# Residual Theory in $\lambda$-calculus: A Formal Development

Gérard Huet

INRIA Rocquencourt[1]

B.P. 105 - 78153 Le Chesnay CEDEX

**Abstract.** We present the complete development, in Gallina, of the residual theory of $\beta$-reduction in pure $\lambda$-calculus. The main result is the Prism Theorem, and its corollary Lévy's Cube Lemma, a strong form of the parallel-moves lemma, itself a key step towards the confluence theorem and its usual corollaries (Church-Rosser, uniqueness of normal forms).

Gallina is the specification language of the Coq Proof Assistant[7, 11]. It is a specific concrete syntax for its abstract framework, the Calculus of Inductive Constructions[15]. It may be thought of as a smooth mixture of higher-order predicate calculus with recursive definitions, inductively defined data-types, and inductive predicate definitions reminiscent of logic programming.

The development presented here was fully checked in the current distribution version Coq V5.8. We just state the lemmas in the order in which they are proved, omitting the proof justifications. The full transcript is available as a standard library in the distribution of Coq.

## 1 Preliminaries

We shall not attempt to define formally the syntax of Gallina in this paper. First of all, we claim that this language is close enough to the standard mathematical notation that the meaning of definitions and statements ought to be obvious, once a few basic conventions are understood. For instance, the sentence `(n:nat)(gt n O)+<nat>n=O` corresponds to the universal quantification:

$$\forall n \in I\!N \quad n > O \vee n =_{I\!N} 0$$

It would be easy to "beautify" our mathematics, by replacing the actual Gallina code by corresponding TEX math-mode code. We have however refrained from such doctoring in this document, where the Gallina development is given verbatim. An introduction to Gallina may be found in another case study[11].

An important feature of Gallina is the definition of inductive structures and inductive predicates. To every inductive type is associated an elimination operator, corresponding to an induction principle for propositions, and a recursion principle for sets. Recursive definitions are currently handled in a primitive recursion style which is not very readable. We shall generally prefer to present these definitions first in a more usual functional programming style, à la ML. It is to be expected that future versions of Gallina will accept a more liberal and intuitive syntax.

Let us now turn to the subject matter. We shall assume a certain familiarity with the elementary notions of pure $\lambda$-calculus, such as $\alpha$-conversion and $\beta$-reduction, and we expect the reader to be familiar with the Tait-Martin-Löf method of proving the Church-Rosser property. The basic idea is to show the parallel-moves lemma, a diamond-like elementary paving diagram, which states that if a term $M$ reduces to terms $N$ and $P$, then $N$ and $P$ reduce in turn to a common term $Q$. Here reduction means parallel reduction of possibly several redexes, since the reduction from say $M$ to $N$ may duplicate some redex used in the reduction from $M$ to $P$ into a set of redexes, called its *residuals*. Furthermore, we may not restrict ourselves to parallel reduction of mutually disjoint redexes, since the residuals of disjoint redexes may not be disjoint. This is an essential difficulty of $\lambda$-calculus, which makes the proof significantly harder than say for first order rewriting of orthogonal systems.

The essential notion for this proof is that of residual. One of the contributions of this paper is to propose a clear inductive definition of this notion, as a refinement of one step of parallel reduction. It is more intuitive

[1]This research was partially supported by ESPRIT Basic Research Action "TYPES."

than the standard "residual map" on occurrences, and allows clear inductive proofs. One of the main new results is a Commutation Theorem, which states that residuals commute with substitution. This gives a nice algebraic structure to the inductive type representing sets of redexes, a natural enrichment of $\lambda$-terms. The parallel-moves lemma generalises on this structure as Lévy's Cube Lemma, presented here as a corollary to a simpler diagram which we call the Prism Theorem.

## 2 Lambda terms

We represent our $\lambda$-terms with de Bruijn's indexes[5]. We need a minimum of arithmetical properties, concerning addition, and the standard orderings $<$, $\leq$ and $>$ on natural numbers. Two lemmas state the decidability of those predicates:

```
Lemma test : (n,m:nat){le n m}+{gt n m}.
Lemma compare : (n,m:nat){lt n m}+{<nat>n=m}+{gt n m}.
```

Here as everywhere in Gallina the prefix `(n:nat)` stands for universal quantification $\forall n \in nat$. The connective `+` is intuitionistic disjunction, with constructive contents. We may use the proofs of these lemmas as boolean conditionals in further definitions.

Throughout this paper, we shall not provide the full formal proofs of the lemmas we state. The basic idea is that the development is sliced into small enough pieces that the logical structure of the mathematics is more or less obvious. Hints such as "this lemma is proved by induction on X" are occasionally provided. The reader interested by proof details may consult the actual transcript files in the Coq distribution.

### 2.1 Abstract syntax

The abstract syntax of $\lambda$-terms is defined as an inductive set with three constructors, corresponding respectively to variable occurrences (represented as the reference depth from their binding abstraction), lambda abstraction and application.

```
Inductive Set lambda =
    Ref : nat -> lambda
  | Abs : lambda -> lambda
  | App : lambda -> lambda -> lambda.
```

### 2.2 Lifting

The first operation, an auxiliary notion necessary for substitution, is *lifting*, which recomputes references to global variables across $n$ levels of extra binders in term $N$. The operation `(lift n N)` is itself defined as the base case ($k = 0$) of a more general `(lift_rec n N k)`, where the operation `lift_rec` is defined recursively as follows:

```
Recursive Definition lift_rec n N k = Match N with
    Ref(i)   -> if i<k then Ref(i)
                       else Ref(n+i)
  | Abs(M)   -> Abs(lift_rec n M (k+1))
  | App(M,N) -> App(lift_rec n M k,lift_rec n N k).
```

Note that the argument `k` is just a counter of the abstraction nodes traversed during the recursion. Thus when we encounter a variable represented as a de Bruijn index `Ref(i)`, this variable is local to the $\lambda$-term `N` when $i < k$ (in which case it is untouched), and global otherwise (in which case is it relocated).

The above definition of `lift_rec` is not accepted in Gallina's current syntax analyser, and we must write our recursive definitions in an equivalent but more awkward primitive-recursive style as follows.

```
Definition lift_rec : nat -> lambda -> nat -> lambda =
    [n:nat][N:lambda](<nat->lambda>Match N with
      (* (Ref i) *) [i,k:nat](<lambda>Match (test k i) with
                  (* k<=i *) [H:(le k i)] (Ref (plus n i))
                  (* k>i  *) [H:(gt k i)] (Ref i))
      (* (Abs M) *) [M:lambda][f:nat->lambda]
                    [k:nat](Abs (f (S k)))
      (* (App M N) *) [M:lambda][f:nat->lambda][N:lambda][g:nat->lambda]
                      [k:nat](App (f k) (g k))).

Definition lift = [n:nat][N:lambda](lift_rec n N O).
```

We use the Automath notation for abstraction: `[n:nat]n` denotes the identity function on type `nat`. We use the Lisp notation for application: `(F x y)` denotes the application of function `F` to arguments `x` and `y`.

## 2.3 Substitution

We now define substitution. We again give first the intuitive recursive definition, then the actual definition as currently accepted by Coq.

```
Recursive Definition subst_rec N M k = Match M with
    Ref(i)   -> if k<i then Ref(i-1)
                if k=i then lift k N
                if k>i then Ref(i)
 |  Abs(P)   -> Abs(subst_rec N P (k+1))
 |  App(P,Q) -> App(subst_rec N P k, subst_rec N Q k).

Definition subst_rec : lambda -> lambda -> nat -> lambda =
    [N,M:lambda](<nat->lambda>Match M with
      (* (Ref i) *) [i,k:nat](<lambda>Match (compare k i) with
           [C:{(gt i k)}+{<nat>k=i}](<lambda>Match C with
              (* k<i *) [H:(gt i k)](Ref (pred i))
              (* k=i *) [H:<nat>k=i](lift k N))
              (* k>i *) [H:(gt k i)] (Ref i))
      (* (Abs M) *) [M:lambda][f:nat->lambda]
                    [k:nat](Abs (f (S k)))
      (* (App M N) *) [M:lambda][f:nat->lambda][N:lambda][g:nat->lambda]
                      [k:nat](App (f k) (g k))).

Definition subst = [N,M:lambda](subst_rec N M O).
```

Let us give a few examples. The concrete $\lambda$-expression usually written $\lambda z \cdot z$, for which we shall prefer the Automath syntax `[z]z`, is represented abstractly as the term `(Abs (Ref O)):lambda`. This representation is canonical, in that it is invariant by renaming of the variables (usually called $\alpha$-conversion). Thus we do not have to burden ourselves from the start with an awkward quotient structure. Similarly, $\lambda$-expression `[y](y x)` is represented as `(Abs (App (Ref O) (Ref (S O))))`. Here we assume that the free variable `x` is bound by the immediately enclosing abstraction, like in the redex `([x,y](y x) [z]z)`. Reducing this redex will produce the computation of term `(subst (Abs (Ref O)) (Abs (App (Ref O) (Ref (S O)))))`, which computes to the normal-form term `(Abs (App (Ref O) (Abs (Ref O))))`, i.e. to the expected $\lambda$-expression `[y](y [z]z)`.

Similarly, the reduction of the redex in the expression: `[u]([x][y](x u) [z]u)` will compute the term `(Abs (subst (Abs (Ref (S O))) (Abs (App (Ref (S O)) (Ref (S (S O)))))))`, with normal-form `(Abs (Abs (App (Abs (Ref (S (S O)))) (Ref (S O)))))`, representing e.g. `[x,y]([z]x x)`.

## 3 Reduction

### 3.1 One-step $\beta$-reduction

We may now axiomatize one step of $\beta$-reduction as the congruence closure of rule `beta`, which reduces redex `(App (Abs M) N)` to the result of the substitution of term `N` in term `M`, i.e. to the term `(subst N M)`. We thus obtain naturally `red1` as an inductively defined relation with four constructors, corresponding to the usual structured operational semantics rules:

```
Inductive Definition red1 : lambda -> lambda -> Prop =
   beta :      (M,N:lambda)(red1 (App (Abs M) N) (subst N M))
 | abs_red :   (M,N:lambda)(red1 M N) -> (red1 (Abs M) (Abs N))
 | app_red_l : (M1,N1:lambda)(red1 M1 N1) -> (M2:lambda)(red1 (App M1 M2) (App N1 M2))
 | app_red_r : (M2,N2:lambda)(red1 M2 N2) -> (M1:lambda)(red1 (App M1 M2) (App M1 N2)).
```

Remark that the above definition corresponds exactly to stating the usual inference rules:

$$beta : \frac{}{(\texttt{[x]M N}) \rightarrow_1 \texttt{M\{N\textbackslash x\}}}$$

$$abs\_red : \frac{\texttt{M} \rightarrow_1 \texttt{N}}{\texttt{[x]M} \rightarrow_1 \texttt{[x]N}}$$

$$app\_red\_l : \frac{\texttt{M}_1 \rightarrow_1 \texttt{N}_1}{(\texttt{M}_1\ \texttt{M}_2) \rightarrow_1 (\texttt{N}_1\ \texttt{M}_2)}$$

$$app\_red\_r : \frac{\texttt{M}_2 \rightarrow_1 \texttt{N}_2}{(\texttt{M}_1\ \texttt{M}_2) \rightarrow_1 (\texttt{M}_1\ \texttt{N}_2)}$$

### 3.2 $\beta$-reduction

We now define $\beta$-reduction `red` as the transitive closure of `red1`.

```
Inductive Definition red : lambda -> lambda -> Prop =
   one_step_red  : (M,N:lambda)(red1 M N) -> (red M N)
 | refl_red      : (M:lambda)(red M M)
 | trans_red     : (M,N,P:lambda)(red M N) -> (red N P) -> (red M P).
```

Here are a few typical lemmas, easy to prove by the induction principle naturally associated with the inductive definition `red`.

```
Lemma red_abs : (M,M':lambda)(red M M') -> (red (Abs M) (Abs M')).

Lemma red_appl : (M,M':lambda)(red M M') -> (N:lambda)(red (App M N) (App M' N)).

Lemma red_appr : (M,M':lambda)(red M M') -> (N:lambda)(red (App N M) (App N M')).
```

Using the transitivity of `red`, we now show that `red` is closed by $\beta$-reduction:

```
Lemma red_app : (M,M',N,N':lambda)(red M M') -> (red N N') -> (red (App M N) (App M' N')).

Lemma red_beta :
  (M,M',N,N':lambda)(red M M') -> (red N N') -> (red (App (Abs M) N) (subst N' M')).
```

## 3.3 $\beta$-conversion

Similarly, $\beta$-conversion `conv` may be defined as the equivalence closure of `red1`. Actually, it is more convenient to first define one step of conversion as one step of reduction or anti-reduction, to take its reflexive-transitive closure, and to prove symmetry:

```
Inductive Definition conv1 : lambda -> lambda -> Prop =
   red1_conv  : (M,N:lambda)(red1 M N) -> (conv1 M N)
 | exp1_conv  : (M,N:lambda)(red1 N M) -> (conv1 M N).

Inductive Definition conv : lambda -> lambda -> Prop =
   one_step_conv  : (M,N:lambda)(conv1 M N) -> (conv M N)
 | refl_conv  : (M:lambda)(conv M M)
 | trans_conv : (M,N,P:lambda)(conv M N) -> (conv N P) -> (conv M P).

Lemma sym_conv : (M,N:lambda)(conv M N) -> (conv N M).
```

## 3.4 Parallel $\beta$-reduction

We define similarly one step of parallel $\beta$-reduction, with the usual bottom-up inductive definition.

```
Inductive Definition par_red1 : lambda -> lambda -> Prop =
   par_beta    : (M,M':lambda)(par_red1 M M') ->
                 (N,N':lambda)(par_red1 N N') -> (par_red1 (App (Abs M) N) (subst N' M'))
 | ref_par_red : (n:nat)(par_red1 (Ref n) (Ref n))
 | abs_par_red : (M,M':lambda)(par_red1 M M') -> (par_red1 (Abs M) (Abs M'))
 | app_par_red : (M,M':lambda)(par_red1 M M') ->
                 (N,N':lambda)(par_red1 N N') -> (par_red1 (App M N) (App M' N')).
```

Again, this should be compared to:

$$par\_beta : \frac{\mathtt{M} \Rightarrow \mathtt{M'} \quad \mathtt{N} \Rightarrow \mathtt{N'}}{([\mathtt{x}]\mathtt{M}\ \mathtt{N}) \Rightarrow \mathtt{M'}\{\mathtt{N'}\backslash\mathtt{x}\}}$$

$$ref\_par\_red : \frac{}{\mathtt{x} \Rightarrow \mathtt{x}}$$

$$abs\_par\_red : \frac{\mathtt{M} \Rightarrow \mathtt{M'}}{[\mathtt{x}]\mathtt{M} \Rightarrow [\mathtt{x}]\mathtt{M'}}$$

$$app\_par\_red : \frac{\mathtt{M} \Rightarrow \mathtt{M'} \quad \mathtt{N} \Rightarrow \mathtt{N'}}{(\mathtt{M}\ \mathtt{N}) \Rightarrow (\mathtt{M'}\ \mathtt{N'})}$$

Let us give a few easy lemmas: `par_red1` is reflexive and extends `red1`.

```
Lemma refl_par_red1 : (M:lambda)(par_red1 M M).

Lemma red1_par_red1 : (M,N:lambda)(red1 M N) -> (par_red1 M N).
```

Both lemmas are immediate by induction on $M$. We now define parallel $\beta$-reduction `par_red` as the transitive closure of `par_red1`.

```
Inductive Definition par_red : lambda -> lambda -> Prop =
   one_step_par_red : (M,N:lambda)(par_red1 M N) -> (par_red M N)
 | trans_par_red    : (M,N,P:lambda)(par_red M N) -> (par_red N P) -> (par_red M P).
```

## 3.5 Equivalence between reduction and parallel reduction

```
Lemma red_par_red : (M,N:lambda)(red M N) -> (par_red M N).
```

```
Lemma par_red_red : (M,N:lambda) (par_red M N) -> (red M N).
```

Again, lemma `red_par_red` is easily proved, by induction on the derivation of `(red M N)`. Similarly, lemma `par_red_red` is proved by induction on `(par_red M N)`. In Coq, such proofs are easy, since the system automatically synthesises an induction principle for any inductively defined notion, and this principle is directly invoked by the `Induction` proof tactic. For instance, the induction principle associated with the inductive predicate `red` is a second-order construction `red_ind`, with type:

```
red_ind : (R:lambda->lambda->Prop)
          ((M:lambda)(N:lambda)(red1 M N)->(R M N)) ->
          ((M:lambda)(R M M)) ->
          ((M,N,P:lambda)(red M N)->(R M N)->(red N P)->(R N P)->(R M P)) ->
          (M,N:lambda)(red M N)->(R M N).
```

## 3.6 Confluence and strip lemmas

We define confluence abstractly, with parameters `A:Set` and relation `R:A->A->Prop`. This definition is Gallina syntax for

$$\text{confluence}(R) \equiv \forall x, y, z \in A \cdot R(x, y) \wedge R(x, z) \Rightarrow \exists u \in A \cdot R(y, u) \wedge R(z, u).$$

The particular order of quantification used below is best suited for the subsequent induction proofs.

```
Definition confluence [A:Set][R:A->A->Prop]
    (x,y:A)(R x y) -> (z:A)(R x z) -> <A>Ex([u:A] (R y u) /\ (R z u)).
```

The next lemma is an easy consequence of the equivalence between reduction and parallel reduction.

```
Lemma lemma1 : (confluence lambda par_red) -> (confluence lambda red).
```

The next lemmas are classical "strip lemmas", like in Barendregt[3, 4].

```
Definition strip = (x,y:lambda)(par_red x y) ->
                     (z:lambda)(par_red1 x z) ->
          <lambda>Ex([u:lambda](par_red1 y u) /\ (par_red z u)).
```

```
Lemma strip_lemma_r : (confluence lambda par_red1) -> strip.
```

In more usual notation, writing $\Rightarrow$ for one step of parallel reduction, this lemma says that if $\Rightarrow$ is confluent, then $\forall x, y, z\ x \Rightarrow^* y \wedge x \Rightarrow z \supset \exists u\ y \Rightarrow u \wedge z \Rightarrow^* u$. In the standard proof, this is an easy induction on $n$ such that $x \Rightarrow^n y$. In Coq we do this by a direct induction on the hypothesis `(par_red x y)`, there is no need to go through an arithmetic coding. A second "strip lemma" in the other direction completes the equivalence between confluence of one step of parallel reduction and its transitive closure:

```
Lemma strip_lemma_l : strip -> (confluence lambda par_red).
```

```
Lemma lemma2 : (confluence lambda par_red1) -> (confluence lambda par_red).
```

# 4 Redexes

We represent sets of redex occurrences as terms with an extra Boolean mark to application nodes. A redex occurrence `(Ap b (Fun M) N)` belongs to the set iff `b=true`.

## 4.1 Redexes as an enrichment of terms

```
Inductive Set redexes =
    Var : nat -> redexes
  | Fun : redexes -> redexes
  | Ap  : bool -> redexes -> redexes -> redexes.
```

We define the translation from terms to (empty) sets of redexes, and the reverse forgetful translation.

```
Definition mark = [M:lambda](<redexes>Match M with
  (* Ref *) [n:nat](Var n)
  (* Abs *) [M:lambda][U:redexes](Fun U)
  (* App *) [M:lambda][U:redexes][N:lambda][V:redexes](Ap false U V)).

Definition unmark = [U:redexes](<lambda>Match U with
  (* Var *) [n:nat](Ref n)
  (* Fun *) [U:redexes][M:lambda](Abs M)
  (* Ap  *) [b:bool][U:redexes][M:lambda][V:redexes][N:lambda](App M N)).

Lemma inverse : (M:lambda)<lambda>M=(unmark (mark M)).
```

## 4.2 The Boolean algebra of sets of redexes

The structure of redexes is going to be used for two orthogonal purposes: as representations of proofs of one step of parallel reduction, and as sets of redexes the residuals of which we are interested in tracing. Sets of redexes have a natural structure of Boolean algebra, with ordering the subset relation `sub`, and join `union` defined inductively below.

```
Inductive Definition sub : redexes -> redexes -> Prop =
    Sub_Var : (n:nat)(sub (Var n) (Var n))
  | Sub_Fun : (U,V:redexes)(sub U V) -> (sub (Fun U) (Fun V))
  | Sub_Ap1 : (U1,V1:redexes)(sub U1 V1) -> (U2,V2:redexes)(sub U2 V2) ->
                    (b:bool)(sub (Ap false U1 U2) (Ap b V1 V2))
  | Sub_Ap2 : (U1,V1:redexes)(sub U1 V1) -> (U2,V2:redexes)(sub U2 V2) ->
                    (b:bool)(sub (Ap true U1 U2) (Ap true V1 V2)).

Definition bool_max = [b,b':bool](<bool>Match b with true b').

Inductive Definition union : redexes -> redexes -> redexes -> Prop =
    Union_Var : (n:nat)(union (Var n) (Var n) (Var n))
  | Union_Fun : (U,V,W:redexes)(union U V W) -> (union (Fun U) (Fun V) (Fun W))
  | Union_Ap  : (U1,V1,W1:redexes)(union U1 V1 W1) ->
                (U2,V2,W2:redexes)(union U2 V2 W2) ->
      (b1,b2:bool)(union (Ap b1 U1 U2) (Ap b2 V1 V2) (Ap (bool_max b1 b2) W1 W2)).

Lemma union_l : (U,V,W:redexes)(union U V W) -> (sub U W).

Lemma union_r : (U,V,W:redexes)(union U V W) -> (sub V W).

Lemma union_sym : (U,V,W:redexes)(union U V W) -> (union V U W).
```

The compatibility relation in this lattice is the equivalence `comp`, with `(comp U V)` if and only if `(unmark U)=(unmark V)`.

```
Inductive Definition comp : redexes -> redexes -> Prop =
    Comp_Var : (n:nat)(comp (Var n) (Var n))
  | Comp_Fun : (U,V:redexes)(comp U V) -> (comp (Fun U) (Fun V))
  | Comp_Ap  : (U1,V1:redexes)(comp U1 V1) ->
               (U2,V2:redexes)(comp U2 V2) ->
                  (b1,b2:bool)(comp (Ap b1 U1 U2) (Ap b2 V1 V2)).

Lemma comp_refl : (U:redexes)(comp U U).

Lemma comp_sym : (U,V:redexes)(comp U V) -> (comp V U).

Lemma comp_trans : (U,V:redexes)(comp U V) -> (W:redexes)(comp V W) -> (comp U W).

Lemma union_defined : (U,V:redexes)(comp U V) -> <redexes>Ex([W:redexes](union U V W)).

Lemma comp_unmark_eq : (U,V:redexes)(comp U V) -> <lambda>(unmark U)=(unmark V).
```

We do not state the converse of this last lemma, which is not needed in the following.

### 4.3 Regularity

An element of type `redexes` is said to be `regular` if its true marks label only redexes. We want to forbid non-regular values of this type such as `(Ap true (Var 0) (Var 0))`, which are meaningless for the representation of sets of redexes. We define this predicate recursively, first informally, then formally.

```
Recursive Definition regular U = Match U with
     Var(n) -> True
   | Fun(V) -> regular(V)
   | Ap(true,V,W) -> (Match V with Fun(_) -> regular(V) /\ regular(W)
                                 | _ -> False)
   | Ap(false,V,W) -> regular(V) /\ regular(W).

Definition regular = [U:redexes](<Prop>Match U with
  (* Var *) [n:nat]True
  (* Fun *) [U1:redexes][P:Prop]P
  (* Ap  *) [b:bool][V:redexes][P:Prop][W:redexes][Q:Prop]
       (<Prop>Match b with (* true *) (<Prop>Match V with
  (* Var *) [n:nat]False
  (* Fun *) [U1:redexes][R:Prop](P /\ Q)
  (* Ap  *) [b':bool][V':redexes][P':Prop][W':redexes][Q':Prop]False)
                          (* false *) (P /\ Q))).

Lemma union_preserve_regular :
    (U,V,W:redexes)(union U V W) -> (regular U) -> (regular V) -> (regular W).
```

## 5 Substitution of redexes inside redexes

We develop the theory of substitution for the structure `redexes`, similarly to what we did for the terms. We first give a number of tedious technical lemmas concerning the extension of substitution to redexes. Corresponding lemmas for terms could be obtained by forgetting the Boolean marks. We advise the reader to skip the following two sections, unless he is interested in the theory of de Bruijn indexes managing, and to look directly at lemma `substitution` below.

## 5.1 Lifting

We just copy the above definition of `lift_rec`, just enriching the application node with its boolean mark:

```
Definition lift_rec_r : nat -> redexes -> nat -> redexes =
    [n:nat][P:redexes](<nat->redexes>Match P with
      (* (Var i) *) [i,k:nat](Var (<nat>Match (test k i) with
                 (* k<=i *) [H:(le k i)] (plus n i)
                 (* k>i  *) [H:(gt k i)] i))
      (* (Fun M) *) [M:redexes][f:nat->redexes]
                    [k:nat](Fun (f (S k)))
      (* (Ap b M N) *) [b:bool][M:redexes][f:nat->redexes]
                               [N:redexes][g:nat->redexes]
                               [k:nat](Ap b (f k) (g k))).

Lemma lift_le : (n,i,k:nat)(le k i) ->
     <redexes>(lift_rec_r n (Var i) k) = (Var (plus n i)).

Lemma lift_gt : (n,i,k:nat)(gt k i) ->
     <redexes>(lift_rec_r n (Var i) k) = (Var i).

Lemma lift1 : (U:redexes)(j,i,k:nat)<redexes>
     (lift_rec_r k (lift_rec_r j U i) (plus j i)) = (lift_rec_r (plus j k) U i).

Lemma lift_lift_rec : (U:redexes)(k,p,n,i:nat)(le i n) -> <redexes>
     (lift_rec_r k (lift_rec_r p U i) (plus p n)) = (lift_rec_r p (lift_rec_r k U n) i).

Definition lift_r = [n:nat][U:redexes](lift_rec_r n U O).

Lemma lift_lift : (U:redexes)(k,p,n:nat)<redexes>
      (lift_rec_r k (lift_r p U) (plus p n)) = (lift_r p (lift_rec_r k U n)).

Lemma liftrecO : (U:redexes)(n:nat)<redexes>(lift_rec_r O U n) = U.

Lemma liftO : (U:redexes)<redexes>(lift_r O U) = U.

Lemma lift_rec_lift_rec : (U:redexes)(n,p,k,i:nat)
      (le k (plus i n)) -> (le i k) -> <redexes>
      (lift_rec_r p (lift_rec_r n U i) k) = (lift_rec_r (plus p n) U i).

Lemma lift_rec_lift : (U:redexes)(n,p,k,i:nat)(le k n) -> <redexes>
      (lift_rec_r p (lift_r n U) k) = (lift_r (plus p n) U).
```

## 5.2 Substitution

```
Definition subst_rec_r : redexes -> redexes -> nat -> redexes =
    [N,M:redexes](<nat->redexes>Match M with
      (* (Var i) *) [i,k:nat](<redexes>Match (compare k i) with
           [C:{(gt i k)}+{<nat>k=i}](<redexes>Match C with
              (* k<i *) [H:(gt i k)](Var (pred i))
              (* k=i *) [H:<nat>k=i](lift_r k N))
              (* k>i *) [H:(gt k i)] (Var i))
      (* (Fun M) *) [M:redexes][f:nat->redexes]
                    [k:nat](Fun (f (S k)))
      (* (Ap b M N) *) [b:bool][M:redexes][f:nat->redexes]
                               [N:redexes][g:nat->redexes]
                               [k:nat](Ap b (f k) (g k))).

Lemma subst_eq : (U:redexes)(n:nat)<redexes>
      (subst_rec_r U (Var n) n) = (lift_r n U).

Lemma subst_gt : (U:redexes)(n,p:nat)(gt n p) -> <redexes>
      (subst_rec_r U (Var n) p) = (Var (pred n)).

Lemma subst_lt : (U:redexes)(n,p:nat)(gt p n) -> <redexes>
      (subst_rec_r U (Var n) p) = (Var n).

Lemma lift_rec_subst_rec : (U,V:redexes)(k,p,n:nat)<redexes>
      (lift_rec_r k (subst_rec_r U V p) (plus p n)) =
      (subst_rec_r (lift_rec_r k U n) (lift_rec_r k V (S (plus p n))) p).
```

## 5.3 The Substitution Lemma

```
Definition subst_r = [V,U:redexes](subst_rec_r V U O).

Lemma lift_subst : (U,V:redexes)(k,n:nat)<redexes>
      (lift_rec_r k (subst_r U V) n) =
      (subst_r (lift_rec_r k U n) (lift_rec_r k V (S n))).

Lemma subst_rec_lift_rec1 : (U,V:redexes)(n,p,k:nat)(le k n) -> <redexes>
      (subst_rec_r V (lift_rec_r p U k) (plus p n)) =
      (lift_rec_r p (subst_rec_r V U n) k).

Lemma subst_lift1 : (U,V:redexes)(n,p:nat)<redexes>
      (subst_rec_r V (lift_r p U) (plus p n)) = (lift_r p (subst_rec_r V U n)).

Lemma subst_rec_lift_rec : (U,V:redexes)(p,q,n:nat)
      (le q (plus p n)) -> (le n q) -> <redexes>
      (subst_rec_r V (lift_rec_r (S p) U n) q) = (lift_rec_r p U n).

Lemma subst_rec_lift : (U,V:redexes)(p,q:nat)(le q p) -> <redexes>
      (subst_rec_r V (lift_r (S p) U) q) = (lift_r p U).

Lemma subst_rec_subst_rec : (U,V,W:redexes)(n,p:nat)<redexes>
      (subst_rec_r W (subst_rec_r U V p) (plus p n)) =
      (subst_rec_r (subst_rec_r W U n) (subst_rec_r W V (S (plus p n))) p).
```

After this painful technical development, we finally get the important Substitution Lemma, which says (roughly):

$$W\,[x \leftarrow U]\,[y \leftarrow V] = W\,[y \leftarrow V]\,[x \leftarrow U\,[y \leftarrow V]\,].$$

```
Lemma substitution : (W,U,V,:redexes)(n:nat)<redexes>
      (subst_rec_r V (subst_r W U) n) =
      (subst_r (subst_rec_r V W n) (subst_rec_r V U (S n))).
```

The argument $n+1$ in the right hand side accounts for the asymmetry in `(subst N M)`, which corresponds to reducing a redex `([x]M N)`: a global variable referenced by index $n$ in $N$ is referenced by index $n+1$ in $M$, since there is the extra binder for the substituted variable $x$ to account for.

### 5.4 Preservation Lemmas

We now show that substitution preserves compatibility and regularity.

```
Lemma lift_rec_preserve_comp : (U1,V1:redexes)(comp U1 V1) ->
     (n,m:nat)(comp (lift_rec_r n U1 m) (lift_rec_r n V1 m)).

Lemma subst_rec_preserve_comp :
     (U1,V1,U2,V2:redexes)(comp U1 V1) -> (comp U2 V2) ->
     (n:nat)(comp (subst_rec_r U1 U2 n) (subst_rec_r V1 V2 n)).

Lemma subst_preserve_comp :
    (U1,V1,U2,V2:redexes)(comp U1 V1) -> (comp U2 V2) ->
    (comp (subst_r U2 U1) (subst_r V2 V1)).

Lemma lift_rec_preserve_regular :
      (U:redexes)(regular U) -> (n,m:nat)(regular (lift_rec_r n U m)).

Lemma subst_rec_preserve_regular :
  (U,V:redexes)(regular U) -> (regular V) -> (n:nat)(regular (subst_rec_r U V n)).

Lemma subst_preserve_regular :
      (U,V:redexes)(regular U) -> (regular V) -> (regular (subst_r U V)).
```

## 6 Residuals

We develop a strengthening of parallel $\beta$-reduction, with residual tracing. Here `(residuals U V W)` means redexes `W` are residuals of redexes `U` by one step of parallel reduction of all redexes `V`. Note how this definition follows naturally the structure of the definition of parallel reduction above, instead of the usual rather arbitrary looking definition of the residual map between positions in the terms.

```
Inductive Definition residuals : redexes -> redexes -> redexes -> Prop =
   Res_Var : (n:nat)(residuals (Var n) (Var n) (Var n))
 | Res_Fun : (U,V,W:redexes)(residuals U V W) -> (residuals (Fun U) (Fun V) (Fun W))
 | Res_Ap : (U1,V1,W1:redexes)(residuals U1 V1 W1) ->
            (U2,V2,W2:redexes)(residuals U2 V2 W2) ->
            (b:bool)(residuals (Ap b U1 U2) (Ap false V1 V2) (Ap b W1 W2))
 | Res_redex : (U1,V1,W1:redexes)(residuals U1 V1 W1) ->
               (U2,V2,W2:redexes)(residuals U2 V2 W2) ->
     (b:bool)(residuals (Ap b (Fun U1) U2) (Ap true (Fun V1) V2) (subst_r W2 W1)).
```

The relation `residuals` defines a partial function:

```
Lemma residuals_function : (U,V,W:redexes)(residuals U V W) ->
                    (W':redexes)(residuals U V W') -> <redexes>W'=W.
```

### 6.1 The Commutation Theorem

We now prove the crucial commutation theorem. First, a few lemmas.

```
Lemma residuals_lift_rec : (U1,U2,U3:redexes)
   (residuals U1 U2 U3) -> (k,n:nat)
   (residuals (lift_rec_r k U1 n) (lift_rec_r k U2 n) (lift_rec_r k U3 n)).

Lemma residuals_lift : (U1,U2,U3:redexes)
```

```
   (residuals U1 U2 U3) -> (k:nat)
   (residuals (lift_r k U1) (lift_r k U2) (lift_r k U3)).

Lemma residuals_subst_rec : (U1,U2,U3,V1,V2,V3:redexes)
   (residuals U1 U2 U3) -> (residuals V1 V2 V3) -> (k:nat)
   (residuals (subst_rec_r V1 U1 k) (subst_rec_r V2 U2 k) (subst_rec_r V3 U3 k)).
```

Thus, we get the commutation theorem, which states that residuals commute with substitution.

```
Theorem commutation : (U1,U2,U3,V1,V2,V3:redexes)
   (residuals U1 U2 U3) -> (residuals V1 V2 V3) ->
   (residuals (subst_r V1 U1) (subst_r V2 U2) (subst_r V3 U3)).
```

Using $V/U$ for the substitution of $V$ in $U$ and $U\backslash V$ for the residuals of $U$ by $V$, unique when they exist, we would write this result, using the standard mathematical conventions for partial operations:

**Commutation Theorem.** If $U_1$ and $V_1$ (resp. $U_2$ and $V_2$) are compatible sets of redexes:

$$(V_1/U_1)\backslash(V_2/U_2) = (V_1\backslash V_2)/(U_1\backslash U_2)$$

To our knowledge, this theorem appeared first in [9]. Remark that, despite its easy formulation, this theorem is not so intuitive. It is simple to say that residuals commute with substitution, but it is another matter to draw a diagram illustrating the general situation...

## 6.2 Residuals, Compatibility and Regularity

We first show two lemmas relating residuals and compatibility.

```
Lemma residuals_comp : (U,V,W:redexes)(residuals U V W) -> (comp U V).

Lemma residuals_preserve_comp : (U,V:redexes)(comp U V) ->
   (W,UW,VW:redexes)(residuals U W UW) -> (residuals V W VW) -> (comp UW VW).
```

We take residuals only by regular redexes. Conversely, residuals by compatible regular redexes always exist (and are unique by the `residuals_function` lemma above). Finally, residuals preserve regularity.

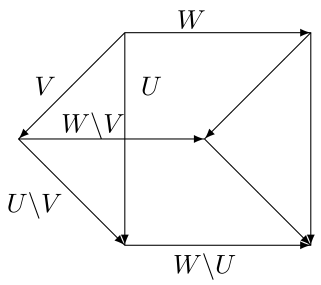
```
Lemma residuals_regular : (U,V,W:redexes)(residuals U V W) -> (regular V).

Lemma residuals_intro : (U,V:redexes)(comp U V) -> (regular V) ->
                        <redexes>Ex([W:redexes](residuals U V W)).

Lemma residuals_preserve_regular :
      (U,V,W:redexes)(residuals U V W) -> (regular U) -> (regular W).
```

## 6.3 The Prism Theorem

We arrive at the main result of this paper.
**The Prism Theorem**. For every compatible sets of redexes $U$, $V$ and $W$:

$$V \subset U \Rightarrow W\backslash U = (W\backslash V)\backslash(U\backslash V).$$

The name "prism theorem" comes from the shape of the picture it evoques, in the same way that Lévy's cube lemma[12] corresponds to the picture of a cube:



In our relational formalization, this theorem is expressed as the conjunction of `prism1` and `prism2`.

```
Lemma prism1 : (U,V,W:redexes)(sub V U) ->
        (UV:redexes)(residuals U V UV) ->
        (WV:redexes)(residuals W V WV) ->
        (WU:redexes)(residuals W U WU) -> (residuals WV UV WU).

Lemma prism2 : (U,V,W:redexes)(sub V U) -> (regular U) ->
        (UV:redexes)(residuals U V UV) ->
        (WV:redexes)(residuals W V WV) ->
        (WU:redexes)(residuals WV UV WU) -> (residuals W U WU).
```

The proof of `prism1` is by simultaneous structural induction on $U$, $V$ and $W$. The key case corresponds to when $U$, $V$ and $W$ are redexes, with $V$ marked, and the result follows then directly from the commutation theorem. We then obtain `prism2` by the `residuals_function` lemma. We now put the two lemmas together as one theorem:

```
Theorem prism : (U,V,W:redexes)(sub V U) ->
                (UV:redexes)(residuals U V UV) ->
                (WV:redexes)(residuals W V WV) ->
   ((WU:redexes)(residuals W U WU) <-> (regular U) /\ (residuals WV UV WU)).
```

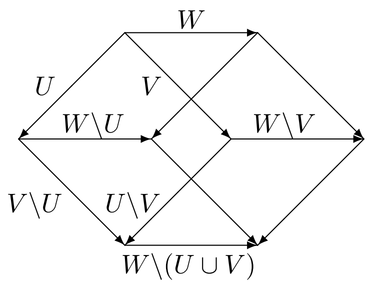
The careful reader may wonder about the slight assymmetry between the two sides of the equivalence. This is due to the fact that `(residuals U V W)` insures only that `V` is regular (this is lemma residuals_regular above), not `U` or `W`. In the informal statement of the theorem, we said "for every compatible sets of redexes $U$, $V$ and $W$", which insures the regularity of $U$, as compatible with regular $V$. Of course here we do not care about non-regular structures, which do not code sets of redexes, but in the formal proof of prism2 the regularity of $U$ is indeed needed to get the theorem in its full generality.

Pictorially, we obtain a cube by "glueing together" two prisms, and the cube lemma below is indeed a direct corollary of the prism theorem.

## 6.4 The Cube Lemma

**The Cube lemma**. For every compatible sets of redexes $U$, $V$ and $W$:

$$(W\backslash V)\backslash(U\backslash V) = (W\backslash U)\backslash(V\backslash U).$$



We first need an auxiliary lemma, showing that for any compatible sets of redexes $U$ and $V$:

$$U\backslash V = (U \cup V)\backslash V.$$

```
Lemma preservation : (U,V,W,UV:redexes)(union U V W) ->
                     (residuals U V UV) -> (residuals W V UV).

Lemma cube :
  (U,V,UV,VU:redexes)  (residuals U V UV) -> (residuals V U VU) ->
  (W,WU,WV,WUV:redexes)(residuals W U WU) -> (residuals WU VU WUV) ->
                       (residuals W V WV) -> (residuals WV UV WUV).
```

The proof of the cube lemma uses `prism1`, `prism2`, and the auxiliary lemma `preservation`.

Combining the cube lemma with the `residuals_intro` property above, we get a general 3-dimensional paving diagram, an essential technical tool for the theory of (parallel) derivations:

```
Lemma paving :
  (U,V,W,WU,WV:redexes)(residuals W U WU) -> (residuals W V WV) ->
  <redexes>Ex([UV:redexes]<redexes>Ex([VU:redexes]<redexes>Ex([WUV:redexes]
          ((residuals WU VU WUV) /\ (residuals WV UV WUV))))).
```

In more usual notation:
**The Paving lemma**. For every compatible sets of redexes $U$, $V$ and $W$, there exist sets of redexes $UV$ and $VU$ such that:

$$(W\backslash U)\backslash VU = (W\backslash V)\backslash UV.$$

This paving diagram is the categorical essence of the diamond property expressed in the next section as the parallel moves lemma. That is, the confluence property of $\beta$-reduction expresses much more than a syntactic coincidence of the common endpoints of the two derivations; it states a categorical diagram, saying that the two derivations preserve any remaining computation (expressed by the argument $W$).

The natural continuation of this theory would be to define the structure of (multi-step) parallel derivations, and to define by induction the residual $A\backslash B$ of a derivation $A$ by a coinitial derivation $B$. We could then define the *permutation* equivalence[12], consider the category whose objects are terms and whose maps are derivations quotiented by permutation, and show that it admits pushouts. We shall not do it here, and rather turn to the application of the paving lemma to confluence results. This is obtained by simply forgetting the marks.

## 7 Confluence

### 7.1 From residuals to reduction

Here we relate redexes and terms, showing that unmarking projects operations on redexes to the corresponding operations on terms.

```
Lemma mark_lift : (M:lambda)(n:nat)<redexes>(lift_r n (mark M)) = (mark (lift n M)).

Lemma mark_subst : (M,N:lambda)<redexes>(subst_r (mark M) (mark N)) = (mark (subst M N)).

Lemma unmark_lift : (U:redexes)(n:nat)
      <lambda>(lift n (unmark U)) = (unmark (lift_r n U)).

Lemma unmark_subst : (U,V:redexes)
     <lambda>(subst (unmark U) (unmark V)) = (unmark (subst_r U V)).
```

We now define reduction of a $\lambda$-term by a set of redexes.

```
Definition reduction = [M:lambda][U:redexes][N:lambda](residuals (mark M) U (mark N)).

Lemma reduction_function :
      (M,N,P:lambda)(U:redexes)(reduction M U N) -> (reduction M U P) -> <lambda>N=P.
```

We then show that `residuals` properly simulates `par_red1`.

```
Lemma simulation : (M,M':lambda)(par_red1 M M') ->
                   <redexes>Ex([V:redexes](reduction M V M')).

Lemma completeness :
     (U,V,W:redexes)(residuals U V W) -> (par_red1 (unmark U) (unmark W)).
```

## 7.2 Parallel Moves and Confluence

We may now get confluence of one-step parallel reduction as a corollary of the paving lemma. Confluence of parallel reduction follows from lemma 2 above. Confluence of $\beta$-reduction follows then from lemma 1.

```
Lemma parallel_moves : (confluence lambda par_red1).

Lemma confluence_parallel_reduction : (confluence lambda par_red).

Theorem confluence_beta_reduction : (confluence lambda red).
```

## 7.3 The Church-Rosser Theorem

Lastly, we give as corollary the Church-Rosser theorem:

```
Theorem Church_Rosser : (M,N:lambda)(conv M N) ->
        <lambda>Ex([P:lambda](red M P) /\ (red N P)).
```

It is proved by an easy induction on `(conv M N)`, using the confluence theorem above and the reflexivity and transitivity of reduction.

## 8 Analysis of this development

We shall briefly analyse this mathematical development, to assess the suitability of the Gallina specification language for this type of mathematical theory, and of Coq as a Proof Assistant.

## 8.1 Resources requirements

An analysis of the suitability of Coq as a Proof Assistant is actually beyond the scope of this paper, since it would require a fuller description of the proof scripts. We limit ourselves here to a sketchy description of the amount of resources which are necessary for working out such a development.

The most crucial resource by far is human time of the proof engineer. The proof engineer must have good familiarity with the relevant mathematical theory, and be well trained in using Coq. Developing the formal proof of a lemma is currently one order of magnitude more complicated than writing the functional program which underlies the proof. In our case, we factored the difficulty by first developing a pre-formal axiomatisation of the same material, using the programming language ML[10]. This pre-formal axiomatisation followed itself an initial development of the main notions in informal mathematics[9].

We then proceeded with the complete development of this example in Coq. We used a mixture of top-down development, using axioms to delay proving technical lemmas, and bottom-up proofs of these. Some of the technical lemmas turned out to be much more complex than anticipated. It took us about one month to get the theory of substitution right, and a couple of weeks to find the right combination of structural induction on redexes and induction on residuals derivation to crack the proof of the prism theorem. The amount of time it takes to get all the arithmetical lemmas and other technical details out of the way, in order to get rid of ALL axioms, is always underestimated. Finally, polishing and restructuring the final development in order to get a honorable-looking piece of mathematics is also a time-consuming task, but a conceptually interesting one. All in all, this was done over a 3 month-period as a kind of background activity interleaved with many other distracting tasks, whose constant interruptions were detrimental to the proper concentration required to manage a large proof. A better estimate would be one full-time monk-month.

In a way, this means that such completely formal mathematical developments are still very costly. On the other hand, it shows that they are indeed feasible.

Let us give briefly what computing resources are involved. The full transcript takes 14' to be verified on a modern Sparc2 workstation, and we are very far from any intrinsic limitation of Coq. Indeed this transcript may be executed on a portable Macintosh micro-computer with only 2 Megabytes of memory for the Coq application. We conclude that modern computing resources are sufficient for developing substantial formal mathematics, and that the real bottleneck is human time.

## 8.2 Comparison with other formalisations of similar material

The first formal development of the Church-Rosser theorem in $\lambda$-calculus was done by N. Shankar on the Boyer-Moore prover[18]. In Shankar's work, `(BUMP X N)` corresponds to our `(lift_rec 1 X N)`, and his `(SUBST X Y N)` corresponds to our `(subst_rec Y X N)`. His `WALK` is basically our `par_red` relation.

Our version of substitution for de Bruijn's indexes is slightly different from Shankar's: when we substitute `Y` inside a `lambda` we do not `BUMP Y`, we lift it only when we encounter the variable to substitute, keeping a counter of the lambdas encountered. Thus we traverse once `Y` for every occurrence of the substituted variable, whereas Shankar traverses `Y` for every abstraction in the term into which he substitutes. The price we pay for this optimisation is that our substitution lemma is harder to prove, since it involves properties of addition. Retrospectically, it is probably a mistake to complicate definitions for computational considerations. The rational for this axiomatisation was to try and follow as faithfully as possible the prior pre-formal development done in [10].

Shankar's development was transposed to the Calculus of Constructions by A. Narayana[14]. Other variants of $\lambda$-calculus formalisations have been developed by S. Berardi[2], C. Coquand[6] and T. Altenkirch[1].

Many variations are possible for representing free variables. Here we fix their relative depth. Thus `(Ref 1)` has more information than just a free variable `x1`, since it tells you that it is `x1` *in a context of the form* `...[x1][x0]`. A still more precise representation would tell the full length $n \geq 2$ of the context. A still more explicit representation would define `lambda` as a dependent type on a `nat` argument denoting the length of the current context. Thus for instance we would get parameterized families of constructors such as `Abs:(n:nat)(lambda (S n))->(lambda n)`. This parameterization seems to complicate matters without definite improvement, though.

All these variations rely on de Bruijn's indexes. A notable exception, using an abstract type of variables, is the recent formalisation of PTS by McKinna and Pollack[13]. Recently, F. Pfenning developed a proof of the Church-Rosser Theorem in Elf using higher-order abstract syntax for its formalisation[16].

## 8.3 Remarks on the contribution to $\lambda$-calculus theory

This type of combinatorial mathematics is well-suited to inductive specifications languages. We claim that the development presented in this paper is rather elegant, without arbitrary codings, and indeed more pleasant esthetically than the classical informal textbook treatment, despite the heavy de Bruijn references machinery and the awkward syntax for recursion. However, we warn the reader that we weeded out of the development a few intermediate notions and lemmas, which were developed to ease the job for the prover.

The commutation theorem is usually presented in the weaker following form, where $\Rightarrow$ stands for one step of parallel reduction. If $M \Rightarrow M'$ and $N \Rightarrow N'$, then $M\texttt{[}x \leftarrow N\texttt{]} \Rightarrow M'\texttt{[}x \leftarrow N'\texttt{]}$. In the development above, we say more about the structure of the resulting reduction. Indeed, by stating it on the structure `redexes` which is used dually for parallel reduction steps and for tracing residuals, we obtain a stronger result on residuals, and the theorem becomes a nice algebraic commutation property.

The prism theorem is not new, in that it is a direct consequence of the cube lemma. What is new is to recognize its importance as the main result indeed. This is where we get a reward from a fully formal development: we have to formulate the notions in exactly the right algebraic way, and we have to find the most elegant composition of arguments in order to get small proofs.

It is essential for the application of the cube lemma to a general theory of derivations to close the diagram with $U \cup V$. If we were only interested in confluence, we could take any upper bound of $U$ and $V$, such as the set of all redexes in the underlying term $unmark(U)$ (this is the usual proof of confluence of 1-step parallel reduction, closing the diagram with full reduction). The minimality of the paving diagram is a key step towards proving that the category of derivations, quotiented with the *permutation* equivalence, admits pushouts.

The regularity requirements are an obnoxious side-effect of our axiomatization of sets redexes as marked terms. Finer-grain representation of $\lambda$-calculus terms, for instance as sharing graphs implemented with interaction nets, may be better adapted to represent sets of redexes (sets of interactions) by labelling the *arcs* of the graph. Here in this term representation we can label only *vertices*, and the regularity requirement comes from the fact that instead of labeling an arc between an `App` node and an `Abs` node, we have to label either the `App` or the `Abs`, opening the spurious case of a non-regular labeling.

A further remark on this point is that we indeed have the choice of whether to label the `App` or the `Abs`, and that Barendregt's *underlining method*[4] is dual to ours, in that he rather labels the `Abs` nodes of redexes. In a way this does not matter, but in another it does. Since the argumentation follows the inductive structure of the marked terms, labeling the (top) `App` tells us whether the current term is a redex or not, without having to look at its first subterm by one further induction, leading to extraneous inductive cases.

## 8.4 Comparison with other logical frameworks and systems

One of the most awkward features of the axiomatization above is the difficulty of dealing with partial functions. We would really prefer to use a functional notation for the residuals map, in order to get the prism theorem in one piece. Similarly we would like to use functional notation for `union`, which is defined only when its two arguments are compatible. This drawback originates from `Coq`'s standard viewpoint towards logic axiomatizations, in contrast to "free logic" treatments such as provided for instance in the IMPS system[8], which are closer to mathematical practice with regards to partial operations.

We remark that it is painful to duplicate all the definitions of lifting and substitution when going from terms to redexes. Somehow this structure enrichment should correspond to a notion of subtyping in the logical framework, in the spirit of object-oriented programming: the Boolean mark ought to be just an *attribute* to the term values, and substitution of terms ought to be applicable when the attribute is present. The notational difficulty is that we want to be able to specify which attributes are copied, and which are initialized to a default value, when recursing over the structure.

`Coq` stands somewhere in the spectrum between the Mizar project and the Boyer-Moore Computational Logic system. It is certainly less advanced than Mizar[17] from the points of view of existence of tools to manipulate theories written in a mathematical vernacular, with automatic synthesis of TeX abstracts, and of coordination of a community of competent users contributing to add to a general library of Formalized

Mathematics. On the other hand, Coq is better adapted to the definition of inductive notions, and more generally to constructive mathematics, than Mizar, based on a variant of classical set theory.

Compared to the Boyer-Moore system, Coq is far less advanced from the point of view of automation of reasoning, but its specification language, based on Type Theory, is definitely closer to mathematical practice.

It would be interesting to make detailed comparisons of similar developments in the various Proof Assistants currently used for formal developments: HOL, Nuprl, B, and more recently LEGO, Isabelle and ALF.

## References

[1] T. Altenkirch. "A formalisation of the strong normalization proof for System F in LEGO." To appear, Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, March 1993.

[2] S. Berardi. "Girard's normalisation proof in LEGO." Unpublished draft note, 1991.

[3] H. Barendregt. "The Lambda-Calculus: Its Syntax and Semantics." North-Holland (1980).

[4] H. Barendregt. "Lambda-Calculus with Types." In Handbook of Logic in Computer Science, Vol II, Ed. S. Abramsky, D. Gabbay and T. Maibaum, Oxford University Press, 1993.

[5] N.G. de Bruijn. "Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem." Indag. Math. **34,5** (1972), 381–392.

[6] C. Coquand. "A proof of normalization for simply typed lambda calculus written in ALF." Proceedings of the 1992 Workshop on Types for Proofs and Programs, Eds. B. Nordström, K. Petersson and G. Plotkin. Available by anonymous ftp from animal.cs.chalmers.se.

[7] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin, B. Werner. "The Coq Proof Assistant User's Guide, Version 5.6." INRIA Technical Report 134, Dec. 1991.

[8] W. M. Farmer, J. D. Guttman and F. J. Thayer. "IMPS: an Interactive Mathematical Proof System." Technical Report M90-19, MITRE Corporation, 1991.

[9] G. Huet. "Initiation à la calculabilité." Notes de Cours, DEA Université Paris 7, Jan. 1988.

[10] G. Huet. "Constructive Computation Theory, Part I." Course Notes, DEA Informatique, Mathématiques et Applications, Paris, Oct. 1992.

[11] G. Huet. "The Gallina specification language : A case study". Proceedings of 12th FST/TCS Conference, New Delhi, Dec. 1992. Ed. R. Shyamasundar, Springer Verlag LNCS 652, pp. 229–240.

[12] J. J. Lévy. "Réductions correctes et optimales dans le $\lambda$-calcul." Thèse d'Etat, U. Paris VII (1978).

[13] J. McKinna and R. Pollack. "Pure Type Systems Formalized." To appear, Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, March 1993.

[14] A. Narayana. "Proof of Church-Rosser Theorem in Calculus of Constructions." MS thesis, IIT Kanpur, April 1991.

[15] C. Paulin-Mohring. "Inductive Definitions in the system Coq: Rules and Properties." In M. Bezem and J. F. Groote, eds, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pp 328–345, Springer Verlag LNCS 664, April 1993.

[16] F. Pfenning. "A Proof of the Church-Rosser Theorem and its Representation in a Logical Framework." Technical Report CMU-CS-92-186, Carnegie Mellon University, Sept. 1992.

[17] P. Rudnicki. “An Overview of the MIZAR project.” Proceedings of the 1992 Workshop on Types for Proofs and Programs, Eds. B. Nordström, K. Petersson and G. Plotkin. Available by anonymous ftp from animal.cs.chalmers.se.

[18] N. Shankar. “A mechanical proof of the Church-Rosser Theorem.” Technical Report 45, Institute for Computing Science, The University of Texas at Austin, March 1985.