# A Generic Approach to Proofs about Substitution

Abhishek Anand
Cornell University
Ithaca, New York, USA
aa755@cs.cornell.edu

Vincent Rahli
Cornell University
Ithaca, New York, USA
rahli@cs.cornell.edu

## ABSTRACT

It is well known that reasoning about substitution is a huge "distraction" that inevitably gets in the way of formalizing interesting properties of languages with variable bindings. Most formalizations have their own separate definitions of terms and substitution, and properties about it. However there is a great deal of uniformity in the way substitution works and the reasons why its properties hold. We expose this uniformity by defining terms, substitution and $\alpha$-equality generically in Coq by parametrizing them over a Context Free Grammar annotated with Variable binding information (CFGV).

We also provide proofs of many properties about the above definitions (enough to formalize the PER semantics of Nuprl in Coq). Unlike many other tools which generate a custom definition of substitution for each input, all instantiations of our term model share the *same* substitution function. The proofs about this function have been accepted by Coq's typechecker once and for all.

## Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*substitution, alpha equality, variable bindings, context free grammars*

## General Terms

Languages, Verification

## 1. INTRODUCTION AND RELATED WORK

Variable bindings are common in almost all high-level "mathematical" languages, e.g. $\lambda$-calculus, first order logic, Riemann integration. All these languages have a rather subtle notion of capture-avoiding substitution. Some of us may remember how we miscomputed definite integration problems in our high school calculus class because a variable got captured by substitution. However, such mistakes are not limited to high school students. Mistakes in symbolic reasoning about substitution have been discovered even in carefully-written and peer-reviewed articles [15, footnote 8].

The advent of proof assistants has made it possible to safely reason about complicated systems involving variable bindings and substitution [4, 8, 17]. However, this safety comes at an exorbitant price. It is well known that while formally proving interesting properties about these languages in proof assistants, a large fraction of the time is spent in reasoning about properties of substitution (and $\alpha$-equality) [3, Sec. 6]. Over the last few decades, a lot of research has gone into remedying this situation.

Most mathematics textbooks use the nominal representation where all variables have (mnemonically helpful) names and there could be multiple representatives of a class of $\alpha$-equal terms. One popular line of research is to develop alternate representations of terms. For example, the de Bruijn indices (DB) representation has the advantage that there is a unique term denoting a class of $\alpha$-equal terms. However it introduces the need to reason about index-lifting operations. The Locally Nameless (LN) representation [6] can be thought of as a hybrid between the DB representation and the nominal representation. Another popular but radically different representation is Higher Order Abstract Syntax (HOAS) [19, 20], where the binders of an eligible meta-language are used to encode the binding structure of the object language. Bound terms are represented as functions in the meta-theory.

While these advanced representations might be able to ease some of the burden, it might be difficult for some people to convince themselves that their definitions adequately represent what they have in mind. Also, it is often desirable that the reasoning steps required in a proof assistant match as closely as possible those in pen-and-paper proofs. We cater to these use-cases by choosing the nominal style of representation. Languages are usually specified by a context free grammar (CFG). Hence our term definition is generically parametrized by a CFG, and proofs by induction on our generic term structure closely resemble proofs by induction on the derivations in the corresponding grammar.

Irrespective of the representation one chooses, there are already tools to ease some of the tedium of formal reasoning about variable bindings and substitution. For example, the tools DBGen [22], LNgen [7] and Nominal Isabelle [25] take a specification of a language and automatically generate a substitution function. They also attempt to generate proofs of many of its properties. In the case of DBGen and LNGen, these Coq proofs are not guaranteed to be accepted by Coq's proof checker because the tactics used in the proof scripts


*LFMTP '14,* July 17 2014, Vienna, Austria

ACM 978-1-4503-2817-3/14/07 ...$15.00.
http://dx.doi.org/10.1145/2631172.2631177.

have not been formally verified.

One common problem with all these tools is that they do not expose the uniformity of the substitution function across different languages with variable bindings. They generate a different substitution function for each language and their users do *not* have any opportunity to generalize their proofs about substitution to all/most languages with variable bindings. However, modulo some syntactic differences, the definition of substitution and the reasons its properties hold are almost identical.

The key contributions of this paper are the following formal definitions in Coq: (1) Sec. 2 presents CFGV, a Record[1] type which defines the signature of an extension of CFGs that can be used to specify languages with Variable bindings. CFGV is heavily inspired by Ott [23] and Unbound [26]. The overhead of specifying a language as a member of the CFGV type should be comparable to that of specifying a language in Ott[2]. (2) Sec. 3 presents an inductively defined Term type-family that is parametrized by a CFGV and a symbol of it. Inuitively, for a CFGV $G$ and a symbol $s$ of $G$, Term $G$ $s$ represents terms that can be derived using the production rules in $G$, such that the root of the derivation tree is formed by a production rule with $s$ as its LHS. (3) Sec. 4 presents and inductive definition of $\alpha$-equality (tAlphaEq) and *simultaneous* substitution (tSSubst ) on the above Term type family. tAlphaEq is defined independently of tSSubst. It is instead based on a more primitive variable swapping operation and meets the specifications of Nominal Logic [21]. (4) Sec. 5 describes many properties of the above definitions. In particular, this set of properties includes nearly all the ones purely about substitution and $\alpha$-equality that we needed while formalizing the semantics of Nuprl (a cousin of Coq) in Coq [4].

None of the proofs mentioned above use any impredictive reasoning or any axiom. This means that our work can be easily translated to other proof assistants with dependent types, e.g., Agda [2], HoTT-Coq [24], and Nuprl [11][3]. The key benefit of our generic programming approach is that there is a single definition of the substitution function that works for all the instantiations and its properties have been proved once and forever. Unlike DBGen and LNgen, it is formally guaranteed that our users will not have to modify these Coq proofs. Also, our users get an opportunity to state their properties about substitution and $\alpha$-equality (if we missed some) in a general way such that it applies to many other languages.

A corollary of the above mentioned advantage is that users can extend their formal definitions of their language without revisiting the definitions and proofs that are purely about substitution and $\alpha$-equality. This work essentially generalizes (and is influenced by) the generic way the term language and substitution are implemented in the Nuprl proof assistant. This generic term language is described in [14, Sec. 2]. Although Nuprl's language has evolved a lot over the last few decades, no change was required to its substitution function. Our appreciation of this genericity snowballed during our recent experience [4] where we realized that that the effort required in proving properties about substitution and $\alpha$-equality is orders of magnitude larger than the effort required in merely defining them. The latter took less than 100 lines of code, while the former runs into more than 10000 lines of code [5].

The idea of using generic programming to implement substitution is also used in the Unbound tool [26], but unlike our work Unbound does not generate the logical infrastructure required to formally reason about properties of substitution. For the DB representation, there have been attempts to formalize parts of a generic metatheory [9, 12, 18], but these are not as general as ours. Also, an interesting difference in [12] is that the binding structure is specified indirectly by writing a "traverse" function.

```
Record CFGV := {
  Terminal : Type; VarSym : Type;
  TNonTerminal : Type; PNonTerminal : Type;

  PatProd : Type; EmbedProd : Type; TermProd : Type;

  varSem : VarSym → VarType;
  vSubstType : VarSym → TNonTerminal;
  tSemType : Terminal → Type;

  ppLhsRhs: PatProd →
    (PNonTerminal × list (PNonTerminal
                          + (Terminal + VarSym)));

  epLhsRhs : EmbedProd → (PNonTerminal × TNonTerminal);

  tpLhsRhs: TermProd →
    (TNonTerminal × list ((PNonTerminal + VarSym)
                          +(Terminal+TNonTerminal)));

  bindingInfo : TermProd → list (nat × nat);
  bindingInfoCorrect: ∀ (p: TermProd),
      ValidIndices (bindingInfo p) (snd (tpLhsRhs p));
}.
```

**Figure 1: Definition of CFGV**

## 2. DEFINITION OF CFGV

Formal languages are usually specified as a CFG. In languages with variable bindings, some annotations are required to express the concept of variable bindings. Our CFGV type (inspired by Ott) formalizes the concept of such grammars with variable-binding annotations. It is defined as a dependent Record in Coq as shown in Fig. 1. Fig. 2 defines a $\lambda$-calculus with a `letrec` construct as an instance of a CFGV. We use this as a running example for all our definitions. CFGV's first four members (Fig. 1) represent the classes of symbols of the grammar. While a regular CFG only has terminals and non-terminals, a CFGV has two more classes of symbols: VarSym and PNonTerminal.

Members of VarSym denote variable symbols. Variables essentially behave like terminals in a regular CFG. However, since they have a special role in defining $\alpha$-equality and substitutions, we chose to treat them separately. Each member of VarSym denotes a class of variables. For example, in the

[1]Coq code is presented in standard syntactic colors: blue is used for inductive types, dark red for constructors of inductive types, green for defined constants, functions and lemmas, red for keywords and some notations, and purple for variables. Some of the colored items are hyperlinked to the place they are defined, either in this document, in the standard library, or in our publicly available code.

[2]Because our target user is a person wishing to formalize the semantics of his/her language in Coq, we assume that he/she is well-versed in Coq

[3]The core of Coq without its impredicative universe Prop can be considered a sub language of these languages.

```
Inductive PNonTerminal : Set := lasgn | asgn | asgnRhs.
Inductive VarSym : Set := vsym.
Inductive TNonTerminal : Set := term.
Inductive Terminal :Set := .
Definition vSubstType (_ : VarSym) : TNonTerminal := term.

Inductive PatProd : Set := aNil | aCons | asgnP.
Inductive TermProd : Set := app | lam | letr.
Inductive EmbedProd : Set := asgnRhsE.
Definition epLhsRhs (_: EmbedProd):
   (PNonTerminal × TNonTerminal) := (asgnRhs, term).

Definition ppLhsRhs (p:PatProd) :
 (PNonTerminal × list (PNonTerminal+ (Terminal + VarSym))):=
match p with
 | aNil ⇒ (lasgn,[])
 | aCons ⇒ (lasgn, [inl lasgn, inl asgn])
 | asgnP ⇒ (asgn, [inrr vsym, inl asgnRhs])
end.
Definition tpLhsRhs (p:TermProd) :
 (TNonTerminal × list ((PNonTerminal + VarSym)
                         + (Terminal + TNonTerminal))):=
match p with
 | app ⇒ (term,[inrr term, inrr term])
 | lam ⇒ (term, [inlr vsym, inrr term])
 | letr ⇒ (term, [inll lasgn, inrr term])
end.

Definition bindingInfo (p:TermProd) : list (nat × nat) :=
match p with
 | app ⇒ [] | lam ⇒ [(0,1)] | letr ⇒ [(0,0),(0,1)]
end.
```

**Figure 2: An example of a CFGV specification**

example in Fig. 2 (and in the untyped $\lambda$-calculus), there is only one class of variables, which is denoted by the symbol vsym. While defining a language like Girard's System F [13], which has distinct term and type variables, the definition VarSym in Fig. 2 would look like:

```
Inductive VarSym : Set := tmVSym | tyVSym.
```

For a member *vs* of VarSym, varSem *vs* is a member of VarType, which specifies the Coq type that is used to implement variables denoted by the variable symbol *vs*.

```
Record VarType := {
  typ : Type; eqdec : Deq typ;
  fresh : ∀ (avoid : list typ) (sugg : list typ),typ;
  freshCorrect: ∀ (avoid : list typ)
    (sugg : list typ), !(LIn (fresh avoid sugg) avoid)
}.
```

A member of VarType contains a Coq type (typ) and some additional information required to implement variables as members of typ. The member eqdec is a function that decides equality in the type typ. The function fresh takes a list *avoid* of variables, and list *sugg* of suggestions and is guaranteed to return a variable that does not occur in *avoid*. It is expected, but not required to pick a name close to the suggestions. We use Coq's nat and string types to implement two instances members of VarType. The fresh function of the latter implementation appends a large enough number to the first suggestion so that it is disjoint from any member of *avoid*. We will use the notation vType *vs* to stand for typ (varSem *vs*).

Like Unbound, we will have two sorts of syntactic entities: *patterns* and *terms*. Variables in patterns are binding occurrences, whereas the ones in terms are references to binding sites. For example, in $\lambda x.z$, $x$ is a pattern, while $z$ is a term. Modern languages have a variety of binding constructs which go beyond single variables. The match construct of Coq is one such example. Consider the simple but contrived btsize function below:

```
Inductive btree: Set :=
  leaf : btree | bnode : btree → btree → btree.
Fixpoint btsize (bt: btree) :nat :=
match bt with
  | leaf ⇒ 1
  | bnode t leaf ⇒ (btsize t)+1
  | bnode t (bnode ta tb) ⇒ (btsize t)+(btsize ta)+(btsize tb)
end.
```

There are three clauses in the pattern matching example above (separated by the | symbol). In each clause, the part on the left of the ⇒ symbol can be considered as a pattern, while the parts on the right can be considered as terms. This example also illustrates that the structure of patterns could be arbitrary finite trees.

We have two kinds of non-terminals in the usual sense of CFGs. Members of PNonTerminal are symbols that denote (composite) patterns, while members of TNonTerminal are symbols that denote (composite) terms of the language. In the example in Fig. 2, there are three kinds of composite patterns which model various parts of the `letrec` construct. These are denoted by three members of PNonTerminal. asgn denotes an assignment, lasgn denotes a list of assignments, and asgnRhs denotes the right-hand-side of an assignment. The purpose of the last one is explained below. There is only one kind of terms, as denoted by the only member term of TNonTerminal. In the case of System F, TNonTerminal would be defined as follows:

```
Inductive TNonTerminal : Set := term | type.
```

For every variable symbol in VarSym, the function vSubstType specifies a TNonTerminal that denotes the terms that can be substituted for those variables. In the example in Fig. 2, vSubstType is the trivial constant function because both its domain and codomain are singletons. However, while modeling System F, it would look like:

```
Definition vSubstType (vc : VarSym) : TNonTerminal :=
match vc with
  tmVSym ⇒ term | tyVSym ⇒ type
end.
```

The vSubstType function implicitly specifies production rules for each element *vc* of VarSym where the left-hand-side is vSubstType *vc* and the right-hand-side is *vc*. As explained below (and in [23, Sec. 3.2]), having these rules is one simple way to make sure that the result of substitution is always well-typed (derivable in the grammar).

There are three other classes of production rules in a CFGV: PatProd, EmbedProd, TermProd. For each of these, there is a function with the suffix "LhsRhs" that gives the symbol(s) at the left and right-hand-sides of the production. Members of PatProd are meant to be used to construct patterns from an ordered collection of patterns, terminals or variables. Members of TermProd are meant to be used to construct terms from an ordered collection of patterns, variables, terminals or non-terminals. Members of EmbedProd denote embeddings of terms into patterns. This is a concept inspired by Unbound. Both Ott and Unbound highlight the need to have patterns which contain terms whose variables are not supposed to be binding occurrences. A classic example is the construct `letrec (x=a,y=b) in c` which is common in many languages. A natural representation of this term will have the part `(x=a,y=b)` represented as a single pattern. While `x` and `y` bind in `c`, `a` and `b`, the variables in `a` and `b` are not supposed to bind in `c`. In a CFGV representation, `a` and `b` will be modeled as embeddings in the pattern `(x=a,y=b)`. In the example in Fig. 2, the PNonTerminal asgnRhs mentioned above represents the result of an embedding. Note that inlr is a notation for (fun $x$ ⇒ inl (inr $x$)). inll and inrr should be understood similarly. The EmbedProd asgnRhsE builds a PNonTerminal asgnRhs from a term. Note that in Fig. 2, there is no TermProd to construct a term from a vsym. As mentioned above, that production is already implicitly specified by vSubstType.

Ott has a more general (and complicated) mechanism where they let users write arbitrary structurally recursive functions that return the variables of a pattern that denote binding occurrences.

Members of TermProd have a distinct property that they also specify variable binding information. For brevity, let the phrase binding variables stand for "variables that denote binding occurrences". As mentioned above, a variable in a pattern is a binding variable iff it is not inside an embedding. The function bindingInfo returns a list of pairs of numbers, where a pair $(i, j)$ specifies that the binding variables in the pattern denoted by the $i^{th}$ symbol of the right-hand-side bind in the term or pattern denoted by the $j^{th}$ symbol. We call the $i^{th}$ element a source of binding variables, and the $j^{th}$ element a binding site. If the $j^{th}$ symbol denotes a pattern, only the occurrences inside embeddings act as binding sites. Note that $i$ and $j$ need not be distinct. This is in fact necessary to define recursive bindings like those in the `letrec` construct mentioned above (see the letr case in the definition of bindingInfo in Fig. 2). Hence, we do not need the `Rec` construct of Unbound.

The member bindingInfoCorrect ensures that the output of bindingInfo is correct; i.e., the indices are within bounds, there are no duplicate pairs and each index that corresponds to either a PNonTerminal or VarSym symbol is mentioned as first element of some pair. Conversely, the first element of each pair must correspond to either a PNonTerminal or VarSym symbol. We provide tactics to ease the process of writing the usually straightforward proofs of bindingInfoCorrect.

Note that the members of PatProd do *not* specify variable binding information. There are no internal bindings inside patterns (except those inside embedded terms). This is analogous to the way there is no `Bind` clause for constructing patterns in [26, Fig. 2]

There are other members in a CFGV record that are not shown in the figure. It includes proofs of decidability of equality in all the symbol and production types, and the types returned by tSemType. These are required for the decidability of equality and $\alpha$-equality[4]. Please refer to our user manual [1] for completely precise definitions of everything mentioned in this paper. Also, just clicking items in colored Coq code might take the reader to the right place.

## 3. TERMS AS DERIVATIONS IN CFGV

As mentioned above, unlike other tools such as Ott and Nominal Isabelle, which generate (using unverified algorithms) customized definitions of syntactic terms (and patterns) for each input specification, we exploit the power of dependent types to have only one definition of each concept. Our definition of syntax of a language is parametrized by an arbitrary CFGV. As shown in Fig. 3, syntactic objects are represented as members of three mutually defined inductive types. These can be thought of as derivations trees of the underlying CFG.

We first define some notation to simplify the remaining presentation. We define the following type to more readably represent members of disjoint unions of grammar symbols (as in the codomain type of functions like tpLhsRhs)

```
Inductive GSym (G: CFGV) : Type :=
| gsymT : Terminal G → GSym G
| gsymV : VarSym G → GSym G
| gsymTN : TNonTerminal G → GSym G
| gsymPN : PNonTerminal G → GSym G.
```

The functions with suffix "Lhs" and "Rhs" are appropriate projections of the corresponding ones with suffix "LhsRhs" in Fig. 1. Finally, the function ppRhsSym converts the disjoints unions in the output of ppRhs to the corresponding members of the GSym type above.

The `Inductive` type Term is additionally parametrized by a symbol of the grammar $G$. For a member $s$ of GSym $G$, Term $G$ $s$ represents terms that can be derived using the production rules in $G$, such that the root of the derivation tree is formed by a production rule with $s$ as its LHS. The vleaf constructor takes a variable symbol and a variable in the corresponding semantic type. The vleaf clause directly constructs a Term of the type of the TNonTerminal that denotes the class of terms that can be substituted for the variable. Note that this can be thought of as using the implicit production rules (associated with variable symbols) mentioned in the previous section. Also note that there is no way to construct a Term corresponding to a PNonTerminal symbol. Patterns are instead elements of the Pattern family. This distinction (inspired by Unbound) is useful as many

```
Section Gram. Context {G : CFGV}.
Inductive Term: (GSym G) → Type :=
| tleaf : ∀ (T : Terminal G),
    (tSemType G T) → Term (gsymT T)
| vleaf : ∀ (vc : VarSym G),
    (vType vc) → Term (gsymTN (vSubstType G vc))
| tnode : ∀ (p: TermProd G),
    Mixture (tpRhsAugIsPat p) → (Term (gsymTN (tpLhs G p)))

with Pattern : (GSym G) → Type :=
| ptleaf : ∀ (T : Terminal G),
    (tSemType G T) → Pattern (gsymT T)
| pvleaf : ∀ (vc : VarSym G),
    (vType vc) → Pattern (gsymV vc)
| embed : ∀ (p: EmbedProd G),
    Term (gsymTN (epRhs G p))
      → Pattern (gsymPN (epLhs G p))
| pnode : ∀ (p: PatProd G),
    Mixture (map (fun x ⇒ (true,x)) (ppRhsSym p))
      → (Pattern (gsymPN (ppLhs G p)))

with Mixture : (list (bool × (GSym G))) → Type :=
| mnil : Mixture []
| mtcons : ∀ {h: (GSym G)} {tl: list (bool × (GSym G))},
    Term h → Mixture tl → Mixture ((false,h)::tl)
| mpcons : ∀ {h: (GSym G)} {tl: list (bool × (GSym G))},
    Pattern h → Mixture tl → Mixture ((true,h)::tl).
```

**Figure 3: Definition of Terms**

[4]More subtly, these are required to avoid using axioms like JMeq_eq [10, Sec. 10.4] while eliminating the dependent datatypes that are parametrized by symbols of a CFGV.

operations like substitution differ significantly for terms and patterns. Note that as intended, there is no clause to construct a Pattern corresponding to a TNonTerminal symbol. We describe the tnode and the pnode clauses below.

An element of Mixture is supposed to represent an ordered collection of entities corresponding to the right-hand-side of a production rule. One would naturally expect a Mixture to be parametrized by a list of grammar symbols. However, trying to define it that way revealed a potential source of ambiguity. Entities corresponding to terminals could be parts of both a term and a pattern. For example, suppose we wanted to formalize the syntax of Coq. Constructors of Inductive types would be best represented as terminals. However, we know that they can appear both in patterns (of a match case) and bodies (terms). So, each member of the list is also augmented with a boolean, that should be true iff the corresponding member is to be a pattern node.

The pnode constructor builds a pattern node from a Mixture that corresponds to a PatProd's right-hand-side. Note that the subterms of a pattern node can only be other pattern nodes (i.e., elements of the Pattern family). Hence, the parameter of its Mixture is obtained by applying (map (fun $x$ ⇒ (true,$x$)) to the list of symbols in the right-hand-side of the PatProd $p$. This operation augments each element in the right-hand-side of $p$ with the boolean value true.

The tnode constructor of Term is a bit more complicated. It constructs a term from a Mixture that corresponds to a TermProd's right-hand-side. Again, the tpRhsAugIsPat function augments each symbol in the the right-hand-side of the TermProd $p$ with booleans indicating whether a pattern node is required. A pattern node is required iff the symbol is a PNonTerminal or a VarSym. As mentioned above, variables that correspond to terms (binding sites) are only formed by implicit production rules embodied in the vleaf clause.

While members of Term might look bloated they exactly resemble the corresponding derivation in the underlying CFG. As a concrete example, the term `letrec (x:=y) in z` can be represented as a member of Term parametrized by the CFGV example in Fig. 2 and its TNonTerminal symbol term as below. Siblings in a mixture (except mnil) are vertically aligned. Please excuse the lack of syntax coloring.

```
node letr
  (mpcons
     (pnode aCons
        (mpcons (pnode aNil mnil)
        (mpcons
           (pnode asgnP
               (mpcons (pvleaf vsym x)
               (mpcons (embed asgnRhsE
                  (vleaf vsym y)) mnil))) mnil)))
  (mtcons (vleaf vsym z) mnil))
```

We use the Scheme mechanism of Coq to automatically generate mutual induction principles for these three mutually inductive definitions (see [1] for details). These induction principles turned out to be strong enough for us to conveniently prove the properties mentioned below.

## 4. VARIABLE BINDING SEMANTICS

The definition of terms in the previous section totally disregarded the binding information. Now, we define concepts that formalize the variable-binding semantics of a CFGV.

Consider a term formed by a TermProd $p$. Such a term is of the form tnode $p$ $mix$, where $mix$ is a member of the type Mixture (tpRhsAugIsPat $p$). Recall that bindingInfo $p$ is a list (nat × nat), whose member $(i, j)$ denotes that the binding variables in the $i^{th}$ member of $mix$ bind in the $j^{th}$ member $mix$. The function pBndngVars below formalizes the concept of binding variables of a pattern as described in Sec. 2. The function getBVarsNth then uses it to define what we mean by binding variables of the $i^{th}$ member. The transport function is only required for well-typedness. For any $a$, transport _ $a$ can be understood as simply $a$. For brevity, we omit types of some arguments which were automatically inferred by Coq. Also arguments in curly braces are implicit.

```
Fixpoint pBndngVars {G} (vc : VarSym G)
   {gs} (p : Pattern gs) : (list (vType vc)) :=
match p with
| ptleaf _ _ ⇒ [] | embed _ _ ⇒ []
| pvleaf vcc var ⇒ match (DeqVarSym vcc vc) with
                     | left eqq ⇒ [transport eqq var]
                     | right _ ⇒ []
                     end
| pnode _ mix ⇒ mBndngVars vc mix
end
with mBndngVars {G} (vc: VarSym G) {lgs}
  (pts : Mixture lgs) : (list (vType vc)) :=
match pts with
| mnil ⇒ [] | mtcons _ _ _ tl ⇒ mBndngVars vc tl
| mpcons _ _ h tl ⇒ (pBndngVars vc h) ++ mBndngVars vc tl
end.
```

```
Fixpoint getBVarsNth {G} (vc : VarSym G) {lgs}
  (pts : Mixture lgs) (n : nat) : (list (vType vc)) :=
match (pts,n) with
| (mnil,_) ⇒ [] | (mtcons _ _ ph pth, 0) ⇒ []
| (mpcons _ _ ph pth, 0) ⇒ (pBndngVars vc ph)
| (mtcons _ _ ph pth, S m) ⇒ (getBVarsNth vc pth m)
| (mpcons _ _ ph pth, S m) ⇒ (getBVarsNth vc pth m)
end.
```

To define concepts like $\alpha$-equality, free variables, or substitution, we need to compute the collection of all variables that bind in a member of a Mixture. For example, while computing free variables, we need to subtract this collection from the free variables of that member. Our allBndngVars function achieves this. We first define a function bndngPatIndices such that for a TermProd $p$, (bndngPatIndices $p$) denotes a list (list nat) whose $j^{th}$ element is the list of all numbers $i$ such that the pair $(i, j)$ occurs in bindingInfo $p$. Intuitively, for a Mixture $mix$ corresponding to the right-hand-side of $p$, the $j^{th}$ element of (bndngPatIndices $p$) is the list of indices of all the patterns of $mix$ whose variables bind in the $j^{th}$ element of $mix$. For example, in the example in Fig. 2, (bndngPatIndices letr) = [[0], [0]]. The definition of allBndngVars (mentioned above) is now a one liner:

```
Definition allBndngVars {G} (vc : VarSym G)
    (p: TermProd G) (mix: Mixture (tpRhsAugIsPat p))
    : list ( list (vType vc)) :=
map (flat_map (getBVarsNth vc mix)) (bndngPatIndices p).
```

For the Mixture $mix$ mentioned above, allBndngVars $mix$ $p$ is a list whose $j^{th}$ element is a list of all the variables that bind in the $j^{th}$ element. For example, the value of allBndngVars for the top level mixture in the representation of `letrec x:=y in z` at the end of Sec. 3 is: [[x],[x]]. Now, we can define the collection of free variables of terms in a straightforward way below. On lists of lists, the function lhead returns the head of its input (or an empty list iff the input was empty).

```
Fixpoint tfreevars {G}(vc : VarSym G) {gs}
    (p : Term gs) : list (vType vc) :=
match p with
| tleaf _ _ ⇒ []
| vleaf vcc var ⇒ match (DeqVarSym vcc vc) with
                    | left eqq ⇒ [transport eqq var]
                    | right _ ⇒ []
                    end
| tnode p m ⇒ mfreevars vc m (allBndngVars vc p m)
end
with pfreevars {G} (vc : VarSym G) {gs}
    (p : Pattern gs) : list (vType vc) :=
match p with
| ptleaf _ _ ⇒ [] | pvleaf vcc var ⇒ []
| pnode p lpt ⇒ mfreevars vc lpt []
| embed vcc nt ⇒ tfreevars vc nt
end
with mfreevars {G} (vc: VarSym G) {lgs} (m: Mixture lgs)
  (llB : list (list (vType vc))) : list (vType vc) :=
match m with
| mnil ⇒ []
| mtcons _ _ h tl ⇒ (tfreevars vc h - lhead llB)
                        ++ mfreevars vc tl (tail llB)
| mpcons _ _ h tl ⇒ (pfreevars vc h - lhead llB)
                        ++ mfreevars vc tl (tail llB)
end.
```

The concept of *simultaneous* substitution is useful to define the operational semantics of languages with more complicated patterns than singleton variables. Hence, instead of defining a substitution as a pair of a variable and a term, we define it as a list of such pairs:

```
Definition SSubstitution {G} (vc : VarSym G):=
  list (vType vc × Term (gsymTN (vSubstType G vc))).
```

The substitution operation is defined as follows:

```
Definition tSSubst {G} {vc: VarSym G } {gs}
    (t : Term gs) (sub : SSubstitution vc) : Term gs :=
  let tr := tRenAlpha t (rangeFreeVars sub)
  in tSSubstAux tr sub.
```

It uses the function tSSubstAux which is simultaneously defined with pSSubstAux and mSSubstAux by mutual structural recursion, similar to the way the free variable functions are defined above. Intuitively, these three functions assume that *all* bound variables[5] of the term are already disjoint from the free variables in the range of the SSubstitution. In other words, they don't worry about $\alpha$-renaming the bound variables. As in tfreevars, in the vleaf case, tSSubstAux first checks whether the variable is of same class as that of the substitution. If so, it finds the *first* pair in the SSubstitution whose first component is the same variable. If there is one, it returns the second component of the pair. In all the remaining subcases of the vleaf case, the output is same as the input. Similar to mfreevars, in the mtcons and mpcons cases, mSSubstAux filters the SSubstitution to remove any pair whose first component is in lhead *llB* and then call tSSubstAux or pSSubstSux respectively.

The functions tRenAlpha, pRenAlpha and mRenAlpha are defined by mutual structural recursion. These functions $\alpha$-rename their input (a Term, a Pattern and a Mixture respectively) so that *all* their bound variables are disjoint from a given list of variables (and the result is $\alpha$-equal). We have proved both these properties. For better usability, we have also proved that these functions return the same term as the input if it already met the disjointness condition. Note that these $\alpha$-renaming functions use the fresh function that comes with a VarType.

## 4.1 Alpha Equality

One could use substitution to define $\alpha$-equality. However, nominal logic [21] shows that it can be defined in terms of a much more simple primitive operation called swapping of variables. For variables $v$ $a$ $b$ of some class, oneSwapVar $v$ $(a,b)$ can be informally defined as: `if v=a then b else if v=b then a else v`. Note that the definition of VarType provides the required decider of equality. This operation can be lifted to a list of pairs as follows:

```
Definition Swapping {G} (vc : VarSym G)
:= list (vType vc × vType vc).
Definition swapVar {G} {vc : VarSym G}
  (sw : Swapping vc) (v : vType vc) : vType vc
:= fold_left oneSwapVar sw v .
```

Note that for any Swapping *sw*, swapVar *sw* is a bijective endofunction in the type vType *vc*. The inverse function is swapVar (rev *sw*). Hence, the swapping operation enjoys much nicer logical properties than the operation of substituting variables for other variables. In particular, almost all operations and relations we usually care about are equivariant [21]. $\alpha$-equality can be defined purely in terms of equivariant functions and relations. The mutually defined tSwap, pSwap and mSwap functions recurse through their input and apply the swapVar *sw* operation on *a*ll variables (of the class of the swapping). The remaining definitions in this section are in the context of (are implicitly parametrized by) a CFGV $G$ and a $vc$ of type VarSym.

Irrespective of the complications of representations of patterns, the core of the definition of $\alpha$-equality is about when two abstractions are $\alpha$-equal. As in [25], we define an abstraction as a pair of a list of variables and a term (or a pattern):

```
Notation vcType := (vType vc).
Inductive Abstraction : Type :=
| termAbs: ∀ {gs : GSym G},
    list vcType → Term gs → Abstraction
| patAbs: ∀ {gs : GSym G},
    list vcType → Pattern gs → Abstraction.
```

The meaning of an abstraction is that the given list of variables bind in the Term or Pattern. When defining $\alpha$-equality for a Mixture, we need to make a list of such abstractions:

```
Fixpoint MakeAbstractions {lgs} (m : Mixture lgs)
 (llB : list (list (vType vc))): list Abstraction :=
match m with
| mnil ⇒ []
| mtcons _ _ h tl⇒ (termAbs (lhead llB) h
                     ::(MakeAbstractions tl (tail llB)))
| mpcons _ _ h tl⇒ (patAbs (lhead llB) h
                     ::(MakeAbstractions tl (tail llB)))
end.
```

When we make abstractions from a Mixture at a tnode, the argument *llB* in the function above is intended to be computed by the allBndngVars function mentioned above. Finally, we can define $\alpha$-equality w.r.t. the class of variables denoted by $vc$ (a member of VarSym $G$) as shown in Fig. 4. If there are multiple members of VarSym $G$, "$\alpha$-equal" terms must be $\alpha$-equal w.r.t. all the members of VarSym $G$. Fig. 4 has four mutually inductively defined components. tAlphaEq

[5] This concept of *all* bound variables is formalized by pBndngVarsDeep. Recall that to see the definition of a colored Coq object, you can just click it.

is the main definition that defines $\alpha$-equality on terms. pAlphaEq defines $\alpha$-equality on patterns. Intuitively, two patterns are $\alpha$-equal iff their corresponding embeddings are $\alpha$-equal (tAlphaEq) and the remaining parts only differ in variable names. Note that the alpvar clause allows two variable patterns with different variables to be $\alpha$-equal. AlphaEqAbs defines $\alpha$-equality on abstractions and is the central concept in these definitions. lAlphaEqAbs lifts AlphaEqAbs to a pair of lists of abstractions in a straightforward way. Both tFresh *lbnew* [*tma*,*tmb*] and pFresh *lbnew* [*tma*,*tmb*] assert that *lbnew* is disjoint from *a*ll the variables of *tma* and *tmb* and that *lbnew* has no repeated elements in it. Two terminals or two variables are tAlphaEq iff they are equal (see the alt and alv clauses). The interesting case (alnode) is the one about two tnodes. In this case, they have to be formed by the same TermProd and the lists of abstractions formed from their Mixture's have to be $\alpha$-equal. Intuitively, the alAbT and alAbP clauses say that two abstractions $(lva, ta)$ and $(lvb, tb)$ are $\alpha$-equal if the lengths of $lva$ and $lvb$ are equal and we can come up with fresh distinct variables $lvnew$ of the same length such that the result of swapping $ta$ and $tb$ with `zip lva lvnew` and `zip lvb lvnew` respectively are $\alpha$-equal.

Again, we use the `Scheme` mechanism of Coq to extract a mutual induction principle for the mutually inductive definitions of $\alpha$-equality. Statements with a hypothesis about $\alpha$-equality were much easier to prove using these induction principles, rather than using induction on the structure of terms. In our experience, the proofs about this definition of $\alpha$-equality which is defined in terms of swapping operations turned out to be much easier to reason about than the one where $\alpha$-equality is defined using substitution operations as in our previous work [4].

## 5. PROVED PROPERTIES

A user of our framework has to look at the definitions above and make sure that they match the corresponding concepts they have in mind. If they match, we provide a wealth of proofs about the above definitions. These proofs use no extra axioms, have been accepted once and for all by Coq's typechecker and are ready to be used for any language that can be described (typechecked) as a member of the CFGV type. Contrast this with other tools like [25, 22, 7] which generate new proofs for each input language using unverified algorithms, and whose outputs (definitions, proofs) are not formally guaranteed to typecheck. For brevity, we semi-formally list the key properties that we have proved and did not already mention. The companion webpage[6] will guide the reader to the exact locations of the formal definitions and proofs of each of these.

*Nominal Logic.* Our swapping functions satisfy the properties in [21]. We generalize these properties to the case when a swapping is defined as a list of pairs, instead of a single pair. Almost all the functions and relations involving variables that we defined are equivariant (w.r.t. Coq's propositional equality eq). This includes the membership predicate on lists of variables, disjointness of lists, $\alpha$-equality, functions computing free variables, binding variables, the substitution functions, the swapping functions, e.t.c. A notable exception is the $\alpha$-renaming function (tRenAlpha).

[6] `http://www.nuprl.org/html/CFGVLFMTP2014/LFMTPProofs.html`

```
Notation zip := (combine).
Inductive tAlphaEq: ∀ {gs : (GSym G)},
      (Term gs) → (Term gs) → Type :=
| alt: ∀ T t, tAlphaEq (tleaf T t) (tleaf T t)
| alv: ∀ V v, tAlphaEq (vleaf V v) (vleaf V v)
| alnode: ∀ (p: TermProd G)
    (ma mb : Mixture (tpRhsAugIsPat p)),
    lAlphaEqAbs (MakeAbstractions ma (allBndngVars vc p ma))
                (MakeAbstractions mb (allBndngVars vc p mb))
    → tAlphaEq (tnode p ma) (tnode p mb)

with pAlphaEq:
    ∀ {gs : (GSym G)},
      (Pattern gs) → (Pattern gs) → Type :=
| alpt: ∀ T t, pAlphaEq (ptleaf T t) (ptleaf T t)
| alpvar: ∀ vcc vara varb, pAlphaEq (pvleaf vcc vara)
                                    (pvleaf vcc varb)
| alembed: ∀ p nta ntb,
    tAlphaEq nta ntb → pAlphaEq (embed p nta) (embed p ntb)
| alpnode:
    ∀ (p: PatProd G)
    (ma mb : Mixture (map (fun x ⇒ (true,x)) (ppRhsSym p))),
    lAlphaEqAbs (MakeAbstractions ma [])
                (MakeAbstractions mb [])
    → pAlphaEq (pnode p ma) (pnode p mb)

with AlphaEqAbs: Abstraction → Abstraction → Type :=
| alAbT : ∀ {gs : GSym G}
    (lbva lbvb lbnew : list vcType)
    (tma tmb : Term gs),
    let swapa := zip lbva lbnew in
    let swapb := zip lbvb lbnew in
    length lbva = length lbnew
    → length lbvb = length lbnew
    → tFresh lbnew [tma,tmb]
    → disjoint (lbva ++ lbvb) lbnew
    → tAlphaEq (tSwap tma swapa) (tSwap tmb swapb)
    → AlphaEqAbs (termAbs lbva tma) (termAbs lbvb tmb)
| alAbP : ∀ {gs : GSym G}
    (lbva lbvb lbnew : list vcType)
    (tma tmb : Pattern gs),
    let swapa := zip lbva lbnew in
    let swapb := zip lbvb lbnew in
    length lbva = length lbnew
    → length lbvb = length lbnew
    → pFresh lbnew [tma,tmb]
    → disjoint (lbva ++ lbvb) lbnew
    → pAlphaEq (pSwap tma swapa) (pSwap tmb swapb)
    → AlphaEqAbs (patAbs lbva tma) (patAbs lbvb tmb)

with lAlphaEqAbs :
  list Abstraction → list Abstraction → Type :=
| lAlAbsNil : lAlphaEqAbs [] []
| lAlAbsCons : ∀ (ha hb: Abstraction)
            (la lb: list Abstraction),
            AlphaEqAbs ha hb
            → lAlphaEqAbs la lb
            → lAlphaEqAbs (ha::la) (hb::lb).
```

**Figure 4: Definition of $\alpha$-equality**

*Alpha Equality.* $\alpha$-equality is an equivalence relation. $\alpha$-equal terms and patterns have the same free variables (equal as lists). Given a term, any swapping whose restriction to the free variables of the term is the identity function results in an $\alpha$-equal term. Also, the set of free variables (as computed by tfreevars) of a term is precisely its *support* [21] w.r.t. $\alpha$-equality; So one does not have to trust our definition of tfreevars. Finally, we provide a verified decision procedure for $\alpha$-equality.

*Substitution.* Suppose we are in the context of a $G$ of type CFGV, a $vc$ of type VarSym, and a $gs$ of type GSym $G$. In the statements below, $a$ and $b$ are arbitrary members of Term $gs$, and $s$, $sa$ and $sb$ are arbitrary members of SSubstitution $vc$, and $l$ is an arbitrary member of list (vType $vc$). fv is a notation for tfreevars $vc$. fvr returns the free variables of the range of a substitution. It is defined as flat_map (fun $p$ ⇒ fv (snd $p$)). Dom returns the domain of a SSubstitution. It is defined as map (fun $p$ ⇒ (fst $p$)). SubstSub $sa$ $sb$ applies the substitution $sb$ to every element in the range of $sa$. It is defined as map (fun $p$ ⇒ ( fst $p$, tSSubst (snd $p$) $sb$ )) $sa$. filter $s$ $l$ filters $s$ to remove all pairs whose first component is in the list $l$. keepFirst $s$ $l$ filters $s$ to only *keep* pairs whose first component is in the list $l$. In case that multiple pairs have the same first component, it only keeps the first one. $\approx$ is an (infix) binary relation on lists that asserts that they are equal as sets (i.e have same members). A blank line separates each property.

$a =_\alpha b \rightarrow sa =_\alpha sb \rightarrow$ tSSubst $a$ $sa$ $=_\alpha$ tSSubst $b$ $sb$.

disjoint $l$ (fv $a$) $\rightarrow$ tSSubst $a$ $s$ $=_\alpha$ tSSubst $a$ (filter $s$ $l$).

fv (tSSubst $a$ $s$) $\approx$ (fv $a$ - Dom $s$) ++ fvr (keepFirst $s$ (fv $a$)).

disjoint (fvr $sa$) (Dom $sb$)
$\rightarrow$ disjoint (fvr $sb$) (Dom $sa$)
$\rightarrow$ disjoint (Dom $sa$) (Dom $sb$)
$\rightarrow$ tSSubst (tSSubst $a$ $sa$) $sb$ $=_\alpha$ tSSubst (tSSubst $a$ $sb$) $sa$.

tSSubst (tSSubst $a$ $sa$) $sb$ $=_\alpha$ tSSubst $a$ ((SubstSub $sa$ $sb$)++$sb$).

## 6. FUTURE WORK

So far, we wrote our functions mainly with correctness in mind. We plan to write more efficient implementations of functions like substitution and prove that the new versions are equivalent (upto $\alpha$-equality). For example, one could implement substitution in a way that requires only one pass over the term.

Once the homotopy type theory [24] version of Coq is mature enough, we plan to use its higher inductive types to define a quotient type over our Term type where propositional equality coincides with $\alpha$-equality.

Also, we plan to reuse some existing work on verified parsers [16] to build one that is parametrized on a CFGV, and perhaps some additional information.

*Acknowledgements.*
We thank Prof. Robert Constable for helpful comments.

## References

[1] Abhishek Anand and Vincent Rahli. *CFGV Online Reference.* URL: http://nuprl.org/html/CFGVLFMTP2014/.

[2] *The Agda Wiki.* http://wiki.portal.chalmers.se/agda/pmwiki.php.

[3] T. Altenkirch. "A formalization of the strong normalization proof for System F in LEGO". In: *TLCA*. Springer, 1993, pp. 13–28.

[4] A. Anand and V. Rahli. "Towards a Formally Verified Proof Assistant". Accepted to ITP 2014. 2014.

[5] A. Anand and V. Rahli. *Towards a Formally Verified Proof Assistant.* Tech. rep. http://www.nuprl.org/html/Nuprl2Coq/. Cornell University, 2014.

[6] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. "Engineering Formal Metatheory". In: *POPL*. 2008, pp. 3–15.

[7] B. Aydemir and S. Weirich. "LNgen: Tool support for locally nameless representations". 2010.

[8] B. Barras. "Sets in Coq, Coq in Sets". In: *Journal of Fromalized Reasoning* 3.1 (2010), pp. 29–48.

[9] A. Chlipala. "A certified type-preserving compiler from lambda calculus to assembly language". In: *PLDI*. 2007.

[10] A. Chlipala. *Certified programming with dependent types.* MIT Press, 2011.

[11] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, Inc., 1986.

[12] François Pottier. *dblib.* Dec. 3, 2013. URL: http://gallium.inria.fr/blog/announcing-dblib/.

[13] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge University Press, 1989.

[14] D. J. Howe. "Equality in Lazy Computation Systems". In: *LICS*. 1989, pp. 198–203.

[15] S. C. Kleene. "Disjunction and Existence Under Implication in Elementary Intuitionistic Formalisms". In: *JSL* 27.1 (Mar. 1, 1962), pp. 11–18.

[16] A. Koprowski and H. Binsztok. "TRX: A formally verified parser interpreter". In: *LMCS* 7.2 (2011), pp. 345–365.

[17] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. "CakeML: a verified implementation of ML". In: POPL. ACM Press, 2014, pp. 179–191.

[18] G. Lee, B. C. Oliveira, S. Cho, and K. Yi. "GMETA: a generic formal metatheory framework for first-order representations". In: *Programming Languages and Systems.* Springer, 2012, pp. 436–455.

[19] F. Pfenning and C. Elliott. "Higher-Order Abstract Syntax". In: *PLDI*. 1988, pp. 199–208.

[20] B. Pientka and J. Dunfield. "Beluga: A framework for programming and reasoning with deductive systems (system description)". In: *Automated Reasoning*. Springer, 2010, pp. 15–21.

[21] A. M. Pitts. "Nominal logic: A first order theory of names and binding". In: *TACS*. Springer, 2001, pp. 219–242.

[22] E. Polonowski. "Automatically generated infrastructure for De Bruijn syntaxes". In: *ITP*. Springer, 2013, pp. 402–417.

[23] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. StrnišA. "Ott: Effective tool support for the working semanticist". In: *JFP* 20.01 (Jan. 2010), p. 71.

[24] I. The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* 2013.

[25] C. Urban and C. Kaliszyk. "General Bindings and Alpha-Equivalence in Nominal Isabelle". In: *LMCS* 8.2 (2012).

[26] S. Weirich, B. A. Yorgey, and T. Sheard. "Binders unbound". In: *ICFP*. 2011, pp. 333–345.