Specifications, Algorithms,

Axiomatisations and Proofs

Commented Case Studies
In the Coq Proof Assistant

Gérard Huet

Summer School on Logic of Computation

Marktoberdorf, August 1995

2$^{\text{nd}}$ Printing September 18$^{\text{th}}$, 1995

Contents

Chapter 1

Introduction

Coq is a Proof Assistant for a Logical Framework known as the Calculus of Inductive Constructions. It allows the interactive construction of formal proofs, and also the manipulation of functional programs consistently with their specifications. It may be obtained by anonymous FTP from site `ftp.inria.fr`, directory `INRIA/coq/V5.10`. We shall not discuss here in detail how to make proofs in Coq, and refer the interested reader to the Coq Tutorial, in the documentation section of the standard distribution. We shall concentrate in the following lessons on axiomatisations and specifications in the Gallina specification language.

1.1 An overview of the specification language Gallina

A formal development in Gallina consists in a sequence of *declarations* and *definitions*.

1.1.1 Declarations

A declaration associates a *name* with a *specification*. A name corresponds roughly to an identifier in a programming language, i.e. to a string of letters, digits, and a few ASCII symbols like underscore (`_`) and prime (`'`), starting with a letter. We use case distinction, so that the names `A` and `a` are distinct. Certain strings are reserved as key-words of Coq, and thus are forbidden as user identifiers.

A specification is a formal expression which classifies the notion which is being declared. There are basically three kinds of specifications: *logical propositions*, *mathematical collections*, and *abstract types*. They are classified by the three basic sorts of the system, called respectively `Prop`, `Set`, and `Type`, which are themselves atomic abstract types.

Every valid expression e in Gallina is associated with a specification, itself a valid expression, called its *type* $\tau(e)$. We write $e : E$ for the judgement that e is of type E. You may request Coq to return to you the type of a valid expression by using the command `Check`:

```
Coq < Check O.
O
     : nat
```

Thus we know that the identifier `O` (the name 'O', not to be confused with the numeral '0' which is not a proper identifier!) is known in the current context, and that its type is the specification `nat`. This specification is itself classified as a mathematical collection, as we may readily check:

```
Coq < Check nat.
nat
     : Set
```

The specification `Set` is an abstract type, one of the basic sorts of the Gallina language, whereas the notions *nat* and O are axiomatised notions which are defined in the arithmetic prelude, automatically loaded when executing the command '`Require Basis.`' in the initialisation file `.coqrc`.

With what we already know, we may now enter in the system a declaration, corresponding to the informal mathematics *let n be a natural number*:

```
Coq < Variable n:nat.
n is assumed
```

If we want to translate a more precise statement, such as *let n be a positive natural number*, we have to add another declaration, which will declare explicitly the hypothesis `Pos_n`, with specification the proper logical proposition:

```
Coq < Hypothesis Pos_n : (gt n O).
Pos_n is assumed
```

Indeed we may check that the relation `gt` is known with the right type in the current context:

```
Coq < Check gt.
gt
     : nat->nat->Prop
```

which tells us that `gt` is a function expecting two arguments of type `nat` in order to build a logical proposition. What happens here is similar to what we are used to in a functional programming language: we may compose the (specification) type `nat` with the (abstract) type `Prop` of logical propositions through the arrow function constructor, in order to get a functional type `nat->Prop`:

```
Coq < Check nat->Prop.
nat->Prop
     : Type
```

which may be composed again with `nat` in order to obtain the type `nat->nat->Prop` of binary relations over natural numbers. Actually `nat->nat->Prop` is an abbreviation for `nat->(nat->Prop)`.

Functional notions may be composed in the usual way. An expression f of type $A \rightarrow B$ may be applied to an expression e of type A in order to form the expression $(f\ e)$ of type B. Here we get that the expression `(gt n)` is well-formed of type `nat->Prop`, and thus that the expression `(gt n O)`, which abbreviates `((gt n) O)`, is a well-formed proposition.

```
Coq < Check (gt n O).
(gt n O)
     : Prop
```

1.1.2 Definitions

The initial prelude `Basis` contains a few arithmetic definitions: `nat` is defined as a mathematical collection (type `Set`), constants `O`, `S`, `plus`, are defined as objects of types respectively `nat`, `nat->nat`, and `nat->nat->nat`. You may introduce new definitions, which link a name to a well-typed value. For instance, we may introduce the constant `one` as being defined to be equal to the successor of zero:

```
Coq < Definition one := (S O).
one is defined
```

We may optionally indicate the required type:

```
Coq < Definition two : nat := (S one).
two is defined
```

Actually `Coq` allows several possible syntaxes:

```
Coq < Definition three := (S two) : nat.
three is defined
```

Here is a way to define the doubling function, which expects an argument `m` of type `nat` in order to build its result as `(plus m m)`:

```
Coq < Definition double := [m:nat](plus m m).
double is defined
```

The abstraction brackets are explained as follows. The expression `[x:A]e` is well formed of type `A->B` in a context whenever the expression `e` is well-formed of type `B` in the given context to which we add the declaration that `x` is of type `A`. Here x is a bound, or dummy variable in the expression `[x:A]e`. For instance we could as well have defined `double` as `[n:nat](plus n n)`.

Bound (local) variables and free (global) variables may be mixed. For instance, we may define the function which adds the constant `n` to its argument as

```
Coq < Definition add_n := [m:nat](plus m n).
add_n is defined
```

However, note that here we may not rename the formal argument m into n without capturing the free occurrence of n, and thus changing the meaning of the defined notion.

Binding operations are well known for instance in logic, where they are called quantifiers. Thus we may universally quantify a proposition such as $m > 0$ in order to get a universal proposition $\forall m \cdot m > 0$. Indeed this operator is available in `Coq`, with the following syntax: `(m:nat)(gt m O)`. Similarly to the case of the functional abstraction binding, we are obliged to declare explicitly the type of the quantified variable. We check:

```
Coq < Check (m:nat)(gt m O).
(m:nat)(gt m O)
     : Prop
```

1.1.3 Inductive Definitions

Inductive sets are defined by their constructors. For instance, here is teh definition of natural numbers in unary notation, as defined in Coq's arithmetic prelude:

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

This defines together an inductive set, bound to name `nat`, its two constructors, bound to names `O` and `S`, an induction principle `nat_ind`:

```
nat_ind : (P:nat->Prop)(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

and a recursion principle:

```
nat_rec : (P:nat->Set)(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

So far, this is similar to the user-defined recursive types one finds in programming languages such as ML. The main difference is that a positivity condition in the definition of inductive type in Coq forbids non-well-founded recursive definitions such as :

```
Inductive danger : Set := Quote : (danger -> danger) -> danger.
```

Such a recursive type would allow the definition of elements of type `danger` whose evaluation may not terminate. By contrast, in Coq all typed constructions evaluate to a finite canonical form, solely composed of constructors of the corresponding type.

It is also possible to define inductively predicates, relations, etc, by giving their defining clauses. For instance, here is a typical inductively defined relation `leq`

```
Inductive leq : nat -> nat -> Prop :=
      leq_O : (n:nat)(leq O n)
    | leq_S : (n,m:nat)(leq n m)->(leq (S n) (S m)).
```

It is also possible to define inductive relations in a context of parameters. For instance, the standard inequality relation $\leq$ over natural numbers is not defined in the standard prelude of Coq as `leq` above, but rather as `le` below, axiomatised as an inductively defined predicate in `m`, the first argument of `le` being a parameter:

```
Inductive le [n:nat] : nat -> Prop :=
      le_n : (le n n)
    | le_S : (m:nat)(le n m)->(le n (S m)).
```

Such a definition may be thought of as a (typed) Prolog specification. However, we have more than just the defining clauses, since we get automatically an induction principle as follows:

```
le_ind : (n:nat)(P:nat->Prop)
         (P n) -> ((m:nat)(le n m)->(P m)->(P (S m)))
               -> (m:nat)(le n m)->(P m)
```

In other words, the predicate `(le n)` is the smallest one which is true of `n` and which is closed by successor. We get logically more than just the logical consequences of the two defining clauses, we get their inductive completion ("Clark's completion", in logical programming parlance). We may show for instance that `(le (S n) (S m))` implies `(le n m)`.

1.1.4 Recursive Definitions

We may also write recursive definitions in a functional style. For instance, we may define integer addition by recursion on its first argument:

```
Recursive Definition plus : nat -> nat -> nat :=
   O     m => m
 | (S n) m => (S (plus n m)).
```

which corresponds to the usual recursion equations for addition. This definition adds to the global context two lemmas entering the recursion equations as equalities:

```
plus_eq1 : (m:nat)(plus O m)=m
plus_eq2 : (n:nat)(m:nat)(plus (S n) m)=(S (plus n m))
```

The standard logical connectives, equality, and existential quantification, are definable as inductive notions. Indeed, the standard prelude of the system declares these basic notions, with their usual syntax. For instance, `P /\ Q` (resp. `P \/ Q`) expresses the conjunction (resp. disjunction) of propositions P and Q.

The term `n=m` expresses equality of expressions n and m, provided they are declared in the current context with the same type, say `nat`. This is actually shorthand for `(eq nat n m)`, where eq is the inductively-defined polymorphic equality relation. In this case the unique constructor is reflexivity of equality, and the induction principle corresponds to the replacement of equals by equals. Such internal codings may be ignored in first approximation. Remark that the `nat` argument is automatically synthesised by `Coq`'s type checker. There is a systematic construction to allow such implicit arguments, which will be demonstrated in Chapter 4.

Thus, a typical mathematical axiomatisation in Coq will consist of a mixture of (higher-order) predicate calculus, non-deterministic specifications in the style of logic-programming, and recursive definitions in the style of functional-programming. Type-checking is done modulo definitional equality, which corresponds to replacing a defined constant by its definition, parameter substitution in functional applications (i.e. β-reduction in the underlying λ-calculus), and recursion unfolding (i.e. applying simplifications such as $(plus\ O\ m) \Rightarrow m$).

This blend of high-level notations leads to a smooth style of axiomatisation, relatively easy to master by anyone with a minimal training in formal methods, without requiring a complete understanding of the logical framework of the Calculus of Inductive Constructions.

We shall see in the rest of the course a few typical examples of `Coq` axiomatisations.

Chapter 2

Shuffling cards

2.1 Introduction

The first case study concerns a non-trivial property of binary sequences, inspired from a card trick due to Norman Gilbreath[29, 27, 28, 12]. The trick is based on a mathematical theorem concerning the shuffles of alternated binary words. A complete formalisation of the problem is given below.

2.2 Words

We first axiomatize binary words, an inductive structure isomorphic to Boolean lists:

```
Inductive word : Set := empty : word
                      | bit : bool -> word -> word.
```

2.2.1 Word concatenation and singleton

Word concatenation may be given a logical definition, *à la Prolog*:

```
Inductive conc : word -> word -> word -> Prop :=
  conc_empty : (v:word)(conc empty v v)
| conc_bit   : (u,v,w:word)(b:bool)(conc u v w)->(conc (bit b u) v (bit b w)).
```

Alternatively, it can be declared as a recursive function:

```
Recursive Definition append : word -> word -> word :=
    empty     v => v
  | (bit b w) v => (bit b (append w v)).
```

Such a definition is actually syntactic sugar for a combination of the primitive constructs of fixpoints and pattern-matching:

```
Fixpoint append [u:word] : word -> word :=
    [v:word]<word>Case u of
    (* empty *)   v
    (* bit b w *) [b:bool][w:word](bit b (append w v)) end.
```

It is easy to relate the two definitions:

```
Lemma conc_append : (u,v,w:word)(conc u v w) -> w=(append u v).
```

and to prove associativity of append:

```
Lemma assoc_append :
      (u,v,w:word)(append u (append v w))=(append (append u v) w).
```

We define the singleton list (single b):

```
Definition single := [b:bool](bit b empty).
```

2.2.2 Word length parities

We define parities odd and even by a mutual inductive definition:

```
Mutual Inductive odd : word -> Prop :=
  even_odd : (w:word)(even w)->(b:bool)(odd (bit b w))

            with even : word -> Prop :=
  even_empty : (even empty)
| odd_even : (w:word)(odd w)->(b:bool)(even (bit b w)).
```

We show a few easy lemmas on parities:

```
Lemma not_odd_empty : ~(odd empty).

Lemma inv_odd : (w:word)(b:bool)(odd (bit b w))->(even w).

Lemma inv_even : (w:word)(b:bool)(even (bit b w))->(odd w).

Lemma odd_or_even : (w:word)(odd w) \/ (even w).

Lemma not_odd_and_even : (w:word)(odd w)->(even w)->False.

Lemma odd_even_conc : (u,v,w:word)(conc u v w) ->
           (odd w) /\
                  ( (odd u) /\ (even v)
                 \/ (even u) /\ (odd v))
        \/ (even w) /\
                  ( (odd u) /\ (odd v)
                 \/ (even u) /\ (even v)).

Lemma even_conc : (u,v,w:word)(conc u v w) -> (even w) ->
                    (odd u) /\ (odd v)
                 \/ (even u) /\ (even v).
```

2.3 Alternating words

2.3.1 Alternate

A word w is *alternate* if for some bit b, w is of the form [b $\sim b$ b $\sim b$...]. We write (alt b w).

```
Inductive alt : bool -> word -> Prop :=
  alt_empty   : (b:bool)(alt b empty)
| alt_bit     : (b:bool)(w:word)(alt (neg b) w)->(alt b (bit b w)).
```

Let us show a few lemmas on alternation:

```
Lemma alt_neg_intro : (b,b':bool)(w:word)(alt b (bit b' w))->(alt (neg b) w).

Lemma alt_neg_elim : (b,b':bool)(w:word)(alt (neg b) (bit b' w))->(alt b w).

Lemma alt_eq : (b,b':bool)(w:word)(alt b (bit b' w))->b=b'.

Lemma alt_back :
      (b,b':bool)(w:word)(alt b (bit b' w))->(b=b' /\ (alt (neg b) w)).
```

We end with the existence property: w is *alternate* if for some bool b we have ($alt\ b\ w$). We use directly an inductive definition, rather than an explicit existential quantification, which is itself defined inductively anyway.

```
Inductive alternate [w:word] : Prop :=
  alter : (b:bool)(alt b w)->(alternate w).
```

2.3.2 Subwords of alternate words

Let us show that subwords of alternate words are alternate:

```
Lemma alt_conc_l : (u,v,w:word)(conc u v w)->(b:bool)(alt b w)->(alt b u).

Lemma alt_conc_r : (u,v,w:word)(conc u v w) -> (b:bool)(alt b w) ->
     (odd u) /\ (alt (neg b) v)
  \/ (even u) /\ (alt b v).

Lemma alt_conc : (u,v,w:word)(conc u v w) -> (alternate w) ->
                 (alternate u) /\ (alternate v).
```

2.4 Paired words

2.4.1 paired

A word w is said to be *paired* if it is of the form: [b_1 $\sim b_1$ b_2 $\sim b_2$ ... b_n $\sim b_n$].

```
Inductive paired : word -> Prop :=
  paired_empty : (paired empty)
| paired_bit : (w:word)(paired w)->(b:bool)(paired (bit (neg b) (bit b w))).
```

Here again we have a non-linear clause. We shall give several variants of “paired”. A paired word must be even. For odd words, we give below variants `paired_odd_l` and `paired_odd_r`.

$$(paired\_odd\_l\ b\ w) \Leftrightarrow w = [b\ b_1\ \ \sim b_1\ b_2\ \ \sim b_2\ ...\ b_n\ \ \sim b_n].$$

```
Definition paired_odd_l := [b:bool][w:word](paired (bit (neg b) w)).

Lemma paired_odd_l_intro :
      (b:bool)(w:word)(paired w)->(paired_odd_l b (bit b w)).

Lemma paired_odd_l_elim :
      (b:bool)(w:word)(paired_odd_l (neg b) w)->(paired (bit b w)).
```

Similarly:

$$(paired\_odd\_r\ b\ w) \Leftrightarrow w = [b_1\ \ \sim b_1\ b_2\ \ \sim b_2\ ...\ b_n\ \ \sim b_n\ \ \sim b].$$

```
Definition paired_odd_r := [b:bool][w:word](paired (append w (single b))).
```

An even word is *paired rotated* iff rotating it by one bit makes it paired:

$$(paired\_rot\ b\ w) \Leftrightarrow w = [b\ b2\ \ \sim b2\ ...\ bn\ \ \sim bn\ \ \sim b].$$

```
Inductive paired_rot : bool -> word -> Prop :=
  paired_rot_empty : (b:bool)(paired_rot b empty)
| paired_rot_bit : (b:bool)(w:word)(paired_odd_r b w)


Lemma paired_odd_r_from_rot :
      (w:word)(b:bool)(paired_rot b w)->(paired_odd_r b (bit (neg b) w)).
```

Finally, a word is said to be *paired between* if it is obtained by prefixing and suffixing a paired word with the same bit b:

$$(paired\_bet\ b\ w) \Leftrightarrow w = [b\ b_1\ \ \sim b_1\ b_2\ \ \sim b_2\ ...\ b_n\ \ \sim b_n\ b].$$

```
Inductive paired_bet [b:bool] :  word -> Prop :=
  paired_bet_bit : (w:word)(paired_odd_r (neg b) w)->(paired_bet b (bit b w)).

Lemma paired_odd_r_from_bet :
      (b:bool)(w:word)(paired_bet (neg b) w)->(paired_odd_r b (bit b w)).
```

2.4.2 rotate

We end this section by the definition of `rotate` and the proof of the claim concerning `paired_rot`:

```
Recursive Definition rotate : word -> word :=
    empty     => empty
  | (bit b w) => (append w (single b)).

Lemma paired_rotate : (w:word)(b:bool)(paired_rot b w)->(paired (rotate w)).
```

2.4.3 Opposite words

Two words are said to be *opposite* if they start with different bits:

```
Inductive opposite : word -> word -> Prop :=
  opp : (u,v:word)(b:bool)(opposite (bit b u) (bit (neg b) v)).

Lemma not_opp_empty_r : (u:word)~(opposite u empty).

Lemma not_opp_empty_l : (u:word)~(opposite empty u).

Lemma not_opp_same : (u,v:word)(b:bool)~(opposite (bit b u) (bit b v)).

Lemma alt_neg_opp : (u,v:word)(b:bool)(odd u)->(alt b u)->
                (odd v)->(alt (neg b) v)->(opposite u v).

Lemma alt_not_opp : (u,v:word)(b:bool)(alt b u)->(alt b v)->~(opposite u v).
```

2.5 Shuffle

2.5.1 Random Shuffle

Here we come to our main notion: `(shuffle u v w)`, meaning word w may be obtained by shuffling words u and v. We deal here with a truly non-deterministic specification.

```
Inductive shuffle : word -> word -> word -> Prop :=
  shuffle_empty :    (shuffle empty empty empty)
| shuffle_bit_left : (u,v,w:word)(shuffle u v w) ->
                           (b:bool)(shuffle (bit b u) v (bit b w))
| shuffle_bit_right : (u,v,w:word)(shuffle u v w) ->
                           (b:bool)(shuffle u (bit b v) (bit b w)).
```

The shuffling lemma

This lemma is the main result of this chapter. It gives the inductive invariant associated with the shuffling of alternated words.

```
Lemma Shuffling : (u,v,w:word)(shuffle u v w) -> (b:bool)(alt b u) ->
  ( (odd u) /\ ( (odd v)  /\ ((alt (neg b) v) -> (paired w))
                          /\ ((alt b v)       -> (paired_bet b w))
              \/ (even v) /\ ((alt b v)       -> (paired_odd_l b w))
                          /\ ((alt (neg b) v) -> (paired_odd_r (neg b) w)))
 \/ (even u) /\ ( (odd v) /\ ((alt (neg b) v) -> (paired_odd_r b w))
                          /\ ((alt b v)       -> (paired_odd_l b w))
              \/ (even v) /\ ((alt b v)       -> (paired_rot b w))
                          /\ ((alt (neg b) v) -> (paired w)))).
```

2.6 Gilbreath

The `Shuffling` lemma has the following corollary:

```
Theorem Gilbreath : (x:word)(even x)
  -> (alternate x)
  -> (u,v:word)(conc u v x)
  -> (w:word)(shuffle u v w)
  -> if (opposite u v) then (paired w) else (paired (rotate w)).
```

This theorem about binary sequences is the basis of a card trick due to Gilbreath, as follows.

The boolean words abstract card decks, with true for red and false for black. Take an even deck x, arranged alternatively red, black, red, black, etc. Ask a spectator to cut the deck, into sub-decks u and v. Now shuffle u and v into a new deck w. When shuffling, note carefully whether u and v start with opposite colors or not. If they do, the resulting deck is composed of pairs red-black or black-red; otherwise, you get the property by first rotating the deck by one card. The trick is usually played by putting the deck behind your back after the shuffle, to perform "magic". The magic is either rotating or doing nothing. When showing the pairing property, say loudly "red black red black..." in order to confuse in the spectator's mind the weak *paired* property with the strong *alternate* one.

There is a variant. If the cut is favorable, that is if u and v are opposite, just go ahead showing the pairing, without the "magic part." If the spectator says that he understands the trick, show him the counter-example in the non-favorable case. Of course now you have to leave him puzzled, and refuse to redo the trick.

Chapter 3

Lambda terms

We shall present in this chapter an example of a mathematical development in the field of λ-calculus, with heavy use of recursion.

We shall assume a certain familiarity with the elementary notions of pure λ-calculus, such as α-conversion and β-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 λ-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 development is to propose a clear inductive definition of this notion, as a refinement of one step of parallel reduction. It is more intuitive 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 λ-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.

3.1 Lambda terms

We represent our λ-terms with de Bruijn's indexes[9]. 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 le_lt : (n,m:nat)(le n m)->{lt n m}+{n=m}.

Lemma compare : (n,m:nat){lt n m}+{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.

3.1.1 Abstract syntax

The abstract syntax of λ-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 lambda : Set :=
    Ref : nat -> lambda
  | Abs : lambda -> lambda
  | App : lambda -> lambda -> lambda.
```

3.1.2 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 k n)`, where the operation `lift_rec` is defined recursively as follows:

```
Definition relocate := [i,k,n:nat]<nat>Case (test k i) of
        (* k<=i *) [H:(le k i)] (plus n i)
        (* k>i  *) [H:(gt k i)] i end.

Recursive Definition lift_rec : lambda -> nat -> nat -> lambda :=
   (Ref i) k n   => (Ref (relocate i k n))
 | (Abs M) k n   => (Abs (lift_rec M (S k) n))
 | (App M N) k n => (App (lift_rec M k n) (lift_rec N k n)).

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

3.1.3 Substitution

```
Definition insert_Ref := [N:lambda][i,k:nat]<lambda>Case (compare k i) of
           [C:{(gt i k)}+{k=i}]<lambda>Case C of
              (* k<i *) [H:(lt k i)](Ref (pred i))
              (* k=i *) [H:k=i]     (lift k N) end
              (* k>i *) [H:(gt k i)](Ref i) end.

Recursive Definition subst_rec : lambda -> lambda -> nat -> lambda :=
   (Ref i) N k    => (insert_Ref N i k)
 | (Abs M) N k    => (Abs (subst_rec M N (S k)))
 | (App M M') N k => (App (subst_rec M N k) (subst_rec M' N k)).

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

Let us give a few examples. The concrete λ-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 α-conversion). Thus we do not have to burden ourselves from the start with an awkward quotient structure. Similarly, λ-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 λ-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)`.

3.2 Reduction

3.2.1 One-step β-reduction

We may now axiomatize one step of β-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 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:

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

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

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

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

3.2.2 β-reduction

We now define β-reduction `red` as the transitive closure of `red1`.

```
Inductive 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 β-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')).
```

3.2.3 β-conversion

Similarly, β-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 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 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).
```

3.2.4 Parallel β-reduction

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

```
Inductive 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 β-reduction `par_red` as the transitive closure of `par_red1`.

```
Inductive 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).
```

3.2.5 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).
```

3.2.6 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) -> (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[4, 5].

```
Definition strip := (x,y:lambda)(par_red x y) ->
                      (z:lambda)(par_red1 x z) ->
          (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).
```

3.3 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`.

3.3.1 Redexes as an enrichment of terms

```
Inductive redexes : Set :=
    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.

```
Recursive Definition mark : lambda -> redexes :=
    (Ref n)   => (Var n)
  | (Abs M)   => (Fun (mark M))
  | (App M N) => (Ap false (mark M)(mark N)).

Recursive Definition unmark : redexes -> lambda :=
    (Var n)    => (Ref n)
  | (Fun U)    => (Abs (unmark U))
  | (Ap b U V) => (App (unmark U)(unmark V)).

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

3.3.2 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 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>Case b of true b' end.

Inductive 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 bool_max_Sym : (b,b':bool)(bool_max b b')=(bool_max b' b).

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 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)(CVW:(comp V W))(comp U W).

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

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

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

3.3.3 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.

```
Recursive Definition regular : redexes -> Prop :=
   (Var n)                     => True
 | (Fun V)                     => (regular V)
 | (Ap true  (Var n)       W) => False
 | (Ap true  (Fun V1)      W) => (regular V1) /\ (regular W)
 | (Ap true  (Ap b V1 V2) W) => False
 | (Ap false V W)              => (regular V) /\ (regular W).

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

3.4 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.

3.4.1 Lifting

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

```
Recursive Definition lift_rec_r : redexes -> nat -> nat -> redexes :=
   (Var i) k n    => (Var (relocate i k n))
 | (Fun M) k n    => (Fun (lift_rec_r M (S k) n))
 | (Ap b M N) k n => (Ap b (lift_rec_r M k n) (lift_rec_r N k n)).

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

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

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

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

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

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

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

Lemma liftO : (U: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)->
    (lift_rec_r (lift_rec_r U i n) k p)=(lift_rec_r U i (plus p n)).

Lemma lift_rec_lift : (U:redexes)(n,p,k,i:nat)(le k n)->
```

```
     (lift_rec_r (lift_r n U) k p)=(lift_r (plus p n) U).
```

3.4.2 Substitution

Again we follow the substitution function for the lambda structure.

```
Definition insert_Var := [N:redexes][i,k:nat]<redexes>Case (compare k i) of
           [C:{(gt i k)}+{k=i}]<redexes>Case C of
              (* k<i *) [H:(lt k i)](Var (pred i))
              (* k=i *) [H:k=i]     (lift_r k N) end
              (* k>i *) [H:(gt k i)](Var i) end.

Recursive Definition subst_rec_r : redexes -> redexes -> nat -> redexes :=
   (Var i) N k      => (insert_Var N i k)
 | (Fun M) N k      => (Fun (subst_rec_r M N (S k)))
 | (Ap b M M') N k => (Ap b (subst_rec_r M N k) (subst_rec_r M' N k)).

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

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

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

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

3.4.3 The Substitution Lemma

```
Definition subst_r := [N,M:redexes](subst_rec_r M N O).

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

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

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

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

```
    (subst_rec_r (subst_rec_r V W (S (plus p n))) (subst_rec_r U W 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 :
  (U,V,W:redexes)(n:nat)(subst_rec_r (subst_r U V) W n) =
                         (subst_r (subst_rec_r U W n) (subst_rec_r V W (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.

3.4.4 Preservation Lemmas

We now show that substitution preserves compatibility and regularity.

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

3.5 Residuals

We develop a strengthening of parallel β-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 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') -> W'=W.
```

3.5.1 The Commutation Theorem

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

```
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 U1 V1 k)
              (subst_rec_r U2 V2 k) (subst_rec_r U3 V3 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 [36]. 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...

3.5.2 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.

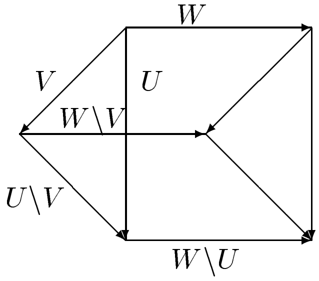
```
Lemma residuals_regular : (U,V,W:redexes)(residuals U V W) -> (regular V).

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

3.5.3 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[42] 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)).
```

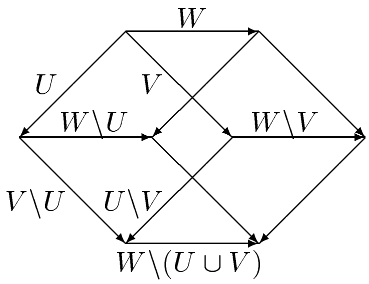
The careful reader may wonder about the slight asymmetry between the two sides of the equivalence. This is due to the fact that `(residuals U V W)` insures only that `V` is regular (this is lemma residuals_regular above), not `U` or `W`. In the informal statement of the theorem, we said "for every compatible 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.

3.5.4 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) ->
  (Ex [UV:redexes](Ex [VU: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 β-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[42], 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.

3.6 Confluence

3.6.1 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)(lift_r n (mark M))=(mark (lift n M)).

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

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

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

We now define reduction of a λ-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) -> N=P.
```

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

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

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

3.6.2 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 β-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).
```

3.6.3 The Church-Rosser Theorem

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

```
Theorem Church_Rosser : (M,N:lambda)(conv M N) ->
        (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.

Chapter 4

Categories

We now move to some more abstract mathematics, more specifically abstract algebra. This development is joint work with Amokrane Saïbi.

In this chapter we develop one possible axiomatisation of the notion of category by modeling objects as types and Hom-sets as Hom-setoids of arrows parameterized by their domain and codomain types. Thus we may quotient arrows, but not objects. We develop in this setting functors, as functions on objects, and extentional maps on arrows. We show that CAT is a category, and we do not need to distinguish to this effect "small" and "big" categories. We rather have implicitly categories as relatively small structured indexed by a universe. Thus we just need two instances of the same notion of category in order to define CAT.

We then construct the Functor Category, with the natural definition of natural transformations. We then show the Interchange Law, which exhibits the 2-categorical structure of the Functor Category.

This incursion in Constructive Category Theory shows that Type Theory is adequate to represent faithfully categorical reasoning. Three ingredients are essential: Σ-types, to represents structures, then dependent types, so that arrows are indexed with their domains and codomains, and finally a hierarchy of universes, in order to escape the foundational difficulties. Some amount of type reconstruction is necessary, in order to write equations between arrows without having to indicate their type other than at their binder, and notational abbreviations, allowing e.g. infix notation, are necessary to offer the formal mathematician a language close to the ordinary informal categorical notation.

4.1 Relations

We assume a number of basic constructions, which define quantifiers and equality at the level of sort `Type`. These definitions are included in the prelude module `Logic_Type` of the Coq system.

We start with a few standard definitions pertaining to binary relations.

```
Require Logic_Type.

Section Orderings.
   Variable U: Type.

   Definition Relation :=  U -> U -> Prop.
```

```
   Variable R: Relation.

   Definition Reflexive :=  (x: U) (R x x).

   Definition Transitive :=  (x,y,z: U) (R x y) -> (R y z) -> (R x z).

   Definition Symmetric :=  (x,y: U) (R x y) -> (R y x).

   Structure equivalence : Prop := {Prf_e_refl  : Reflexive;
                                    Prf_e_trans : Transitive;
                                    Prf_e_sym   : Symmetric}.

   Structure per : Prop := {Prf_pe_sym   : Symmetric;
                            Prf_pe_trans : Transitive}.

End Orderings.

Syntactic Definition Equivalence := equivalence | 1.
Syntactic Definition Partial_equivalence := per | 1.
```

The "Section" mechanism of Coq allows to parameterize the notions defined in the body of the section by the variables and axioms on which they depend. In our case, all the notions defined inside section `Orderings` are parameterized by parameters `U` and `R`. Thus, for instance, the definition of `Reflexive` becomes, upon closing of this section:

```
Definition Reflexive :=  [U:Type][R:(Relation U)](x: U) (R x x).
```

The combinator `Equivalence` is defined by the `Syntactic Definition` above. This is just a macro definition facility, which will in the rest of the session replace every occurrence of `Equivalence` by `(equivalence ?)`. These 'question marks arguments' will be automatically synthesized from the context. We use systematically this facility in the following development, with the convention that all the generic notions defined with macros have a name starting with an upper-case letter, and generate internally the same name with the corresponding lower-case letter, applied to question marks. The number of these question marks is indicated in the macro after the symbol `|`. In each context of use of such a macro, we shall expect the corresponding parameters to be derivable mechanically during type-checking.

4.2 Setoid

We now move to the development of "Setoids". Setoids are triples composed of a Type S, a relation R over S, and a proof that R is an equivalence relation over S. Thus a Setoid is a set considered as the quotient of a Type by a congruence. Setoids were first investigated by M. Hofmann in the framework of Martin-Löf's type theory[33]. This terminology is due to R. Burstall.

4.2.1 The Setoid structure

```
Structure Setoid : Type := {Carrier   : Type;
                            Equal     : (Relation Carrier);
                            Prf_equiv : (Equivalence Equal)}.
```

Let us understand what happens by type synthesis. The subformula `(Equivalence Equal)` gets replaced by `(equivalence ? Equal)`. Now the type checker instantiates the question mark as `Carrier`, since `Equal` is declared to be a relation over `Carrier`. This way we use type synthesis to elide information which is implicit from the context.

Remark that in Coq Σ-types (records) are not primitive, but are built as inductive types with one constructor. The macro `Structure` constructs the corresponding inductive type and defines the projection functions for destructuring a `Setoid` into its constituent fields. It defines also a constructor `Build_Setoid` to build a `Setoid` from its constituents. One can choose a different name for the constructor by putting the desired name before the opening brace of the Structure defining expression. Such specialized macros are user-definable in the same way as tactics.

In order to have a more pleasant notation, we first introduce a concrete syntax declaration allowing to parse `(Carrier A)` as `|A|`:

```
Grammar command command1 := [ "|" command0($s) "|"] -> [$0 = <<(Carrier $s)>>].
```

Such a `Grammar` declaration may be read as follows. A grammar production for the `command` language consists of two parts. The first part represents a production which is added to the corresponding non-terminal entry; here `command1`, receives the new production `"|" command0($s) "|"`. The second part is the semantic action which builds the corresponding abstract syntax tree when firing this production; here we indicate that we build an application of the constant `Carrier` to the result of parsing with entry `command0` what is enclosed in the vertical bars. The various entries `commandn` stratify the commands according to priority levels.

Given a Setoid `A`, `(Equal A)` is its associated relation. We give it the infix notation `=%S`, the parameter `A` being synthesised by type-checking. Since the symbol `=%S` is not predefined, we have to declare it as a new token (for the "extensible" lexer of Coq).

```
Token "=%S".
Grammar command command1 := [ command0($a) "=%S" command0($b) ] ->
                            [$0 = <<(Equal ? $a $b)>>].
```

Note that `=%S` is a generic Setoid equality, since the type of its elements may in general be inferred from the context, as we shall see immediately.

The last extracted field is the proof that the equality of a Setoid is an equivalence relation. Right after this equivalence proof, we give as corollaries reflexivity, symmetry and transitivity of equality. We get these proofs easily with the help of Coq's proof engine, driven by tactics. Here as in the rest of the document, we do not give the proof scripts, just the statements of lemmas.

```
Lemma Prf_refl : (A:Setoid)(Reflexive |A| (Equal A)).

Lemma Prf_sym : (A:Setoid)(Symmetric |A| (Equal A)).

Lemma Prf_trans : (A:Setoid)(Transitive |A| (Equal A)).
```

4.2.2 An example

As example of the preceeding notions, let us define the Setoid of natural numbers. The type of its elements cannot be directly the inductively defined `nat:Set`, but it is easy to define an isomorphic `Nat:Type`.

```
Inductive Nat : Type := Z : Nat | Suc : Nat -> Nat.

Definition Eq_Nat := [N1,N2:Nat] N1==N2.
```

The `==` symbol which appears in the body of the definition of `Eq_Nat`, is the standard polymorphic Leibniz equality defined in the `Logic_Type` module. In our case, it is instantiated over the Type `Nat`, inferred from the type of `N1`.

Right after this, we give the equivalence proof of `Eq_Nat` and build the setoid of natural numbers:

```
Lemma Eq_Nat_equiv : (Equivalence Eq_Nat).

Definition Set_of_nat := (Build_Setoid Nat Eq_Nat Eq_Nat_equiv).
```

4.2.3 Alternative: Partial Setoids

Alternatively, we could build Partial Setoids, where the equality equivalence is replaced by a weaker partial equivalence relation of coherence; total elements are defined as being coherent with themselves:

```
Structure PSetoid : Type := {PCarrier  : Type;
                             Coherence : (Relation PCarrier);
                             Prf_PER   : (Partial_equivalence Coherence)}.

Definition Total := [A:PSetoid][x:(PCarrier A)](Coherence ? x x).
```

4.2.4 The Setoid of Maps between two Setoids

We now define a Map between Setoid `A` and Setoid `B` as a function from `|A|` to `|B|` which respects equality. Remark the use of generic equality in `Map_law`.

```
Section maps.

Variables A,B: Setoid.

Definition map_law := [f:|A|->|B|](x,y:|A|) x =%S y -> (f x) =%S (f y).

Structure Map : Type := build_Map {ap   : |A|->|B|;
                                   Pres : (map_law ap)}.
```

A Map `m` over `A` and `B` is thus similar to a pair, packing a function `(ap A B m)` (of type `|A|->|B|`) with the proof `(Pres A B m)` that this function respects equality.

Two Maps `f` and `g` are defined to be equal iff they are extensionaly equal, i.e. $\forall x.f(x) = g(x)$:

```
Definition Ext := [f,g:Map](x:|A|) (ap f x) =%S (ap g x).

Lemma Ext_equiv : (Equivalence Ext).

Definition Map_setoid := (Build_Setoid Map Ext Ext_equiv).

End maps.
```

```
Syntactic Definition Map_law := map_law | 2.
Syntactic Definition Build_Map := build_Map | 2.
```

We write `f =%M g` for $f = g$.

```
Token "=%M".
Grammar command command2 := [ command1($f) "=%M" command2($g) ] ->
                            [$0 = <<(Ext ? ? $f $g)>>].

Grammar command command2 := [ command1($A) "=>" command2($B) ] ->
                            [$0 = <<(Map_setoid $A $B)>>].
```

This last command allows writing `A=>B`, with appropriate precedence level, for the Setoid of Maps between Setoids `A` and `B`.

We end this section by defining a generic `Ap`, denoting the application function associated with a Map, and a (curried) binary application `ap2`, useful for what follows.

```
Syntactic Definition Ap := ap | 2.

Definition ap2 := [A,B,C:Setoid][f:|(A=>(B=>C))|][a:|A|] (Ap (Ap f a)).

Syntactic Definition Ap2 := ap2 | 3.
```

4.3 Categories

4.3.1 The category structure

We now axiomatise a category as consisting of a Type of Objects and a family of Hom Setoids indexed by their domain and codomain types.

```
Section cat.

Variable Ob  : Type.
Variable Hom : Ob->Ob->Setoid.
```

The next component of a category is a composition operator, which for any Objects `a,b,c,` belongs to `(Hom a b) => ((Hom b c) => (Hom a c))`. We write this operator (parameters `a,b,c,` being implicit by type synthesis) as infix `o`.

```
Variable Op_comp : (a,b,c:Ob)|((Hom a b) => ((Hom b c) => (Hom a c)))|.

Definition Cat_comp := [a,b,c:Ob](Ap2 (Op_comp a b c)).

Grammar command command2 := [ command1($f) " o " command2($g) ] ->
                            [$0 = <<(Cat_comp ? ? ? $f $g)>>].
```

Composition is assumed to be associative:

```
Definition assoc_law := (a,b,c,d:Ob)(f:|(Hom a b)|)(g:|(Hom b c)|)(h:|(Hom c d)|)
                        (f o (g o h)) =%S ((f o g) o h).
```

The final component of a category is, for every object a, an arrow in (H a a) which is an identity for composition:

```
Variable id : (a:Ob)|(Hom a a)|.

Definition idl_law := (a,b:Ob)(f:|(Hom a b)|)((id a) o f) =%S f.

Definition idr_law := (a,b:Ob)(f:|(Hom b a)|)f =%S (f o (id ?)).

End cat.
```

We give generic notations for the various laws:

```
Syntactic Definition Assoc_law := assoc_law | 2.
Syntactic Definition Idl_law := idl_law | 2.
Syntactic Definition Idr_law := idr_law | 2.
```

We are now able to define synthetically a Category:

```
Structure Category : Type :=
 {Ob      : Type;
  hom     : Ob -> Ob -> Setoid;
  Op_comp : (a,b,c:Ob)|((hom a b) => ((hom b c) => (hom a c)))|;
  id      : (a:Ob)|(hom a a)|;
  Prf_ass : (Assoc_law Op_comp);
  Prf_idl : (Idl_law Op_comp id);
  Prf_idr : (Idr_law Op_comp id)}.
```

We successively define the projections which extract the various components of a category.

```
Syntactic Definition Hom := hom | 1.
Syntactic Definition Id := id | 1.

Definition Comp := [C:Category][a,b,c:(Ob C)](Cat_comp (Ob C) (hom C) (Op_comp C) a b c).

Grammar command command2 := [ command1($f) " o " command2($g) ] ->
                            [$0 = <<(Comp ? ? ? ? $f $g)>>].
```

Remark that we now use the infix notation o in the context of a local Category parameter C. It must be noticed that Grammar definitions inside Sections disappear when their section is closed. Thus the new rule giving syntax for Comp does not conflict with the previous one giving syntax for Cat_Comp.

Actually, a composition operator is nothing else than a binary function verifying the congruence laws for both arguments. Thus we provide a general method allowing the construction of a composition operator from such a function. We shall use systematically this tool in the following, for every category definition.

```
Section composition_to_operator.

Variable Ob       : Type.
```

```
Variable Hom      : Ob->Ob->Setoid.
Variable Comp_fun : (a,b,c:Ob)|(Hom a b)|->|(Hom b c)|->|(Hom a c)|.

Definition Congl_law := (a,b,c:Ob)(f,g:|(Hom b c)|)(h:|(Hom a b)|)
           f =%S g -> (Comp_fun a b c h f) =%S (Comp_fun a b c h g).

Definition Congr_law := (a,b,c:Ob)(f,g:|(Hom a b)|)(h:|(Hom b c)|)
           f =%S g -> (Comp_fun a b c f h) =%S (Comp_fun a b c g h).

Definition Cong_law := (a,b,c:Ob)(f,f':|(Hom a b)|)(g,g':|(Hom b c)|)
           f =%S f' -> g =%S g' -> (Comp_fun a b c f g) =%S (Comp_fun a b c f' g').

Hypothesis pcgl : Congl_law.
Hypothesis pcgr : Congr_law.

Variable a, b, c : Ob.

Lemma Comp1_map_law : (f:|(Hom a b)|)(Map_law (Comp_fun a b c f)).

Definition Comp1_map := [f:|(Hom a b)|](Build_Map (Comp_fun a b c f) (Comp1_map_law f)).

Lemma Comp_map_law : (map_law (Hom a b) (Hom b c)=>(Hom a c) Comp1_map).

Definition Build_Comp := (build_Map (Hom a b) (Hom b c)=>(Hom a c)
                                    Comp1_map Comp_map_law).

End composition_to_operator.
```

We now check that composition preserves the morphisms equalities, to the left and to the right, and prove as a corollary the congruence law for composition:

```
Lemma Prf_congl : (C:Category)(Congl_law ? ? (Comp C)).

Lemma Prf_congr : (C:Category)(Congr_law ? ? (Comp C)).

Lemma Prf_cong : (C:Category)(Cong_law ? ? (Comp C)).
```

4.3.2 Hom equality

We need for the following a technical definition: two arrows in $(Hom\ a\ b)$ of category C are equal iff the corresponding elements of the Setoid $(Hom\ a\ b)$ are equal. This is a typical example where Type Theory obliges us to make explicit an information which does not even come up in the standard mathematical discourse based on set theory. Of course we would like the standard "abus de notation" to be implemented in a more transparent way, through overloading or implicit coercions. We deal with this problem here by implicit synthesis of the category parameter and of the object parameters in order to write simply `f =%H g` for the equality of arrows `f` and `g`.

```
Inductive Equal_hom [C:Category;a,b:(Ob C);f:|(Hom a b)|] :
                    (c,d:(Ob C))|(Hom c d)| -> Prop :=
Build_Equal_hom : (g:|(Hom a b)|) f =%S g -> (Equal_hom C a b f a b g).
```

```
Token "=%H".
Grammar command command1 := [ command0($f) "=%H" command0($g) ] ->
                            [$0 = <<(Equal_hom ? ? ? $f ? ? $g)>>].
```

Here the reader may be puzzled at our seemingly too general type for arrow equality: the predicate `Equal_hom` takes as arguments a Category `C`, objects `a,b,c,d` of `C`, and arrows `(f:|(Hom a b)|)` and `(g:|(Hom c d)|)`. Since the only possible constructor for this equality is `Build_Equal_hom`, which requires the second arrow `g` to have the same type as the first one `f`, it might seem sufficient to restrict the type of `Equal_hom` accordingly. However, this generality is needed, because we want to be able to state the equality of two arrows whose respective domains are not definitionally equal, but will be equal for certain instanciations of parameters. For instance, later on, the problem will arise when defining functor equality: we want to be able to write $F(f) = G(f)$, which will force say $F(A)$ and $G(A)$ to be definitionally equal objects, but there is no way to specify F and G with type declarations such that $F(A) = G(A)$. This would necessitate an extension of type theory with definitional constraints, which could be problematic with respect to decidability of definitional equality. This extension is not really needed if one takes care to write dependent equalities with sufficiently general types.

```
Section equal_hom_equiv.

Variable C     : Category.
Variable a , b : (Ob C).
Variable f     : |(Hom a b)|.

Lemma Equal_hom_refl : f =%H f.

Variable c, d : (Ob C).
Variable g    : |(Hom c d)|.

Lemma Equal_hom_sym : f =%H g -> g =%H f.

Variable i, j : (Ob C).
Variable h    : |(Hom i j)|.

Lemma Equal_hom_trans : f =%H g -> g =%H h -> f =%H h.

End equal_hom_equiv.
```

4.3.3 Dual Categories

The dual category C° of a category C has the same objects as C. Its arrows however are the opposites of the arrows of C, i.e. $f^\circ : a \to b$ is a morphism of C° iff $f : b \to a$ is a morphism of C.

```
Structure darrow [C:Category;a,b:(Ob C)] : Type := build_Darrow {dmor : |(Hom b a)|}.
```

We may extract from a morphism `df` of C° a morphism of C, denoted `(Dmor df)`.

```
Syntactic Definition Darrow := darrow | 1.
```

```
Syntactic Definition Build_Darrow := build_Darrow | 3.
Syntactic Definition Dmor := dmor | 3.

Section Dcat.

Variable C :Category.

 Section dhom_def.

 Variable a, b : (Ob C).

 Definition Equal_Darrow := [df,dg:(Darrow a b)](Dmor df) =%S (Dmor dg).

 Lemma Equal_Darrow_equiv : (Equivalence Equal_Darrow).

 Definition DHom := (Build_Setoid (Darrow a b) Equal_Darrow Equal_Darrow_equiv).

 End dhom_def.
```

Composition is defined as expected: $f^\circ \circ^\circ g^\circ = (g \circ f)^\circ$. Identity arrows are invariant. We then check the category laws.

```
Definition Comp_Darrow := [a,b,c:(Ob C)][df:(Darrow a b)][dg:(Darrow b c)]
                          (Build_Darrow ((Dmor dg) o (Dmor df))).

Lemma Comp_dual_congl : (Congl_law (Ob C) DHom Comp_Darrow).

Lemma Comp_dual_congr : (Congr_law (Ob C) DHom Comp_Darrow).

Definition Comp_Dual := (Build_Comp (Ob C) DHom Comp_Darrow Comp_dual_congl
                                       Comp_dual_congr).

Lemma Assoc_Dual : (assoc_law (Ob C) DHom Comp_Dual).

Definition Id_Dual := [a:(Ob C)](Build_Darrow (Id a)).

Lemma Idl_Dual : (idl_law (Ob C) DHom Comp_Dual Id_Dual).

Lemma Idr_Dual : (idr_law (Ob C) DHom Comp_Dual Id_Dual).
```

We write `(Dual C)` for the dual category of `C`.

```
Definition Dual := (Build_Category (Ob C) DHom Comp_Dual Id_Dual
                                   Assoc_Dual Idl_Dual Idr_Dual).

End Dcat.
```

4.3.4 Category exercises

We define epics, monos, and isos. As an exercise, we show that two initial objects are isomorphic.

A morphism $f : a \to b$ is *epi* when for any two morphisms $g, h : b \to c$, the equality $f \circ g = f \circ h$ implies $g = h$.

```
Section cat_prop.

Variable C:Category.

 Section epic_monic_def.

 Variable a, b : (Ob C).

 Definition Epic_law := [f:|(Hom a b)|](c:(Ob C))(g,h:|(Hom b c)|)
                       (f o g) =%S (f o h) -> g =%S h.

 Structure isEpic [f:|(Hom a b)|] : Type := {Epic_l : (Epic_law f)}.
```

A morphism $f : b \to a$ is monic when for any two morphisms $g, h : c \to b$, the equality $g \circ f = h \circ f$ implies $g = h$.

```
 Definition Monic_law := [f:|(Hom b a)|](c:(Ob C))(g,h:|(Hom c b)|)
                        (g o f) =%S (h o f) -> g =%S h.

 Structure isMonic [f:|(Hom b a)|] : Type := {Monic_l : (Monic_law f)}.

 End epic_monic_def.
```

A morphism f is iso if there is a morphism $f^{-1}1 : b \to a$ whith $f^{-1} \circ f = Id_b$ and $f \circ f^{-1} = Id_a$.

```
 Section iso_def.

 Definition Iso_law := [a,b:(Ob C)][f:|(Hom a b)|][f1:|(Hom b a)|]
                      (f1 o f) =%S (Id b).

 Variable a, b : (Ob C).

 Structure isIso [f:|(Hom a b)|] : Type := {inv_iso : |(Hom b a)|;
                                            Idl_inv : (Iso_law ? ? f inv_iso);
                                            Idr_inv : (Iso_law ? ? inv_iso f)}.
```

We now say that two objects a and b are isomorphic ($a \cong b$) if they are connected by an iso arrow.

```
 Structure iso : Type := {Iso_mor   : |(Hom a b)|;
                         Prf_isIso : (isIso Iso_mor)}.

 End iso_def.
```

Now we say that object a is *initial* in Category C iff for any object b there exists a unique arrow in $(Hom\ a\ b)$.

```
Definition At_most_1mor := [a,b:(Ob C)](f,g:|(Hom a b)|) f =%S g.

Structure isInitial [a:(Ob C)] : Type := {morI    : (b:(Ob C))|(Hom a b)|;
                                          UniqueI : (b:(Ob C))(At_most_1mor a b) }.
```

Dually we define when an object b is *terminal* in Category C: for any object a there exists a unique arrow in $(Hom\ a\ b)$.

```
Structure isTerminal [b:(Ob C)] : Type := {morT    : (a:(Ob C))|(Hom a b)|;
                                          uniqueT : (a:(Ob C))(At_most_1mor a b)}.

End cat_prop.

Syntactic Definition IsEpic := isEpic | 3.
Syntactic Definition IsMonic := isMonic | 3.
Syntactic Definition IsIso := isIso | 3.
Syntactic Definition Inv_iso := inv_iso | 4.
Syntactic Definition Iso := iso | 1.
Syntactic Definition IsInitial := isInitial | 1.
Syntactic Definition MorI := morI | 2.
Syntactic Definition IsTerminal := isTerminal | 1.
Syntactic Definition MorT := morT | 2.
```

As an exercise we may prove easily that any two initial objects must be isomorphic:

```
Lemma I_unic : (C:Category)(I1,I2:(Ob C))(IsInitial I1) ->  (IsInitial I2) -> (Iso I1 I2).
```

We also prove that the property of being terminal is dual to that of being initial: an initial object in C is terminal in C°.

```
Lemma Initial_dual : (C:(Category))(a:(Ob C))(IsInitial a) -> (isTerminal (Dual C) a).
```

We remark that these properties have been defined at the `Type` level. They could all be defined at the level `Prop`, except that in the case of `IsIso`, we could not extract the field `inv_iso` of type `Type`.

4.3.5 The Category of Setoids

We now define the Category of Setoids with Maps as Homs. First we have to define composition and identity of Maps. The composition of two Maps is defined from the composition of their underlying functions; we have to check extensionality of the resulting function. We use the infix notation `o%M`.

```
Section mcomp.

Variable A, B, C : Setoid.
Variable f       : (Map A B).
Variable g       : (Map B C).

Definition Comp_arrow := [x:|A|](Ap g (Ap f x)).

Lemma Comp_arrow_map_law : (Map_law Comp_arrow).

Definition Comp_map := (Build_Map Comp_arrow Comp_arrow_map_law).

End mcomp.
```

```
Token "%M".
Grammar command command2 := [ command1($f) "o" "%M" command2($g) ] ->
                            [$0 = <<(Comp_map ? ? ?  $f $g)>>].
```

The operator `Map_comp` is just a function. We shall now "mapify" it, by proving that it is extensional in its two arguments, in order to get a Map composition operator.

```
Lemma Comp_map_congl : (Congl_law Setoid Map_setoid Comp_map).

Lemma Comp_map_congr : (Congr_law Setoid Map_setoid Comp_map).

Definition Comp_SET := (Build_Comp Setoid Map_setoid Comp_map
                                      Comp_map_congl Comp_map_congr).
```

After checking the associativity of our composition operation, we define the identity Map from the identity function $\lambda x.x$, checking other category laws.

```
Lemma Assoc_SET : (assoc_law Setoid Map_setoid Comp_SET).

Section id_set_def.

Variable A : Setoid.

Definition Id_SET_arrow := [x:|A|]x.

Lemma Id_SET_map_law : (Map_law Id_SET_arrow).

Definition Id_SET := (Build_Map Id_SET_arrow Id_SET_map_law).

End id_set_def.

Lemma Idl_SET : (idl_law Setoid Map_setoid Comp_SET Id_SET).

Lemma Idr_SET : (idr_law Setoid Map_setoid Comp_SET Id_SET).
```

Now we have all the ingredients to form the Category of Setoids SET.

```
Definition SET := (Build_Category Setoid Map_setoid Comp_SET Id_SET Assoc_SET
                                  Idl_SET Idr_SET).
```

4.4 Functors

4.4.1 Definition of Functor

Functors between categories C and D are defined in the usual way, with two components, a function from the objects of C to the objects of D, and a Map from Hom-sets of C to Hom-sets of D. Remark how type theory expresses in a natural way the type constraints of these notions, without arbitrary codings.

```
Section Functors.

Variable C, D : Category.
Variable FOb  : (Ob C) -> (Ob D).
Variable FMap : (a,b:(Ob C))(Map (Hom a b) (Hom (FOb a) (FOb b))).
```

Functors must preserve the Category structure, and thus verify the two laws: $F(f \circ g) = F(f) \circ F(g)$ and $F(id_a) = id_{F(a)}$.

```
Definition fcomp_law := (a,b,c:(Ob C))(f:|(Hom a b)|)(g:|(Hom b c)|)
           (Ap (FMap a c) (f o g)) =%S ((Ap (FMap a b) f) o (Ap (FMap b c) g)).

Definition fid_law := (a:(Ob C))(Ap (FMap a a) (Id a)) =%S  (Id (FOb a)).

End Functors.

Syntactic Definition Fcomp_law := fcomp_law | 2.
Syntactic Definition Fid_law := fid_law | 2.

Structure Functor [C,D:Category] : Type :=
 {fOb           : (Ob C) -> (Ob D);
  FMap          : (a,b:(Ob C))(Map (Hom a b) (Hom (fOb a) (fOb b)));
  Prf_Fcomp_law : (Fcomp_law fOb FMap);
  Prf_Fid_law   : (Fid_law  fOb FMap)}.
```

As usual, we define some syntactical abbreviations. Thus $F(a)$ will be written `(FOb F a)` and $F(f)$ will be written `(FMor F f)`.

```
Definition fMor := [C,D:Category][a,b:(Ob C)][F:(Functor C D)][f:|(Hom a b)|]
                   (Ap (FMap ? ? F a b) f).

Syntactic Definition FOb := fOb | 2.
Syntactic Definition FMor := fMor | 4.

Lemma Prf_FMap_law : (C,D:Category)(F:(Functor C D))(a,b:(Ob C))
                     (Map_law (Ap (FMap ? ? F a b))).
```

We now define the Setoid of Functors. The equality of Functors is extensional equality on their morphism function component, written as infix `=%F` with appropriate type synthesis:

```
Definition Equal_Functor := [C,D:Category][F,G:(Functor C D)](a,b:(Ob C))(f:|(Hom a b)|)
                            (FMor F f) =%H (FMor G f).

Token "=%F".
Grammar command command1 := [ command0($F1) "=%F" command0($F2) ] ->
                            [$0 = <<(Equal_Functor ? ? $F1 $F2)>>].

Lemma Equal_Functor_equiv : (C,D:Category)(Equivalence (Equal_Functor C D)).
```

We now have all the ingredients to form the Functor Setoid.

```
Definition Functor_Setoid:= [C,D:Category](Build_Setoid (Functor C D) (Equal_Functor C D)
                                                         (Equal_Functor_equiv C D)).
```

4.4.2 Hom Functors

We give in this section an example of functor construction, with the family of Hom-Functors.

Let C be a category and a an object of C. The functor $Hom(a, -) : C \to SET$ is defined by:

- for every object b of C, $Hom(a, -)(b) = (Hom\ a\ b)$

```
Section funset.

Variable C : Category.
Variable a : (Ob C).

Definition FunSET_ob := [b:(Ob C)](Hom a b).
```

- for every $f : b \to c$, $Hom(a, -)(f) : (Hom\ a\ b) \to (Hom\ a\ c)$ is a Map, mapping morphism $g : a \to b$ of C to morphism $g \circ f$.

```
Section funset_map_def.

Variable b, c : (Ob C).

 Section funset_mor_def.

 Variable f : |(Hom b c)|.

 Definition FunSET_mor1 := [g:|(Hom a b)|] (g o f).

 Lemma FunSET_map_law1 : (Map_law FunSET_mor1).

 Definition FunSET_mor := (Build_Map FunSET_mor1 FunSET_map_law1).

 End funset_mor_def.

Lemma FunSET_map_law : (map_law (Hom b c) (hom SET (FunSET_ob b) (FunSET_ob c))
                                FunSET_mor).

Definition FunSET_map := (build_Map (Hom b c) (hom SET (FunSET_ob b) (FunSET_ob c))
                                    FunSET_mor FunSET_map_law).

End funset_map_def.
```

We check the functorial properties for $Hom(a, -)$ and define it, with notation `(FunSET a)`.

```
Lemma Fun_comp_law : (fcomp_law C SET FunSET_ob FunSET_map).

Lemma Fun_id_law : (fid_law C SET FunSET_ob FunSET_map).

Definition funSET := (Build_Functor C SET FunSET_ob FunSET_map Fun_comp_law Fun_id_law).

End funset.

Syntactic Definition FunSET := funSET | 1.
```

4.4.3 The Category of Categories

In this section we now reflect the theory upon itself: Categories may form the type of a category of Categories CAT, the arrows being Functors. All is really needed is to define Functor composition and Identity, and to prove a few easy lemmas exhibiting the Category structure of CAT.

The first step consists in defining the composition of two functors. We compose functors $G : C \to D$ and $H : D \to E$ to form a functor $F \circ G : C \to E$, by composing separately their object functions and their morphism maps. We write `o%F` for this functor composition.

```
Section Comp_F.

Variable C, D, E : Category.
Variable G       : (Functor C D).
Variable H       : (Functor D E).

Definition comp_FOb := [a:(Ob C)](FOb H (FOb G a)).

 Section comp_functor_map.

 Variable a, b : (Ob C).

 Definition comp_FMor := [f:|(Hom a b)|](FMor H (FMor G f)).

 Lemma Comp_FMap_law : (Map_law comp_FMor).

 Definition Comp_FMap := (Build_Map comp_FMor Comp_FMap_law).

 End comp_functor_map.

Lemma Comp_Functor_comp_law : (Fcomp_law comp_FOb Comp_FMap).

Lemma Comp_Functor_id_law : (Fid_law comp_FOb Comp_FMap).

Definition Comp_Functor := (Build_Functor C E comp_FOb Comp_FMap
                                          Comp_Functor_comp_law Comp_Functor_id_law).

End Comp_F.

Syntactic Definition Comp_FOb := comp_FOb | 3.
Syntactic Definition Comp_FMor := comp_FMor | 3.

Token "%F".
Grammar command command2 := [ command1($F) "o" "%F" command2($G) ] ->
                            [$0 = <<(Comp_Functor ? ? ? $F $G)>>].
```

As before, we construct a composition operator after checking the Congruence laws.

```
Lemma Comp_Functor_congl : (Congl_law Category Functor_setoid Comp_Functor).

Lemma Comp_Functor_congr : (Congr_law Category Functor_setoid Comp_Functor).
```

```
Definition Comp_CAT := (Build_Comp Category Functor_setoid Comp_Functor
                                   Comp_Functor_congl Comp_Functor_congr).

Lemma Assoc_CAT : (assoc_law Category Functor_setoid Comp_CAT).
```

For every category C, we construct the identity functor id_C from the identity function on objects and the identity map on morphisms.

```
Section idCat.

Variable C : Category.

Definition Id_CAT_ob := [a:(Ob C)]a.

Definition Id_CAT_map := [a,b:(Ob C)](Build_Map ([f:|(Hom a b)|] f)
                                          ([x,y:?][p:(x =%S y)]p)).

Lemma Id_CAT_comp_law : (Fcomp_law Id_CAT_ob Id_CAT_map).

Lemma Id_CAT_id_law : (Fid_law Id_CAT_ob Id_CAT_map).

Definition Id_CAT := (Build_Functor C C Id_CAT_ob Id_CAT_map
                                    Id_CAT_comp_law Id_CAT_id_law).

End idCat.

Lemma Idl_CAT : (idl_law Category Functor_setoid Comp_CAT Id_CAT).

Lemma Idr_CAT : (idr_law Category Functor_setoid Comp_CAT Id_CAT).
```

We now have all the ingredients to recognize in CAT the structure of a Category. All we need to do is to take a second copy of the notion of Category, called Category'. The implicit universe adjustment mechanism will make sure that its Type refers to a bigger universe.

```
Structure Category' : Type :=
 {Ob'      : Type;
  hom'     : Ob' -> Ob' -> Setoid;
  opcomp'  : (a,b,c:Ob')|((hom' a b) => ((hom' b c) => (hom' a c)))|;
  id'      : (a:Ob')|(hom' a a)|;
  Prf_ass' : (Assoc_law opcomp');
  Prf_idl' : (Idl_law opcomp' id');
  Prf_idr' : (Idr_law opcomp' id')}.

Definition CAT : Category' := (Build_Category' Category Functor_setoid Comp_CAT
                                    Id_CAT Assoc_CAT Idl_CAT Idr_CAT).
```

Note that here we make an essential use of the universes hierarchy: There is not a unique CAT, there is a family of CAT_i, and each CAT_i is a $Category_j$ for $i < j$. Thus we do not have "small" and "large" categories, but "relatively small" categories. Thus the construction of CAT above is consistent with the analysis by Coquand[16] of paradoxes related to the category of categories.

It is to be remarked that this example justifies the mechanism called "universe polymorphism" defined by Harper and Pollack[32]. That is, with universe polymorphism, we could directly define CAT as a Category, without having to make an explicit copy of the notion, the copying being done implicitly for each occurrence of the name Category. Coq does not implement universe polymorphism at present, because this mechanism is rather costly in space, and seldom used in practice.

4.4.4 Functor exercises

It is easy to check that a functor preserves the property of being iso, i.e. $a \cong b \Longrightarrow F(a) \cong F(b)$.

```
Section functor_prop.

Variable C, D : Category.
Variable F    : (Functor C D).

Lemma Functor_preserves_iso : (a,b:(Ob C))(Iso a b) -> (Iso (FOb F a) (FOb F b)).
```

A functor $F : C \to D$ is said to be *faithful* if for any pair a, b of objects of C, and any pair $f, g : a \to b$ of morphisms, we have $F(f) = F(g)$ only if $f = g$.

```
Definition Faithful_law := (a,b:(Ob C))(f,g:|(Hom a b)|)
                           ((FMor F f) =%S (FMor F g)) -> (f =%S g).

Structure isFaithful : Type := {faithful : Faithful_law}.
```

A functor is said to be *full* if, for any pair a, b of objects of C, and any morphism $h : F(a) \to F(b)$, there exists a morphism $f : a \to b$ such that $h = F(f)$. According to the axiom of choice, there exists a function H such that for every morphism $h : F(a) \to F(b)$, $H(h)$ corresponds to f, i.e. $h = F(H(h))$.

```
Definition Full_law := [H:(a,b:(Ob C))|(Hom (FOb F a) (FOb F b))| -> |(Hom a b)|]
                       (a,b:(Ob C))(h:|(Hom (FOb F a) (FOb F b))|)
                       h =%S (FMor F (H a b h)).

Structure isFull : Type :=
 {Full_mor     : (a,b:(Ob C))|(Hom (FOb F a) (FOb F b))| -> |(Hom a b)|;
  Prf_full_law : (Full_law Full_mor) }.

End functor_prop.

Syntactic Definition IsFaithful := isFaithful | 2.
Syntactic Definition IsFull := isFull | 2.
```

These two properties are closed by composition:

```
Section comp_functor_prop.

Variable C, D, E : Category.
Variable F       : (Functor C D).
Variable G       : (Functor D E).
```

```
Lemma IsFaithful_comp : (IsFaithful F) -> (IsFaithful G) -> (IsFaithful (F o%F G)).

Lemma IsFull_comp : (IsFull F) -> (IsFull G) -> (IsFull (F o%F G)).

End comp_functor_prop.
```

4.5 The Functor Category

The type of Functors from Category C to Category D admits a Category structure. The corresponding arrows are called Natural Transformations.

4.5.1 Natural Transformations

We now define Natural Transformations between two Functors F and G from C to D. A Natural Transformation T from F to G maps an object a of Category C to an arrow T_a from object $F(a)$ to object $G(a)$ in Category D such that the following naturality law holds: $F(f) \circ T_b = T_a \circ G(f)$.

$$
\begin{tikzcd}
a \arrow[d, "f"'] & F(a) \arrow[r, "T_a"] \arrow[d, "F(f)"'] & G(a) \arrow[d, "G(f)"] \\
b & F(b) \arrow[r, "T_b"'] & G(b)
\end{tikzcd}
$$

Note that Natural Transformations are defined as functions, not as Maps, since objects are Types and not necessarily Setoids.

```
Section nat_transf.

Variable C, D : Category.
Variable F, G : (Functor C D).

Variable T : (a:(Ob C))|(Hom (FOb F a) (FOb G a))|.

Definition nt_law := (a,b:(Ob C))(f:|(Hom a b)|)
                     ((FMor F f) o (T b)) =%S ((T a) o (FMor G f)).

End nat_transf.

Syntactic Definition NT_law := nt_law | 2.

Structure nt [C,D:Category;F,G:(Functor C D)] : Type := build_NT
 {apNT       : (a:(Ob C))|(Hom (FOb F a) (FOb G a))|;
  Prf_NT_law : (NT_law ? ? apNT)}.

Syntactic Definition Build_NT := build_NT | 2.
Syntactic Definition NT := nt | 2.
```

```
Syntactic Definition ApNT := apNT | 4.
```

We now define $\{F \Rightarrow G\}$, the Natural Transformations Setoid between Functors F and G. Equality of natural transformations is also extensional. Thus, two natural transformations T and T' are said to be equal whenever their components are equal for an arbitrary object: $\forall a.T_a = T'_a$. As previously, we write `=%NT` for this equality.

```
Section setoid_nt.

Variable C, D : Category.
Variable F, G : (Functor C D).

Definition Equal_NT := [T,T':(NT F G)](a:(Ob C))(ApNT T a) =%S (ApNT T' a).

Lemma Equal_NT_equiv : (Equivalence Equal_NT).

Definition NT_setoid := (Build_Setoid (NT F G) Equal_NT Equal_NT_equiv).

End setoid_nt.

Token "=%NT".
Grammar command command1 := [ command0($T1) "=%NT" command0($T2) ] ->
                            [$0 = <<(Equal_NT ? ? ? ? $T1 $T2)>>].

Grammar command command1 := [ "{" command0($F) "=>" command0($G) "}" ] ->
                            [$0 = <<(NT_setoid ? ? $F $G)>>].
```

4.5.2 An example of Natural Transformation

Let C be any category, and $f^\circ : b \to a$ be a morphism of C°. We define a natural transformation $H(f^\circ) : Hom(b,-) \to Hom(a,-)$, called the *Yoneda map*, as follows.

$$
\begin{tikzcd}
b \arrow[d, "f^\circ"'] & & C \arrow[r, "{Hom(b,-)}"] \arrow[d, phantom, ""] & SET \arrow[d, phantom, ""] \\
a & & C \arrow[r, "{Hom(a,-)}"'] & SET
\end{tikzcd}
\qquad
H(f^\circ) \downarrow
$$

For every object c of C, $H(f^\circ)(c)$ is a Map mapping a morphism $h : b \to c$ of C to the morphism $f \circ h$.

```
Section funset_nt.

Variable C    : Category.
Variable b, a : (Ob C).
Variable fo   : |(hom (Dual C) b a)|.
```

```
Section nth_map_def.

 Variable c : (Ob C).

 Definition NTH_arrow := [h:|(Hom b c)|] (Dmor fo) o h.

 Lemma NTH_map_law : (Map_law NTH_arrow).

 Definition NTH_map := (Build_Map NTH_arrow NTH_map_law).

End nth_map_def.
```

We check the naturality of this map and define the natural transformation $H(f^\circ)(c)$, written as (NTH fo).

```
Lemma NTH_nt_law : (NT_law (FunSET b) (FunSET a) NTH_map).

Definition ntH := (Build_NT (FunSET b) (FunSET a) NTH_map NTH_nt_law).

End funset_nt.

Syntactic Definition NTH := ntH | 3.
```

4.5.3 Constructing the Category of Functors

We now have all the tools to define the category of Functors from C to D. Objects are Functors, arrows are corresponding Natural Transformations Setoids.

We define the composition of two natural transformations T and T' as $(T \circ_v T')_a = T_a \circ T'_a$. The v subscript stands for "vertical", since we shall define later another "horizontal" composition.

```
Section cat_functor.

Variable C, D : Category.

 Section compnt.

 Variable F, G, H : (Functor C D).
 Variable T       : (NT F G).
 Variable T'      : (NT G H).

 Definition Comp_tau := [a:(Ob C)](ApNT T a) o (ApNT T' a).

 Lemma Comp_tau_nt_law : (NT_law ? ? [a:(Ob C)](Comp_tau a)).

 Definition CompV_NT := (Build_NT ? ? Comp_tau Comp_tau_nt_law).

 End compnt.

Lemma CompV_NT_congl : (Congl_law (Functor C D) (NT_setoid C D) CompV_NT).
```

```
Lemma CompV_NT_congr : (Congr_law (Functor C D) (NT_setoid C D) CompV_NT).

Definition Comp_CatFunct := (Build_Comp (Functor C D) (NT_setoid C D)
                                        CompV_NT CompV_NT_congl CompV_NT_congr).

Lemma Assoc_CatFunct : (assoc_law (Functor C D) (NT_setoid C D) Comp_CatFunct).
```

To every functor F, we associate an identity natural transformation id_F defined as $\lambda a.Id_{F(a)}$:

```
 Section id_catfunct_def.

 Variable F : (Functor C D).

 Definition Id_CatFunct_tau := [a:(Ob C)](Id (FOb F a)).

 Lemma Id_CatFunct_nt_law: (NT_law ? ? Id_CatFunct_tau).

 Definition Id_CatFunct := (Build_NT ? ? Id_CatFunct_tau Id_CatFunct_nt_law).

 End id_catfunct_def.

Lemma Idl_CatFunct : (idl_law (Functor C D) (NT_setoid C D) Comp_CatFunct Id_CatFunct).

Lemma Idr_CatFunct : (idr_law (Functor C D) (NT_setoid C D) Comp_CatFunct Id_CatFunct).
```

Having checked that we have all categorical properties, we may now define the functor category.

```
Definition CatFunct := (Build_Category (Functor C D) (NT_setoid C D) Comp_CatFunct
                             Id_CatFunct Assoc_CatFunct Idl_CatFunct Idr_CatFunct).

End cat_functor.

Token "%NTv".
Grammar command command2 := [ command1($T1) "o" "%NTv" command2($T2) ] ->
                            [$0 = <<(CompV_NT ? ? ? ? ? $T1 $T2)>>].
```

4.6 The Interchange Law

In order to put to the test our categorical constructions, we prove the *interchange law*. This is one of the laws of 2-categories (categories whose arrows have themselves a category structure). This notion is used in theoretical computer science for the study of programming languages semantics and type theory.

Let A, B and C be categories, $F, G : A \to B$ and $F', G' : B \to C$ be functors. We define the *horizontal composition* of natural transformations $T : F \to G$ and $T' : F' \to G'$ as $(T \circ_h T')_a = T'_{F(a)} \circ G'(T_a)$. We check that $T \circ_h T'$ is indeed a natural transformation from $F \circ F'$ to $G \circ G'$.

$$
\begin{tikzcd}
A \arrow[r, "F"] \arrow[d, "T"'] & B \arrow[r, "F'"] \arrow[d, "T'"'] & C & A \arrow[r, "F \circ F'"] \arrow[d, "T \circ_h T'"] & C \\
A \arrow[r, "G"'] & B \arrow[r, "G'"'] & C & A \arrow[r, "G \circ G'"'] & C
\end{tikzcd}
$$

```
Section horz_comp.

Variable A, B, C : Category.
Variable F, G    : (Functor A B).
Variable F', G'  : (Functor B C).
Variable T       : (NT F G).
Variable T'      : (NT F' G').

Definition Ast : (a:(Ob A))|(Hom (FOb (F o%F F') a) (FOb (G o%F G') a))| :=
                 [a:(Ob A)](ApNT T' (FOb F a)) o (FMor G' (ApNT T a)).
```

In order to prove the naturality of $T \circ_h T'$, we use the equality:

$$(T \circ_h T')_a = T'_{F(a)} \circ G'(T_a) = F'(T_a) \circ T'_{G(a)}$$

which follows from the naturality diagram of T' for the morphism T_a:

$$
\begin{tikzcd}
F(a) \arrow[d, "T_a"'] & F'(F(a)) \arrow[r, "T'_{F(a)}"] \arrow[d, "F'(T_a)"'] & G'(F(a)) \arrow[d, "G'(T_a)"] \\
G(a) & F'(G(a)) \arrow[r, "T'_{G(a)}"'] & G'(G(a))
\end{tikzcd}
$$

```
Lemma Ast_eq : (a:(Ob A)) ((FMor F' (ApNT T a)) o (ApNT T' (FOb G a))) =%S
                          ((ApNT T' (FOb F a)) o (FMor G' (ApNT T a))).

Lemma Ast_nt_law : (NT_law (F o%F F') (G o%F G') Ast).

Definition CompH_NT := (Build_NT (F o%F F') (G o%F G') Ast Ast_nt_law).

End horz_comp.
```

We shall write `o%NTh` for horizontal composition.

```
Grammar command command2 := [ command1($T1) "o" "%NTh" command2($T2) ] ->
                            [$0 = <<(CompH_NT ? ? ? ? ? ? ? ? $T1 $T2)>>].
```

We shall now verify an important algebraic property, the *Interchange Law*, which links horizontal and vertical composition.

Let A, B, C be categories, $F, G, H : A \to B$, $F', G', H' : B \to C$ be functors, and $T : F \to G$, $T' : F' \to G'$, $U : G \to H$, $U' : G' \to H'$ be natural transformations.

$$\begin{tikzcd}
A \arrow[rr, "F"] & {} \arrow[d, "T"'] & B \arrow[rr, "F'"] & {} \arrow[d, "T'"] & C \\
A \arrow[rr, "G"] & {} \arrow[d, "U"'] & B \arrow[rr, "G'"] & {} \arrow[d, "U'"] & C \\
A \arrow[rr, "H"'] & {} & B \arrow[rr, "H'"'] & {} & C
\end{tikzcd}$$

Interchange Law:

$$(T \circ_h T') \circ_v (U \circ_h U') = (T \circ_v U) \circ_h (T' \circ_v U')$$

```
Section interchangelaw.

Variable A, B, C    : Category.
Variable F, G, H    : (Functor A B).
Variable F', G', H' : (Functor B C).
Variable T          : (NT F G).
Variable T'         : (NT F' G').
Variable U          : (NT G H).
Variable U'         : (NT G' H').

Lemma InterChange_law : ((T o%NTh T') o%NTv (U o%NTh U')) =%NT
                        ((T o%NTv U) o%NTh (T' o%NTv U')).

End interchangelaw.
```

We end this section by showing that the horizontal composition of natural transformations is associative. Here lurks a small difficulty.

Given Categories A, B, C, Functors F, G from A to B and Functors F', G' from B to C, the horizontal composition of Natural Transformation T from F to G and Natural Transformation T' from F' to G' yields a Natural Transformation $T \circ T'$ from $F \circ F'$ to $G \circ G'$. Expressing the associativity of this horizontal composition operation would need to identify, as types, say $(F \circ F') \circ F''$ and $F \circ (F' \circ F'')$. But here we run into a problem. Although these two terms are equal in the sense of Functor equality, they are *not* definitionally equal, and thus we are unable to even write the statement of associativity of horizontal composition: it does not typecheck.

In order to circumvent this problem, we need to define a less constrained equality `=%NTH` between natural transformations as follows.

```
Definition EqualH_NT := [C,D:Category][F,G:(Functor C D)]
                        [F',G':(Functor C D)][T:(NT F G)][T':(NT F' G')]
                        (a:(Ob C)) (ApNT T a) =%H (ApNT T' a).

Token "=%NTH".
Grammar command command1 := [ command0($T) "=%NTH" command0($T') ] ->
                            [$0 = <<(EqualH_NT ? ? ? ? ? ? $T $T')>>].

Section assoc_horz_comp.

Variable A,B,C,D : Category.
Variable F,G     : (Functor A B).
Variable F',G'   : (Functor B C).
Variable F'',G'' : (Functor C D).
Variable T       : (NT F G).
Variable T'      : (NT F' G').
Variable T''     : (NT F'' G'').

Lemma CompH_NT_assoc : ((T o%NTh T') o%NTh T'') =%NTH (T o%NTh (T' o%NTh T'')).

End assoc_horz_comp.
```

4.7 Conclusion

The development shown in this paper is but a tiny initial fragment of the theory of categories. However, it is quite promising, in that the power of dependent types and inductive types (or at least Σ-types) is put to full use; note in particular the dependent equality between morphisms of possibly non-convertible types.

We also point out that the syntactic facilities offered by the new version V5.10 of Coq are a first step in the right direction: the user may define his own notations through extensible grammars, types which are implicitly known by dependencies are synthesised automatically, and the macro-definition facility (so-called `Syntactic Definition`) allows a certain amount of high-level notation.

In order to show how crucial these tools are, we give below the statement of the interchange law without syntactic abbreviations:

```
(Equal_NT A C (Comp_Functor A B C F F') (Comp_Functor A B C H H')
          (CompV_NT A C (Comp_Functor A B C F F') (Comp_Functor A B C G G')
                    (Comp_Functor A B C H H') (CompH_NT A B C F G F' G' T T')
                     (CompH_NT A B C G H G' H' U U'))
          (CompH_NT A B C F H F' H' (CompV_NT A B F G H T U)
                    (CompV_NT B C F' G' H' T' U')))
```

We are thus closing the gap with standard mathematical practice, although some supplementary overloading mechanisms are obviously still lacking in order to implement the usual “abus de notation”.

This logical reconstruction of the basics of category theory follows initial attempts by R. Dyckhoff[25] in Martin-Löf type theory. It shows that intentional type theory is sufficient for developing this kind of mathematics, and we may thus hope to develop more sophisticated notions such as adjunction, which so far have been formally pursued only in extensional type theory[2].

Burstall and Rydeheard[51] have implemented a substantial number of concepts and constructions of category theory in SML (an ML dialect). The essential difference with our approach is that they do not include in their formalisation the properties (such as equations deriving from diagrams) of their categorical constructions. Thus they cannot mechanically check that their constructions have the intended properties. This exhibits the essential expressivity increase from a functional programming language with simple types to a type theory with dependent types, whose Curry-Howard interpretation includes the verification of predicate calculus conditions.

The above axiomatisation may indeed be pursued to include a significant segment of category theory. Thus A. Saïbi shows in [52] how to define adjunction and limits, develops standard constructions such as defining limits from equalisers and products, and shows the existence of left adjunct functors under the conditions of Freyd's theorem.

Chapter 5

Sorting

In this chapter we shall present a concrete algorithm specification for heapsort, and show how to extract from the proof a running ML program verifying this specification. This shows that it is possible to use Coq to build certified and relatively efficient programs, extracting them from the proofs of their specifications. We finally show how to reverse this program synthesis paradigm into a more usual program proving paradigm.

5.1 Lists

5.1.1 Genericity

In order to make our development parametric over the type of the list elements, we import a shared module Generic which just contains the declaration of an abstract Set parameter A, which denotes the concrete type of the list elements.

```
Parameter A:Set.
```

5.1.2 List structure and operators

Let us briefly review the List_of_A module of the Coq library. This module declares

```
Inductive list : Set :=
    nil : list
  | cons : A -> list -> list.
```

The List module defines several operations, a typical one being append

```
Recursive Definition append : list -> list -> list :=
    nil        v => v
  | (cons a u) v => (cons a (append u v)).
```

Here are a few typical lemmas concerning append:

```
Lemma append_nil_unit : (l:list)l=(append l nil).

Lemma append_assoc :
  (l,m,n: list)(append (append l m) n)=(append l (append m n)).
```

5.2 Multisets

In order to compare the contents of lists and other data structures storing elements of A, we need to axiomatize multisets. We shall model here a multiset as a function counting multiplicities of such elements.

5.2.1 Equality

In order to identify multiple instances of an element, we need to postulate a decidable equality equ on A, as follows.

```
Parameter equ : A -> A -> Prop.
Axiom equ_decide : (x,y:A){(equ x y)}+{~(equ x y)}.
```

5.2.2 Multiset structure

A multiset packages a function counting multiplicities inside a Bag.

```
Inductive multiset : Set :=
      Bag : (A->nat) -> multiset.

Definition EmptyBag := (Bag [a:A]O).
Definition SingletonBag := [a:A](Bag [a':A]
              <nat>Case (equ_decide a a') of
                   [h:(equ a a')]  (S O)
                   [h:~(equ a a')] O end).

Definition multiplicity : multiset -> A -> nat :=
  [m:multiset][a:A]<nat> Case m of [f:A->nat](f a) end.
```

We now define multiset equality as extensional equality of the corresponding multiplicities.

```
Definition meq := [m1,m2:multiset]
    (a:A)(multiplicity m1 a)=(multiplicity m2 a).
```

We now proceed to show that meq is an equivalence relation:

```
Lemma meq_refl : (x:multiset)(meq x x).

Lemma meq_trans : (x,y,z:multiset)(meq x y)->(meq y z)->(meq x z).

Lemma meq_sym : (x,y:multiset)(meq x y)->(meq y x).
```

5.2.3 Multiset union

We now axiomatise union of multisets, using addition of their multiplicities. We then proceed to show that this operation munion is commutative, associative, and compatible with meq. This uses the corresponding properties of integer addition, from the system module Plus.

```
Definition munion := [m1,m2:multiset]
    (Bag [a:A](plus (multiplicity m1 a)(multiplicity m2 a))).

Lemma munion_empty_left :
     (x:multiset)(meq x (munion EmptyBag x)).

Lemma munion_empty_right :
     (x:multiset)(meq x (munion x EmptyBag)).

Lemma munion_comm : (x,y:multiset)(meq (munion x y) (munion y x)).
Hint munion_comm.

Lemma munion_ass :
      (x,y,z:multiset)(meq (munion (munion x y) z) (munion x (munion y z))).

Lemma meq_left :  (x,y,z:multiset)(meq x y)->(meq (munion x z) (munion y z)).

Lemma meq_right : (x,y,z:multiset)(meq x y)->(meq (munion z x) (munion z y)).
```

5.2.4 List contents

As an application, we define recursively the contents of a list as a multiset:

```
Recursive Definition list_contents : list -> multiset :=
   nil        => EmptyBag
 | (cons a l) => (munion (SingletonBag a) (list_contents l)).
```

5.2.5 Permutations

Sorting algorithms use sometimes tricky permutation lemmas. It would be foolish to prove these lemmas concretely on our arithmetic implementation of multisets, since they are general identities of the free commutative associative monoid. We thus develop once and for all a module `Permut` where such properties are proved by abstract algebra. Here we just postulate an abstract Set `U`, given with a commutative-associative operator `op` and a congruence `cong`.

```
Section Axiomatisation.

Variable U: Set.

Variable op: U -> U -> U.

Variable cong : U -> U -> Prop.

Hypothesis op_comm : (x,y:U)(cong (op x y) (op y x)).
Hypothesis op_ass  : (x,y,z:U)(cong (op (op x y) z) (op x (op y z))).
```

```
Hypothesis cong_left :  (x,y,z:U)(cong x y)->(cong (op x z) (op y z)).
Hypothesis cong_right : (x,y,z:U)(cong x y)->(cong (op z x) (op z y)).
Hypothesis cong_trans : (x,y,z:U)(cong x y)->(cong y z)->(cong x z).
Hypothesis cong_sym : (x,y:U)(cong x y)->(cong y x).

Lemma cong_congr :
 (x,y,z,t:U)(cong x y)->(cong z t)->(cong (op x z) (op y t)).

Lemma comm_right : (x,y,z:U)(cong (op x (op y z)) (op x (op z y))).

Lemma comm_left : (x,y,z:U)(cong (op (op x y) z) (op (op y x) z)).

Lemma perm_right : (x,y,z:U)(cong (op (op x y) z) (op (op x z) y)).

Lemma perm_left : (x,y,z:U)(cong (op x (op y z)) (op y (op x z))).

Lemma op_rotate : (x,y,z,t:U)(cong (op x (op y z)) (op z (op x y))).

Lemma twist : (x,y,z,t:U)
   (cong (op x (op (op y z) t)) (op (op y (op x t)) z)).

End Axiomatisation.
```

Note the tricky `twist`, which we shall indeed need for heapsort later. It would be nice to be able to use the technology of associative-commutative rewriting in order to prove all these lemmas on a call-by need fashion, using a tactic implementing the decision procedure for the free abelian monoid as canonical AC rewriting. Such a tactic, parametric on `op` and `congr`, would be exported by the `Permut` module.

5.2.6 Multiset permutations

Using the `Permut` module, we get by instanciation the corresponding multiset equalities, and a few more:

```
Lemma munion_rotate :
   (x,y,z:multiset)(meq (munion x (munion y z)) (munion z (munion x y))).

Lemma meq_congr : (x,y,z,t:multiset)(meq x y)->(meq z t)->
                  (meq (munion x z) (munion y t)).

Lemma munion_perm_left :
   (x,y,z:multiset)(meq (munion x (munion y z)) (munion y (munion x z))).

Lemma multiset_twist1 : (x,y,z,t:multiset)
   (meq (munion x (munion (munion y z) t)) (munion (munion y (munion x t)) z)).
```

```
Lemma multiset_twist2 : (x,y,z,t:multiset)
   (meq (munion x (munion (munion y z) t)) (munion (munion y (munion x z)) t)).

Lemma heapsort_twist1 : (x,y,z,t,u:multiset) (meq u (munion y z)) ->
   (meq (munion x (munion u t)) (munion (munion y (munion x t)) z)).

Lemma heapsort_twist2 : (x,y,z,t,u:multiset) (meq u (munion y z)) ->
   (meq (munion x (munion u t)) (munion (munion y (munion x z)) t)).
```

5.3 Treesort specification

We now consider the specification of treesort, and first of all the data-structure of heap trees.

5.3.1 Trees

We first define the datatype of binary trees labeled with elements of `A`:

```
Inductive Tree : Set :=
      Tree_Leaf : Tree
    | Tree_Node : A -> Tree -> Tree -> Tree.
```

In order to define sorting on `A`, we must postulate that this `Set` is equipped with a decidable total ordering `inf`:

```
Parameter inf : A -> A -> Prop.

Axiom inf_total : (x,y:A){(inf x y)}+{(inf y x)}.
Axiom inf_tran : (x,y,z:A)(inf x y)->(inf y z)->(inf x z).
Axiom inf_refl : (x:A)(inf x x).
```

5.3.2 Heaps

We now define a relation `Tree_Lower` between an `A`-element `a` and a tree `t`, which is true either if `t` is a leaf, or if it carries a value `b` with `(inf a b)`:

```
Recursive Definition Tree_Lower : A -> Tree -> Prop :=
   a Tree_Leaf           => True
 | a (Tree_node b T1 T2) => (inf a b).
```

We now define inductively the predicate `is_heap`:

```
Inductive is_heap : Tree -> Prop :=
     nil_is_heap  : (is_heap Tree_Leaf)
   | node_is_heap : (a:A)(T1:Tree)(T2:Tree)
                    (Tree_Lower a T1) ->
                    (Tree_Lower a T2) ->
                    (is_heap T1) -> (is_heap T2) ->
                    (is_heap (Tree_Node a T1 T2)).
```

To this inductively defined predicate corresponds an induction principle, as usual automatically synthesised by Coq:

```
is_heap_ind : (P:Tree->Prop)
   (P Tree_Leaf) ->
   ((a:A)(T1,T2:Tree)(Tree_Lower a T1) -> (Tree_Lower a T2) ->
         (is_heap T1) -> (P T1) ->
         (is_heap T2) -> (P T2) -> (P (Tree_Node a T1 T2))) ->
   (t:Tree)(is_heap t) -> (P t)
```

It is also possible to define the corresponding recursion operator:

```
Lemma is_heap_rec : (P:Tree->Set)
   (P Tree_Leaf) ->
   ((a:A)(T1,T2:Tree)(Tree_Lower a T1) -> (Tree_Lower a T2) ->
         (is_heap T1) -> (P T1) ->
         (is_heap T2) -> (P T2) -> (P (Tree_Node a T1 T2))) ->
   (t:Tree)(is_heap t) -> (P t)
```

5.3.3 Contents of a tree

We define the contents of a tree recursively as a multiset:

```
Recursive Definition contents : Tree -> multiset :=
   Tree_Leaf            => EmptyBag
 | (Tree_Node a t1 t2) => (munion (contents t1)
                           (munion (contents t2) (SingletonBag a))).
```

Two trees are equivalent if their contents are equal as multisets:

```
Definition equiv_Tree := [t1,t2:Tree](meq (contents t1) (contents t2)).
```

5.3.4 Heaps

We first define an auxiliary predicate: element a is lower than a Tree T if T is a Leaf, or T is a Node holding element $b > a$:

```
Recursive Definition Tree_lower : A -> Tree -> Prop :=
   a Tree_Leaf            => True
 | a (Tree_Node b T1 T2) => (inf a b).
```

A heap is now defined as a tree such that all its branches are increasing:

```
Inductive is_heap : Tree -> Prop :=
      nil_is_heap : (is_heap Tree_Leaf)
    | node_is_heap : (a:A)(T1,T2:Tree)
                     (Tree_lower a T1) ->
                     (Tree_lower a T2) ->
                     (is_heap T1) -> (is_heap T2) ->
                     (is_heap (Tree_Node a T1 T2)).
```

We are now ready to write the specification of insertion of an element in a heap:

```
Inductive insert_spec [a:A; T:Tree] : Set :=
 insert_exist : (T1:Tree)(is_heap T1) ->
                (meq (contents T1) (munion (contents T) (SingletonBag a))) ->
                ((b:A)(inf b a)->(Tree_lower b T)->(Tree_lower b T1)) ->
                (insert_spec a T).
```

Insertion in a heap, according to the above specification, may be "programmed" in the proof of the following lemma:

```
Lemma insert : (T:Tree)(is_heap T)-> (a:A)(insert_spec a T).
```

5.3.5 Sorting

Let us now define the property of being sorted for a list.

```
Inductive list_lower [a:A] : list -> Prop :=
      nil_low : (list_lower a nil)
    | cons_low : (b:A)(l:list)(inf a b)->(list_lower a (cons b l)).

Inductive sort : list -> Prop :=
      nil_sort : (sort nil)
    | cons_sort : (a:A)(l:list)(sort l)->(list_lower a l)->(sort (cons a l)).
```

5.3.6 Merging

We also define merging of two sorted lists as a sorted list

```
Inductive merge_lem [l1:list;l2:list] : Set :=
  merge_exist : (l:list)(sort l) ->
    (meq (list_contents l)
         (munion (list_contents l1) (list_contents l2))) ->
    ((a:A)(list_lower a l1)->(list_lower a l2)->(list_lower a l)) ->
    (merge_lem l1 l2).

Lemma merge : (l1:list)(sort l1)->(l2:list)(sort l2)->(merge_spec l1 l2).
```

5.3.7 Conversions

We now coerce a list into a tree, and conversely flatten a heap into a sorted list:

```
Inductive build_heap [l:list] : Set :=
   heap_exist : (T:Tree)(is_heap T) ->
                (meq (list_contents l)(contents T)) ->
                (build_heap l).

Lemma list_to_heap : (l:list)(build_heap l).
```

```
Inductive flat_spec [T:Tree] : Set :=
 flat_exist : (l:list)(sort l) ->
              ((a:A)(Tree_lower a T)->(list_lower a l)) ->
              (meq (contents T)(list_contents l)) ->
              (flat_spec T).

Lemma heap_to_list : (T:Tree)(is_heap T)->(flat_spec T).
```

We now have all the ingredients for treesort:

```
Theorem treesort : (l:list)
   {m:list | (sort m) & (meq (list_contents l) (list_contents m))}.
```

5.4 Extracting a treesort ML program

Now that the hard proving work is over, we may leave programming to the machiene; i.e., we may extract mechanically a computer program from the above constructive proof, in a completely automatic fashion. All we have to provide are semantic attachments to our parameters `A` and `inf_total`, all other axioms and parameters being logical notions (over *Prop*) without constructive contents (over Set).

The programming language used for our extracted programs is Caml Light, the version of ML which is the actual implementation language of Coq. We thus specify these semantic attachments as Caml Light constructs, and specify to use Caml Light's `bool` and `int` types as the implementations of `bool` and `A` respectively: as follows:

```
ML Import int : Set.
Link A := int.
ML Inductive bool [ true false ] ==
         Inductive BOOL : Set := TRUE  : BOOL
                               | FALSE : BOOL.
ML Import lt_int : int->int->BOOL.
Link inf_total :=
         [x,y:int]<sumbool>Case (lt_int x y) of
                            (* TRUE *) left
                           (* FALSE *) right
                           end.
```

Then we extract the Caml Light program:

```
Write Caml File "treesort" [ treesort ].
```

Let us look at the ML program automatically produced by Coq:

```
type sumbool = left
             | right
;;
```

```
type A == int;;

type list = nil
          | cons of A * list
;;

type Tree = Tree_Leaf
          | Tree_Node of A * Tree * Tree
;;

let inf_total x y =
  match lt_int x y with
    true -> left
  | false -> right
;;

let insert T =
  let rec F = function
    Tree_Leaf -> (fun a -> Tree_Node(a,Tree_Leaf,Tree_Leaf))
  | Tree_Node(a,t0,t1) ->
      (fun a0 -> match inf_total a a0 with
                   left -> Tree_Node(a,t1,(F t0 a0))
                 | right -> Tree_Node(a0,t1,(F t0 a)))
  in F T
;;

let list_to_heap l =
  let rec F = function
    nil -> Tree_Leaf
  | cons(a,l1) -> insert (F l1) a
  in F l
;;

let merge l =
  let rec F = function
    nil -> (fun l1 -> l1)
  | cons(a,l0) ->
      (fun l2 -> let rec F0 = function
                   nil -> cons(a,l0)
                 | cons(a0,l2) ->
                     (match inf_total a a0 with
                        left -> cons(a,(F l0 (cons(a0,l2))))
                      | right -> cons(a0,(F0 l2)))
                 in F0 l2)
```

```
  in F l
;;

let heap_to_list T =
  let rec F = function
    Tree_Leaf -> nil
  | Tree_Node(a,t0,t1) -> cons(a,(merge (F t0) (F t1)))
  in F T
;;

let treesort l =
  heap_to_list (list_to_heap l)
;;
```

We may now test the program:

```
% camllight
# include "treesort";;
# let rec listn = function 0 -> nil
                         | n -> cons(random__int 10000,listn (pred n));;
# treesort (listn 10);;
- : list = cons (136, cons (760, cons (1512, cons (2776, cons (3064,
cons (4536, cons (5768, cons (7560, cons (8856, cons (8952, nil))))))))))
```

Some tests on longer lists (10000 elements) show that the program is quite efficient for Caml code.

5.5 Deriving proofs from algorithms

The above section gave a sketch of the development of the treesort algorithm. But so far it is very mysterious. Since I did not give the detail of the proofs, and I did not discuss either the realisability interpretation which is the principle behing the compiling algorithm, it is hard to understand why we actually obtain a program which may be recognized as (a functional implementation of) treesort. Clearly, a lot of knowledge about treesort, such as its recursion structure, has been hidden in the proof method, and a different proof of the specifications would have yielded another sorting program. This argument is a well-known objection to the general methodology of program synthesis.

Hopefully, there is a solution to this problem. It lies in the recognition that the translation from the proof to the program is basically a forgetful functor: the proof term is analysed as intermixing constructive arguments and logical argumentation without constructive contents, which ought to be done once and for all at compile time. The algorithm is thus a substructure of the proof, where all the logical argumentation has been erased as a comment. Conversely, the proof term may be obtained from the algorithm by just restoring this logical argumentation about the data, without modifyng the control structure.

This methodology has been successfully implemented by Catherine Parent in a recent thesis, which is the basis for a package of specialised tactics, whose aim is to help the user into constructing the proof from the algorithm. We shall not describe this facility in detail here, and will just show by example what happens for the treesort example.

The algorithm language we use is still under design. It is called Real, standing for "Realisation Language". For specialists, this language may be thought of as Girard's $F\omega$ enriched with inductive types. Its main constructs are recursion and pattern-matching. A Real program may be annotated by comments in the form of Coq assertions decorating its abstract syntax tree. These annotations are useful to state invariants used to generate induction hypotheses, in order to help the `Program` tactic in its proof-generation process. A Real program may be given as input to `Program` at any point in the proof by the `Realizer Coq` command. The `Program` tactic tries to infer as much of the proof as possible from the program, leaving as subgoals to be proved by the user only the purely logical lemmas. Thus we have turned effectively Coq into a verification condition generator, converting our program synthesis tools back into a program verification methodology.

For instance, the work of the human prover is minimal, once he gives the Real algorithm for treesort, i.e.

```
Theorem treesort : (l:list)
  {m:list | (sort m) & (meq (list_contents l) (list_contents m))}.
1 subgoal

  ============================
   (l:list)
    {m:list| (sort m) & (meq (list_contents l) (list_contents m))}

treesort < Realizer [l:list](heap_to_list (list_to_heap l)).
treesort < Program_all.
1 subgoal

  l : list
  T : Tree
  i : (is_heap T)
  m : (meq (list_contents l) (contents T))
  l0 : list
  s : (sort l0)
  l1 : (a:A)(Tree_lower a T)->(list_lower a l0)
  m0 : (meq (contents T) (list_contents l0))
  ============================
   (meq (list_contents l) (list_contents l0))

treesort < Apply meq_trans with (contents T); Trivial.
Subtree proved!
```

We may similarly inspect the proofs of the various sub-specifications of treesort in order to retrieve the full algorithm in a top-down fashion:

```
heap_to_list := [t:Tree][{H:(is_heap t)}]
   (is_heap_rec list
```

```
      nil
      [a:A][t1,t2:Tree][l1,l2:list](cons a (merge l1 l2))
      t).

merge := [l:list][{H:(sort l)}]
    (sort_rec list->list
          [l1:list]l1
          [a:A][l1:list][H0:list->list][l2:list][{H':(sort l2)}]
              (sort_rec list
                   (cons a l1)
                   [a0:A][l3,l4:list]
                       <list>if (inf_total a a0)
                             then (cons a (H0 (cons a0 l3)))
                             else (cons a0 l4)
                     l2)
              l).

sort_rec := [P:Set][H:P][H0:A->list->P->P][y:list]
               <P>Match y with
                    H
                    [a,l,Pl](H0 a l Pl)
                  end.

list_to_heap := [l:list]<Tree>Match l with
                     Tree_Leaf
                     [a,y,H](insert H a)
                             end.
```

Most importantly, `insert` contains the crux of treesort:

```
insert := [t:Tree][{H:(is_heap t)}]
            (is_heap_rec A->Tree
               [a:A](Tree_Node a Tree_Leaf Tree_Leaf)
               [a:A][t1,t2:Tree][H0,H1:A->Tree][a0:A]
                     <Tree>if (inf_total a a0) then
                                (Tree_Node a t2 (H0 a0))
                           else (Tree_Node a0 t2 (H0 a))
               t).

is_heap_rec := [P:Set][H:P][H0:A->Tree->Tree->P->P->P][y:Tree]
                    <P>Match y with
                        H
                        [a,G,PG,D,PD](H0 a G D PG PD)
                       end.
```

The Real language is still under design. It lacks the smooth pattern-matching definition principle of ML and `Recursive Definition`. It does not allow imperative style with assignable variables

and mutable data structures, and does not possess non-local control structures such as exceptions. But it is a step in the direction of a secure programming language for safety critical applications.
Acknowledgements. The realisability interpretation is due to Christine Paulin-Mohring. Benjamin Werner and Jean-Christophe Filliâtre implemented the extraction to ML program. Catherine Parent implemented the `Program` tactic library. This example of treesort was developed by B. Werner and G. Huet.

Many algorithms were proved correct in `Coq` using this methodology: various sorting programs, insertion in AVL trees, transitive closure, shortest path in graphs, unification, etc. A tautology checker obtained by reduction to canonical forms of IF trees is described in detail in [47].

Bibliography

[1] P. Aczel. "Galois: A Theory Development Project." Turin workshop on the representation of mathematics in Logical Frameworks, January 1993.

[2] J. A. Altucher and P. Panangaden. "A Mechanically Assisted Constructive Proof in Category Theory." In proceedings of CADE 10, Springer-Verlag LNCS, 1990.

[3] R. Asperti and G. Longo. "Categories, Types, and Structures." MIT Press, 1991.

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

[5] 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.

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

[7] R. Boyer, J Moore. "A Computational Logic." Academic Press (1979).

[8] N.G. de Bruijn. "The mathematical language AUTOMATH, its usage and some of its extensions." Symposium on Automatic Demonstration, IRIA, Versailles, 1968. Printed as Springer-Verlag Lecture Notes in Mathematics **125**, (1970) 29–61.

[9] 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.

[10] N.G. de Bruijn. "Automath a language for mathematics." Les Presses de l'Université de Montréal, (1973).

[11] N.G. de Bruijn. "A survey of the project Automath." (1980) in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[12] N.G. de Bruijn. "A riffle shuffle card trick and its relation to quasi crystal theory." Nieuw Archief Voor Wiskunde **5** 3 (1987) 285–301.

[13] R.L. Constable et al. "Implementing Mathematics in the Nuprl System." Prentice-Hall (1986).

[14] R.L. Constable and N.P. Mendler. "Recursive Definitions in Type Theory." In Proc. Logic of Programs, Springer-Verlag Lecture Notes in Computer Science **193** (1985).

[15] Th. Coquand. “Une théorie des constructions.” Thèse de troisième cycle, Université Paris VII (Jan. 85).

[16] T. Coquand. “An analysis of Girard's paradox.” Proceedings of LICS, Cambridge, Mass. July 1986, IEEE Press.

[17] T. Coquand. “Metamathematical Investigations of a Calculus of Constructions.” Rapport de recherche INRIA 1088, Sept. 89. In “Logic and Computer Science,” ed. P. Odifreddi, Academic Press, 1990, 91–122.

[18] Th. Coquand, G. Huet. “Constructions: A Higher Order Proof System for Mechanizing Mathematics.” EUROCAL85, Linz, Springer-Verlag LNCS 203 (1985).

[19] Th. Coquand, G. Huet. “Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions.” Logic Colloquium'85, Edited by the Paris Logic Group, North-Holland, 1987.

[20] T. Coquand and C. Paulin-Mohring. “Inductively defined types.” Workshop on Programming Logic, Göteborg University, Båstad, (89). International Conference on Computer Logic COLOG-88, Tallinn, Dec. 1988. LNCS 417, P. Martin-Löf and G. Mints eds., pp. 50-66.

[21] 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.

[22] Cristina Cornes and Judicaël Courant and Jean-Christophe Filliâtre and Gérard Huet and Pascal Manoury and Christine Paulin-Mohring and César Muñoz and Chetan Murthy and Catherine Parent and Amokrane Saïbi and Benjamin Werner. *The Coq Proof Assistant Reference Manual Version 5.10.* To appear as an INRIA Technical Report.

[23] 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.

[24] G. Dowek and A. Felty and H. Herbelin and G. Huet and C. Murthy and C. Parent and C. Paulin-Mohring and B. Werner. *The Coq Proof Assistant User's Guide Version 5.8.* Technical Report 154, INRIA, May 1993.

[25] R. Dyckhoff. “Category theory as an extension of Martin-Löf type theory.” Internal Report CS 85/3, Dept. of Computational Science, University of St. Andrews, Scotland.

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

[27] M. Gardner. Mathematical Recreation column, Scientific American, Aug. 1960.

[28] M. Gardner. Chapter 9, “New Mathematical Diversions from Scientific American.” George Allen and Unwin Ltd, London, 1966. Reprinted Simon and Schuster, 1971.

[29] N. Gilbreath. “Magnetic Colors.” The Linking Ring, **38** 5 (1959).

[30] J.Y. Girard, Y. Lafont & P. Taylor. "Proofs and Types." *Cambridge Tracts in Theoretical Computer Science*, (89) Cambridge University Press.

[31] M. J. Gordon, A. J. Milner, C. P. Wadsworth. "Edinburgh LCF." Springer-Verlag LNCS **78** (1979).

[32] R. Harper and R. Pollack. "Type checking with universes." Theoretical Computer Science 89, 1991.

[33] M. Hofmann. "Elimination of extensionality in Martin-Löf type theory." Proceedings of workshop TYPES'93, Nijmegen, May 1993. In "Types for Proofs and Programs", Eds. H. Barendregt and T. Nipkow, LNCS 806, Springer-Verlag 1994.

[34] G. Huet. "Initiation à la Théorie des Catégories." Notes de Cours, DEA Paris 7, Nov. 1985.

[35] G. Huet. "Induction Principles Formalized in the Calculus of Constructions." TAPSOFT87, Pisa, March 1987. Springer-Verlag Lecture Notes in Computer Science 249, 276–286.

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

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

[38] 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.

[39] G. Huet. "Residual theory in λ-calculus: a formal development". J. of Functional Programming **4,3** (1994) 371–394.

[40] Gérard Huet and Gilles Kahn and Christine Paulin-Mohring. *The Coq Proof Assistant Version 5.10. A Tutorial.* To appear as a technical report.

[41] P. B. Jackson. "Enhancing the NuPRL proof development system and applying it to computational abstract algebra." Ph.D. dissertation, Cornell University, Ithaca, NY, 1995.

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

[43] S. Mac Lane. "Categories for the working mathematician". Springer-Verlag, 1971.

[44] 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.

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

[46] 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.

[47] C. Paulin-Mohring and B. Werner. *Synthesis of ML programs in the system Coq. Journal of Symbolic Computation*, 15:607–640, 1993.

[48] L. C. Paulson. "A higher implementation of rewriting." Science of Computer Programming, 3:119-149, 1983.

[49] 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.

[50] 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.

[51] D. E. Rydeheard and R. M. Burstall. "Computational Category Theory". Prentice Hall, 1988.

[52] A. Saïbi. "Une axiomatisation constructive de la théorie des catégories." Rapport de Recherche, en préparation.

[53] 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.