\forall y, H_y. \\ \quad\quad U\ (H_{\mathsf{Lam}}\ C\ y\ \sigma\ u\ H_y \\ \quad\quad\quad (\mathcal{R}_{\Lambda_0}\ T\ C\ H_{\mathsf{Par}}\ H_{\mathsf{App}}\ H_{\mathsf{Lam}}\ (u\lceil y \rceil)))) \Rightarrow \\ \quad U\ (\mathcal{R}_{\Lambda_0}\ T\ C\ H_{\mathsf{Par}}\ H_{\mathsf{App}}\ H_{\mathsf{Lam}}\ (\mathsf{Lam}\ x\ \sigma\ (u\lceil x \rceil)))) \end{array}$$

This property is more constraining than the previous equations: it 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.

3. A LOCALLY NAMELESS MODEL

A locally nameless representation [3] 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*:

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

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 $\mathsf{var}\ k$ is considered to be bound to the $(k+1)$-th outer abstraction.

In such a representation, indices whose value is 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:

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

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.

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

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

We are now able to give an interpretation of ostensibly named terms and contexts:

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

$$\begin{array}{lcl} [\![\nu x.u]\!] & \triangleq & \nu_0 x.[\![u]\!] \\ [\![u\lceil x \rceil]\!] & \triangleq & [\![u]\!]\lceil x \rceil_0 \end{array}$$

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

Lemma 3.1.

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* $[\![u\lceil x \rceil]\!] : ctx\ i$

Lemma 3.2. *(α-conversion)*

If $x, y \notin [\![\text{FV}(u)]\!]$, *then* $[\![\mathsf{Lam}\ x\ \sigma\ (u\lceil x \rceil)]\!] = [\![\mathsf{Lam}\ y\ \sigma\ (u\lceil y \rceil)]\!]$

Lemma 3.3.

1. $[\![(\nu x.u)\lceil x \rceil]\!] = [\![u]\!]$
2. *if* $x \notin \text{FV}(u)$, *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 (later showing that it can be lifted to proper terms). Here we only give the idea behind the definition in the key case of abstractions, due to tricky dependently typed details that require too much space for a thorough discussion.

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 $x = \varphi(C@\text{FV}(u))$ and open u with respect to that name; we then perform the recursive call on the opened $u\lceil x \rceil_0$. In symbols:

$$\begin{array}{l} \text{pretm\_rec } T\ C\ f_{\mathsf{var}}\ f_{\mathsf{par}}\ f_{\mathsf{app}}\ f_{\mathsf{Lam}}\ (\mathsf{abs}\ \sigma\ u) \triangleq \\ \quad \text{let } x := \varphi(C, \text{FV}(u)) \text{ in} \\ \quad f_{\mathsf{Lam}}\ x\ \sigma\ (u\lceil x \rceil_0) \ldots \\ \qquad (\text{pretm\_rec } T\ C\ f_{\mathsf{var}}\ f_{\mathsf{par}}\ f_{\mathsf{app}}\ f_{\mathsf{Lam}}\ (u\lceil x \rceil_0)) \end{array}$$

The real 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 a typical theorem prover).

4. 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 quite well understood. Most locally nameless formalizations employ the following concrete introduction rule for lambda abstractions:

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

Following [10], 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 $\mathsf{abs}\ \sigma\ u\,[^{\mathsf{var}\ 0}/_{x}]$. 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 (typechecking 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 [5], 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 derivation 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:

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

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:

$$\frac{\left(\begin{array}{l}\forall x. x \notin \mathrm{dom}(\Gamma) @ \mathrm{FV}(u) \Rightarrow\\ \quad \langle x, \sigma\rangle :: \Gamma \vdash u\,[^{\mathsf{par}\ x}/_{\mathsf{var}\ 0}] : \tau)\end{array}\right)}{\Gamma \vdash \mathsf{abs}\ \sigma\ u : \sigma \to \tau}\ (\text{LN-T-Abs'})$$

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:

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

4.1 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.

The typing rules, shown in Figure 2, 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 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.

The ostensibly named induction principle we associate to these rules (Figure 3) is more interesting. The induction

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

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

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

Figure 2: Typing rules for $\boldsymbol{\lambda}_{\to}$, ostensibly named style.

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

Figure 3: Rule induction for the $\boldsymbol{\lambda}_{\to}$ typing derivations, ostensibly named style.

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.

4.2 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 4. In the nameless encoding, typing environments are just lists of types: we denote them as $\gamma, \gamma', \ldots$.

Since in this presentation no named parameter appears, con-

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

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

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

Figure 4: Typing rules for $\boldsymbol{\lambda}_{\to}$, pure de Bruijn style.

text 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 5), an ideal model for the abstract rules of the previous section.

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

Figure 5: 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:

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

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

where:

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

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 to refer to lists, in this case to a list of names. Our second task is to model the rules of Figure 2 as instances of their nameless counterparts. This is expressed by the following lemma:

LEMMA 4.1.

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* $x \notin \mathrm{dom}(\Gamma)$ *implies* $[\![\langle x, \sigma\rangle :: \Gamma \vdash_O u : \tau]\!]$, *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* $[\![\Gamma \vdash_O u\ v : \tau]\!]$.

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

THEOREM 4.2. *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 extended vector notation in the form $\overrightarrow{x_n}$ indicates 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 3.3).

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

$$\begin{array}{c} \sigma_0 :: \gamma_0 \vdash_D u_0 : \tau_0 \\ \hat{P}(\sigma_0 :: \gamma_0, u_0, \tau_0) \end{array}$$

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

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

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

5. 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.

Even though our internal representation of binding structures employs nameless dummies, other models are possible as long as they are canonical, the most obvious alternative being the canonical locally named representation. 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 our discussion, we have not mentioned the induction principle over terms. It is not necessary to provide axioms for it, as it can be obtained from (a dependently typed version of) the recursion principle $\mathcal{R}_{\Lambda_0}$, and later strengthened by means of the usual technique of equivariance. However an alternative strategy which is more coherent with the techniques shown in this paper is to formalize a well-formedness judgment for terms by means of inductive rules, similarly to our formalization of typing, and perform strong induction on well-formedness derivations. Induction on well-formedness derivations has been a common idiom for a long time (especially in locally nameless formalizations) and in our case has the benefit of not needing any reasoning about permutations.

We have not considered whether the ostensibly named style is sufficiently expressive for real formalizations. This is a report about a novel technique and we still have to investigate whether complex proofs like confluence, subject reduction, or decidability of typechecking are feasible or even possible without assuming stronger axioms. However we have carried out simple proofs about the substitution operation and the weakening of typing in the simply typed lambda calculus. Quite remarkably, the weakening proof is as easy as in a locally nameless formalization, despite the fact that internally the rules use a de Bruijn representation, which would normally require complex arguments about permutations of the typing environment.

Moreover, our interpretation of rule induction into the de Bruijn model of the type system essentially proves a strong induction principle without resorting to the usual technique of equivariance, but only employing the operations of variable opening and variable closing. While this is hardly an improvement over equivariance, we believe it provides an alternative, interesting point of view on the topic.

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 substantial 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.

In the future, we plan to investigate 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 [8] and LNgen [2] provide for pure de Bruijn and locally nameless formalizations). 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 Randy Pollack for his valuable remarks about a preliminary version of this work.

6. REFERENCES

[1] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. 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.

[2] B. Aydemir and S. Weirich. Lngen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, University of Pennsylvania, Department of Computer and Information Science, 2010.

[3] A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012.

[4] A. Gordon and T. Melham. Five axioms of alpha-conversion. In G. Goos, J. Hartmanis, J. Leeuwen, J. Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer Berlin Heidelberg, 1996.

[5] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3):373–409, 1999.

[6] N. G. de Bruijn. 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.

[7] R. Pollack, M. Sato, and W. Ricciotti. A canonical locally named representation of binding. *Journal of Automated Reasoning*, 49(2):185–207, 2012.

[8] E. Polonowski. Automatically generated infrastructure for De Bruijn syntaxes. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 402–417. Springer Berlin Heidelberg, 2013.

[9] W. Ricciotti. *Theoretical and implementation aspects in the mechanization of the metatheory of programming languages*. PhD thesis, Università di Bologna, 2011.

[10] M. Sato and R. Pollack. External and internal syntax of the lambda-calculus. *J. Symb. Comput.*, 45(5):598–616, May 2010.

[11] C. Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.