Mechanized Reasoning about Languages with Binding

Martijn Vermaat

Abstract

Formalization of programming language metatheory is an increasingly active research field. Most languages have a notion of variable binding which presents some of the main technical difficulties in mechanical reasoning about languages. Traditional concrete representations with named variables have the issues of α-equivalence and α-conversion. An alternative representation uses de Bruijn indices, eliminating the issue of α-equivalence but introducing the need for lifting of indices. The locally nameless representation offers a hybrid approach with names for free variables and the de Bruijn indices for bound variables. We study these three representations with untyped λ-calculus in the Coq proof assistant as a running example. We also briefly consider two real-world cases of mechanized reasoning about languages with variable binding.

1 Introduction

Theory and practice in programming language research are in large part two separate worlds. Language designers often take a pragmatic approach, where the focus is on the observable behavior of implementations and not much attention is paid to prove properties of the language in a formal way. Research on the theory of programming languages has a long tradition and provides us with a large base of techniques and results, but is mostly isolated in academic settings. Applying formal reasoning to current programming language development has the promise of rigorously defined practical languages with provable properties, but is unfortunately not commonplace.

Many tools have become available to aid formal reasoning in a mechanical way, such as proof assistants and theorem provers. Researchers can use these tools to obtain machine-checked and reusable proofs while language designers can use them to formally define languages, clearing the way for formal reasoning. There are several issues holding back large-scale application of such tools, both technical and non-technical. In this literature study, we focus on one specific technical aspect.

A major difficulty in formal reasoning about languages are *bindings* and *bound variables*. Traditionally, named variables are used in the representation of terms and reasoning is done modulo α-equivalence. While this approach yields easy to

read and sufficiently precise reasoning on paper, mechanical reasoning with named variables is hard: identification of α-equivalent terms has to be made precise and substitution requires renaming of bound variables in order to prevent accidental binding of free variables.

Several approaches exist to remedy these problems. Some of them try to make it easier to work with named variables, while others propose alternate representations for bindings and variables. We study some of these approaches in the context of mechanical reasoning, focussing on concrete representations for terms using *names* and *numbers* for variables. The approaches are illustrated with applications using the Coq proof assistant [11, 5].

2 Representing Bindings

The most common problems associated with variable bindings are *α-equivalence* and *α-conversion*. These are actually notions related to the traditional representation where names are used for variables, and may or may not be relevant issues with other representations. Some representations however, have analogue notions, for example *lifting* of so-called *de Bruijn indices* is dual to α-conversion.

We use untyped pure λ-calculus [4] to introduce three concrete representations for terms using names and numbers for variables. To illustrate, the representations are implemented in the Coq proof assistant, where we make use of a type for names on which equality is decidable.

```
Variable name : Set.
Hypothesis eq_name : forall (x y : name), {x = y} + {x <> y}.
```

We also assume a way to generate fresh names with `fresh_name`, which returns a name that is not in a given list of names.

```
Variable fresh_name : (list name) -> name.
Hypothesis fresh_name_fresh : forall l, ~ In (fresh_name l) l.
```

2.1 Named Variables

Untyped λ-calculus has function application and abstraction over variables. Using named variables, its terms can be represented with the grammar

$$\begin{array}{lll} M ::= x & & \text{variable} \\ \quad \mid \lambda x.\, M & & \text{abstraction} \\ \quad \mid M\ M & & \text{application} \end{array}$$

where x ranges over a countably infinite set of names. An abstraction introduces a name, occurrences of which are said to be bound by it in its *scope*. The scope of an abstraction $\lambda x.M$ is the body M minus any subterm of M that is an abstraction over the same name x.

α-equivalence and α-conversion

α-conversion is the process of renaming bound variables, such that abstractions bind exactly the variables they did originally. Two terms are α-equivalent if they are the same modulo α-conversion. For example, the following are α-equivalent pairs of terms:

$\lambda x.\ x$	$(\lambda z.\ z\ x)\ z$	$\lambda x.\ \lambda y.\ x\ (y\ z)$
$\lambda z.\ z$	$(\lambda y.\ y\ x)\ z$	$\lambda y.\ \lambda x.\ y\ (x\ z)$

When reasoning about terms, it is often needed to identify α-equivalent terms. This can result in a lot of extra work if the reasoning has to be rigorous, as is the case in a tool like a proof assistant.

Manipulation of terms can result in accidental binding of variables that where previously not bound or bound by another abstraction. Consider substituting x for y in $\lambda x.\ y\ x$. The free variable x becomes bound by the abstraction over x and looses its possible contextual meaning. This is a phenomenon that should be avoided. A possible solution is applying α-conversion to rename the bound variable x to a fresh name, say z, before performing the substitution. Substituting x for y in $\lambda z.\ y\ z$ does not bind x.

Barendregt's variable convention is a way to avoid accidental binding of variables by maintaining the invariant that in any context, all bound variables in a term are chosen to be different from the free variables. It is however not straightforward to implement this invariant directly in mechanical developments and its effects are therefore often mimicked by other technicalities (e.g., explicit renaming or using disjoint sets for bound and free variables).

Substitution

Let us consider substitution some more. Substituting N for x in M is usually defined as

$$
\begin{aligned}
x[N/x] &= N \\
y[N/x] &= y && x \neq y \\
(\lambda y.\ M')[N/x] &= \lambda y.\ M'[N/x] && x \neq y \text{ and } y \text{ not free in } N \\
(M_1\ M_2)[N/x] &= M_1[N/x]\ M_2[N/x]
\end{aligned}
$$

which is cannot be directly implemented as an operation without using the variable convention, because of the declarative side condition in the case of abstraction. Of course, α-conversion can be used to eliminate the undefined case, yielding

$$
\begin{aligned}
x[N/x] &= N \\
y[N/x] &= y && x \neq y \\
(\lambda y.M')[N/x] &= \lambda z.M'[z/y][N/x] && z \text{ not free in } N, M' \\
(M_1\ M_2)[N/x] &= M_1[N/x]\ M_2[N/x]
\end{aligned}
$$

Note that this is not a *structurally* recursive definition, since $M'[z/y]$ is not generally a subterm of $\lambda y.M'$. Structural recursion is preferred over recursion on some other measure (e.g. term size), especially in a tool like Coq where structural induction principles are automatically generated for inductive definitions and structurally recursive functions don't need explicit termination proofs.

Named Variables in Coq

Using named variables, λ-calculus terms can be represented in the Coq proof assistant with the inductive type `term`. We also define simple renaming of variables, to be used in other definitions.

```
Inductive term : Set :=
  | Var : name -> term
  | Abs : name -> term -> term
  | App : term -> term -> term.

Fixpoint rename (n m : name) (t : term) {struct t} : term :=
  match t with
  | Var x =>   if eq_name x n then Var m else t
  | Abs x b => Abs (if eq_name x n then m else x) (rename n m b)
  | App f a => App (rename n m f) (rename n m a)
  end.
```

As a first try in translating the substitution operation to Coq, the following definition will fail because, as noted above, it is not structurally recursive on `t`.

```
Fixpoint subst_ill (s : term) (n : name) (t : term)
    {measure size t} : term :=
  match t with
  | Var x   => if eq_name x n then s else t
  | Abs x b => let z := fresh_name (n :: (free_vars s)
                                       ++ (free_vars b))
               in
               Abs z (subst_ill s n (rename x z b))
  | App f a => App (subst_ill s n f) (subst_ill s n a)
  end.
```

Note that for simplicity we rename every abstraction. In practice, this could of course be improved.

A solution to the failing definition `subst_ill` is to use the term size as a well-founded recursion measure. This definition of `subst` requires us to additionally provide three short proofs for Coq to be assured of its termination.

```
Fixpoint size (t : term) : nat :=
  match t with
  | Var _   => 0
  | Abs _ b => S (size b)
  | App f a => S (size f + size a)
  end.
```

```
Lemma size_rename : forall n m t, size (rename n m t) = size t.
Proof.
induction t as [x | |]; simpl;
[destruct (eq_name x n); reflexivity | congruence | congruence].
Qed.

Function subst (s : term) (n : name) (t : term)
    {measure size t} : term :=
  match t with
  | Var x   => if eq_name x n then s else t
  | Abs x b => let z := fresh_name (n :: (free_vars s)
                                          ++ (free_vars b))
               in
               Abs z (subst s n (rename x z b))
  | App f a => App (subst s n f) (subst s n a)
  end.
intros. rewrite size_rename. auto.
intros. simpl. destruct f; omega.
intros. simpl. destruct f; omega.
Defined.
```

Simultaneous Substitution

In search of a structural recursive substitution operation, Stoughton [14] suggests the *simultaneous substitution* $M\sigma$

$$\begin{aligned} x\sigma &= \sigma x \\ (\lambda x.M')\sigma &= \lambda y.(M'\ \sigma[y/x]) \qquad\qquad y \text{ not free in } M', \sigma \\ (M_1\ M_2)\sigma &= M_1\sigma\ M_2\sigma \end{aligned}$$

where

$$\sigma[N/y]\ x = \begin{cases} N & \text{if } x = y, \\ \sigma x & \text{otherwise} \end{cases}$$

Substituting N for x in M is now $M\ \iota[N/x]$ with ι the identity substitution. This structurally recursive operation is easily implemented in Coq, as shown by the following definitions.

```
Fixpoint apply_subst (l : list (term*name)) (n : name)
    {struct l} : term :=
  match l with
  | nil        => Var n
  | (t, x)::l' => if eq_name x n then t else apply_subst l' n
  end.
```

```
Fixpoint sim_subst (l : list (term*name)) (t : term)
    {struct t} : term :=
  match t with
  | Var x =>   apply_subst l x
  | Abs x b => let z := fresh_name ((free_vars_sub l)
                                    ++ (free_vars b))
               in
               Abs z (sim_subst ((Var z, x) :: l) b)
  | App f a => App (sim_subst l f) (sim_subst l a)
  end.
```

Simple substitution can now be written as a special case of simultaneous substitution.

```
Definition subst' (s : term) (n : name) (t : term) : term :=
  sim_subst ((s, n) :: nil) t.
```

Summary

The above examples illustrate that α-conversion to prevent accidental binding of variables is not a trivial issue in an automated setting. The assumption of being able to generate fresh names might be troublesome and standard operations such as simple substitution are not structurally recursive.

Perhaps an even greater problem is that of α-equivalence, depending on the task at hand. Terms can be treated as their α-equivalence class, properties of terms with bound variables can sometimes be parameterized by their names, or terms can be translated to some canonically named variant, to name just a few techniques to overcome this problem. Most of these require a lot of tedious work that distracts from the actual task at hand.

An obvious advantage of using a representation with named variables is that it is the same as the usual on-paper notation. It can therefore be read by anyone familiar with traditional work and there is no need to justify the relation between the original object language and the way it is represented.

2.2 de Bruijn Indices

As an alternative representation for λ-calculus terms, de Bruijn [6] proposes a representation using natural numbers instead of names. A variable is represented by the number of binders between itself and its abstraction (called a de Bruijn index), yielding the grammar

$$
\begin{array}{lll}
M ::= n & & \text{variable} \\
\quad \mid \lambda.\, M & & \text{abstraction} \\
\quad \mid M\ M & & \text{application}
\end{array}
$$

where n ranges over the natural numbers. The following are some example terms in the traditional representation (top) and in the representation with de Bruijn indices:

$\lambda x.\ x$	$(\lambda z.\ z\ x)\ z$	$\lambda x.\ \lambda y.\ x\ (y\ z)$
$\lambda.\ 0$	$(\lambda.\ 0\ 1)\ 2$	$\lambda.\ \lambda.\ 1\ (0\ 2)$

A major advantage of using de Bruijn indices in machine-assisted reasoning is that α-equivalence is just term equality.

Lifting

Abstractions no longer introduce names, eliminating the need for α-conversion of bound variables. However, manipulation of terms may still accidentally capture variables when a de Bruijn index is moved over an abstraction. In that case the index should be updated in order to still refer to the original binding site. This operation is called lifting. We denote the simple lifting operation of incrementing all free variables in M by $\uparrow M$. For example, $\lambda.\ \uparrow\lambda.\ 1\ (0\ 2)$ is $\lambda.\lambda.\ 2\ (0\ 3)$.

Substitution

An example of an operation where lifting is used, is substitution. Substituting N for free de Bruijn index n can be defined

$$
\begin{aligned}
n[N/n] &= N \\
m[N/n] &= m && m \neq n \\
(\lambda.M')[N/n] &= \lambda.M'[\uparrow N/n+1] \\
(M_1\ M_2)[N/n] &= M_1[N/n]\ M_2[N/n]
\end{aligned}
$$

which is structurally recursive. Note that the lifting of N in the case of abstraction is needed because N is moved into the abstraction $\lambda.M'$. For the same reason, recursively substituting $\uparrow N$ in the body M' has to be done for indices $n + 1$ instead of n.

de Bruijn Indices in Coq

The following is a datatype definition in Coq for λ-calculus terms using de Bruijn indices:

```
Inductive term : Set :=
  | Var : nat -> term
  | Abs : term -> term
  | App : term -> term -> term.
```

Substitution is easily implemented using lifting.

```
Fixpoint lift (l : nat) (t : term) {struct t} : term :=
  match t with
  | Var n   => Var (if le_lt_dec l n then (S n) else n)
  | Abs u   => Abs (lift (S l) u)
  | App u v => App (lift l u) (lift l v)
end.

Fixpoint subst (s : term) (n : nat) (t : term)
    {struct t} : term :=
  match t with
  | Var m   => if eq_nat_dec n m then s else t
  | Abs u   => Abs (subst (lift 0 s) (S n) u)
  | App u v => App (subst s n u) (subst s n v)
end.
```

More complex operations on terms may involve quite some manipulation of indices. Although these manipulations are easier to carry out than α-conversion in the named variables representation, they are still cumbersome and obfuscate proof scripts.

Summary

The most obvious advantage of a representation with de Bruijn indices is that α-equivalence is just term equality. This does away with the problems usually associated with α-equivalence, a major win in automated reasoning. Additionally, lifting is easier than α-conversion (but still tedious) and there is no need to generate fresh names.

The representation diverges quite far from the traditional representation using named variables. Therefore, it is arguably harder to read and the relation between the object language and formal developments using de Bruijn indices might need some argumentation.

2.3 Locally Nameless Representation

A way to avoid accidental capturing of variables and thus the need for α-conversion is to make sure free and bound variables are drawn from disjoint sets. This strategy is implemented by the *locally nameless* representation, where free variables are represented by names and bound variables are represented by de Bruijn indices [12], effectively obeying Barendregt's variable convention. The resulting grammar is

$M ::= x$	free variable
$\mid n$	bound variable
$\mid \lambda.\, M$	abstraction
$\mid M\ M$	application

where x ranges over a countably infinite set of names, and n over the natural numbers. The following are some example terms in the traditional representation (top), using de Bruijn indices (middle), and in the locally nameless representation:

$\lambda x.\ x$	$(\lambda z.\ z\ x)\ z$	$\lambda x.\ \lambda y.\ x\ (y\ z)$
$\lambda.\ 0$	$(\lambda.\ 0\ 1)\ 2$	$\lambda.\ \lambda.\ 1\ (0\ 2)$
$\lambda.\ 0$	$(\lambda.\ 0\ x)\ z$	$\lambda.\ \lambda.\ 1\ (0\ z)$

Just like pure de Bruijn indices, the locally nameless representation has α-equivalence as term equality. What sets it apart from both representations discussed above, is that there is no need to worry about accidental capturing of variables, since the invariant is maintained that all bound variables are different from free variables. There is, unfortunately, a cost to this.

Freshening

Consider the term $\lambda.\ \lambda.\ 1\ (0\ z)$. How should we proceed in order to reason about the body of the outer abstraction? Surely we cannot say much about the verbatim subterm $\lambda.\ 1\ (0\ z)$, since it is not locally nameless. What happened is that a previously bound variable was taken from the scope of its binder, making it a free variable. To maintain the locally nameless invariant, this variable should now be represented by a name. The substitution of a name for index 0 is called *freshening*. In this example, $\mathit{freshen}(\lambda.\ 1\ (0\ z), v)$ would yield $\lambda.\ v\ (0\ z)$.

Substitution

Substitution of terms for free variables in a locally nameless term means substitution of terms for names.

$$
\begin{aligned}
x[N/x] &= N \\
y[N/x] &= y && x \neq y \\
n[N/x] &= n \\
(\lambda.M')[N/x] &= \lambda.M'[N/x] \\
(M_1\ M_2)[N/x] &= M_1[N/x]\ M_2[N/x]
\end{aligned}
$$

There is no need for renaming, because abstractions do not bind names. There is no need for lifting, because there are no free de Bruijn indices.

To implement the freshening operation, it must also be possible to replace de Bruijn indices by a term. Substitution for free variables thus has a dual notion for

bound variables.

$$
\begin{aligned}
x[N/n] &= x \\
n[N/n] &= N \\
m[N/n] &= m && m \neq n \\
(\lambda.M')[N/n] &= \lambda.M'[N/n+1] \\
(M_1\ M_2)[N/n] &= M_1[N/n]\ M_2[N/n]
\end{aligned}
$$

Freshening M with name x can now be written $\mathit{freshen}(M, x) = M[x/0]$.

Locally Nameless Representation in Coq

λ-calculus terms can be represented locally nameless in Coq with the following datatype:

```
Inductive term : Set :=
  | FreeVar  : name -> term
  | BoundVar : nat  -> term
  | Abs      : term -> term
  | App      : term -> term -> term.
```

Substitution of terms for free variables is defined by `subst_free`. The `subst_bound` function is meant to be used only in the definition of `freshen`.

```
Fixpoint subst_free (s : term) (x : name) (t : term)
    {struct t} : term :=
  match t with
  | FreeVar y  => if eq_name x y then s else t
  | BoundVar n => t
  | Abs b      => Abs (subst_free s x b)
  | App f a    => App (subst_free s x f) (subst_free s x a)
end.

Fixpoint subst_bound (s : term) (n : nat) (t : term)
    {struct t} : term :=
  match t with
  | FreeVar x  => t
  | BoundVar m => if eq_nat_dec m n then s else t
  | Abs b      => Abs (subst_bound s (S n) b)
  | App f a    => App (subst_bound s n f) (subst_bound s n a)
end.

Definition freshen (t : term) (x : name) : term :=
  subst_bound (FreeVar x) 0 t.
```

Summary

In a way, the locally nameless representation tries to be a *best of both worlds* combination of named variables and de Bruijn indices. Inherited from pure de Bruijn

indices are the pleasant properties that α-equivalence is term equality and bound variables do not need renaming. Using names, however, feels like a natural way to represent free variables that presumably take some meaning from a context, and as such makes locally nameless terms arguably easier to read than their pure de Bruijn equivalents.

The main technical difficulty we get is that of maintaining the invariant that all terms are de Bruijn-closed, sometimes requiring freshening. A consequence is that we might lose structural recursion for some operations (an abstraction body might not be a subterm after freshening). This does not apply to our current definition of substitution, where freshening is omitted since it can easily be seen that the invariant is only needed for the substituted term and not for the term that is recursed on. For more complex operations, however, this may not be the case.

3 Implementations

In this section we discuss two real-world cases of representing languages with binding. Again, we focus on applications using the Coq proof assistant.

3.1 The POPLmark Challenge

In the introduction we debated that mechanized formalization of programming language metatheory has the potential of bringing the academic tradition of rigorous formal reasoning to the implementation of everyday programming languages. The POPLmark Challenge [3] proposes a set of benchmarks to measure progress in this area, under the slogan *mechanized metatheory for the masses*. The benchmarks are based on the metatheory of System $F_{<:}$ [8], a polymorphic λ-calculus with subtyping, and revolve in large part around binding. Several solutions, some of them partial, to the challenge have since been submitted. We discuss some of the solutions to part 1a that are implemented in the Coq proof assistant.

POPLmark 1a considers only the type language of System $F_{<:}$ and consists of proving transitivity of subtyping. The grammar

$T ::= X$	type variable
$\mid$ `Top`	maximum type
$\mid T \to T$	type of functions
$\mid \forall X{<:}T.T$	universal type
$\Gamma ::= \emptyset$	empty type environment
$\mid \Gamma, X{<:}T$	type variable binding

defines types T and type environments Γ. The following are the five subtyping rules

of System $F_{<:}$:

$$\frac{}{\Gamma \vdash S <: \mathsf{Top}}\ \text{SA-Top} \qquad \frac{}{\Gamma \vdash X <: X}\ \text{SA-Refl-TVar}$$

$$\frac{X <: U \in \Gamma \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}\ \text{SA-Trans-TVar}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2}\ \text{SA-Arrow}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: S_1 . S_2 <: \forall X <: T_1 . T_2}\ \text{SA-All}$$

Stump: Named Variables

Stump [15] implements the type language of System $F_{<:}$ using names for bound variables. Two main techniques are employed to avoid difficulties with binding of named variables. First, free and bound variables are taken from disjoint sets, implementing Barendregt's variable convention. This reduces capture-avoiding substitution to the simple method of grafting. Second, the SA-All rule is slightly adjusted to use a common variable in the bodies of the two universal types:

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X <: T_1 \vdash S_2[X/X_1] <: T_2[X/X_2]}{\Gamma \vdash \forall X_1 <: S_1 . S_2 <: \forall X_2 <: T_1 . T_2}\ \text{SA-All}$$

This avoids α-equivalence issues in comparing S_2 and T_2.

Vouillon: de Bruijn Indices

A very clear implementation (even suggested as baseline by the POPLmark team) using de Bruijn indices is the solution by Vouillon [17]. As is to be expected with de Bruijn indices, a lot of the code deals with shifting of indices. However, this low-level manipulation of terms is largely kept from polluting the definitions and lemma statements, which closely follow the paper presentation. The proofs are by induction on the size of types instead of by structural induction. Unfortunately, this solution has no accompanying paper.

Leroy: Locally Nameless Representation

The solution by Leroy [10] uses a locally nameless representation of types. Two substitution operations, one for names and one for numbers, are used and no renaming or lifting is needed. As in the solution by Vouillon, proofs are not by structural induction but by induction on the size of types. A large part of the code deals with

swaps, a special case of renaming. Swaps are used in equivariance proofs. Charguéraud [9] improves on this solution in some ways. His solution uses proofs by induction on the derivation of the well-formation relation instead of by induction on the size of types.

3.2 Engineering Formal Metatheory

Engineering Formal Metatheory [1] proposes a complete style for mechanically formalizing programming language metatheory. The approach builds on the experience from solutions to the POPLMARK Challenge and employs a locally nameless representation. Reasoning about equivariance is obviated by using cofinite quantification of names in inductive definitions of relations on terms. The style is implemented using COQ and applied to carry out several large developments (e.g., parts of the POPLMARK Challenge, type soundness of core ML, and subject reduction of the Calculus of Constructions).

LNGEN [2] takes this style of formalizing metatheory and provides a tool for automatically generating the COQ infrastructure for such formalizations. The input language of LNGEN is the same as that of OTT, a tool for translating language specifications to definitions in proof assistants such as COQ. The goal of LNGEN is to automate as much as possible of language formalization in the locally nameless style. OTT specifications are used to generate recursion schemes and infrastructure lemmas, allowing users to focus on the more interesting aspects of their developments.

4 Discussion

4.1 Related Work

We presented three common concrete representations for terms with bindings. An alternative is the *nominal* representation based on nominal logic [13], using names for variables and equivalence classes for terms, in accordance with what one usually does on paper. Nominal reasoning is implemented in NOMINAL ISABELLE [16] which is actively used.

Another alternative representation uses *higher-order abstract syntax* [7], where the issues of binding are delegated to the meta language: variables are represented by meta-variables and bindings are represented by meta-bindings. This approach leads to statements that are quite different from what we write on paper, but gives us α-equivalence and capture-avoiding substitution for free.

4.2 Conclusion

Representation of languages with binding enjoys active research, and a number of approaches have been successfully applied to large-scale formalizations. Using names for bound variables in a concrete representation is close to what we do

on paper, but probably brings too much technical trouble in large formalizations. By using de Bruijn indices, much of the technical trouble reappears in another form that is usually easier to work with. A disadvantage of de Bruijn indices is that they are far away from on-paper notations. The locally nameless representation tries to combine the best of both worlds, but requires quite some boilerplate code. Fortunately, tools like LNgen might make using this representation feasible. Although not discussed here, nominal representations provide a promising approach and are under active study.

It is impossible to choose a definite preferred representation for language formalization in general, since different formalizations have different needs. For example, one can imagine that for a language implementation such as a compiler it adds no value to its users if the language representation is close to its concrete syntax. On the other hand, metatheory that is studied by many researchers does benefit from a representation that is close to the original representation of the language. Furthermore, the environment of a formalization (e.g., some proof assistant) might suit one representation better than another.

References

[1] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 3–15, 2008.

[2] B. Aydemir and S. Weirich. Lngen: Tool support for locally nameless representations. Draft, `http://www.cis.upenn.edu/~baydemir/papers/lngen/`, 2009.

[3] B. E. Aydemir, A. Bohannon, M. Fairbairn, N. J. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics*, pages 50–65, 2005.

[4] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. 1984.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[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*, pages 381–392, 1972.

[7] V. Capretta and A. Felty. Higher order abstract syntax in type theory. In *Logic Colloquium*, 2006.

[8] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of System F with subtyping. In *Information and Computation*, volume 109, pages 750–770, 1994.

[9] A. Charguéraud. The locally nameless representation. To appear in Journal of Automated Reasoning, `http://arthur.chargueraud.org/research/2009/ln/`, 2009.

[10] X. Leroy. A locally nameless solution to the POPLmark challenge. Research Report RR-6098, INRIA, 2007.

[11] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2009. Version 8.2, `http://coq.inria.fr/doc/`.

[12] C. McBride and J. McKinna. Functional pearl: I am not a number – I am a free variable. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–9, 2004.

[13] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[14] A. Stoughton. Substitution revisited. *Theoretical Computer Science*, 59(3):317–325, 1988.

[15] A. Stump. POPLmark 1a with named bound variables. `http://www.cs.uiowa.edu/~astump/poplmark-coq/`, 2005.

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

[17] J. Vouillon. POPLmark challenge, parts 1 and 2, 2005. Coq code available at `http://www.cis.upenn.edu/~plclub/wiki-static/vouillon-coq.tar.gz`.