The Constructive Engine

Gérard Huet

INRIA

Rocquencourt, France

Introduction

The Calculus of Constructions is a higher-order formalism for writing constructive proofs in a natural deduction style, inspired from work of de Bruijn [4, 7], Girard [21] and Martin-Löf [33]. The calculus and its syntactic theory were presented in Coquand's thesis [12], and an implementation by the author was used to mechanically verify a substantial number of proofs demonstrating the power of expression of the formalism [15]. The Calculus of Constructions is proposed as a foundation for the design of programming environments where programs are developed consistently with formal specifications[37]. This note presents in detail an implementation in CAML[18, 44] of a proof-checker for the calculus. This proof-checker proceeds by operating an abstract machine, called the constructive engine.

The description in this paper is close in spirit to the inference system described in section 10.2 of [13]. The main departure is the addition of a system of constants, allowing a form of definitional equality. The implementation shown corresponds to a simplification of version 4.9 of the system. Differences with the actual implementation are discussed below.

1 Technical Preliminaries

The constructive engine is an environment machine which manipulates λ-expressions with a rich language of types. There are several sorts of variables which may appear in the terms of the calculus. The global variables, as well as the global constants, are referenced with their name, a concrete string of ASCII characters. The environment is looked up through a hash table for fast access to the declaration information of the global. The locally bound variables are represented by their binding depth, in the manner of de Bruijn[3]. Since the constructive engine works by weaving back and forth between the environment and the currently constructed term, we need to switch representations efficiently. Also, we need to be able to compare two terms modulo β-reduction, in order to implement type equality checking. This section presents various programming techniques developed for such algorithms.

1.1 λ-calculus in the de Bruijn notation

We now recall the de Bruijn's notation of λ-calculus. This abstract representation of λ-expressions was originally introduced by de Bruijn[3], and later extended to Automath's

[1]A preliminary version of this paper, describing version 4.5 of the engine, was presented as an invited lecture at CAAP88. This paper describes a simplified view of the current 4.9 implementation.

Λ structures by Jutting[29].

We first consider untyped λ-calculus. We use CAML as our specification language. The type of *concrete* λ-terms may be defined as:

```
#type concrete =
#       Var of string                   (* x *)
#   |   Lambda of string * concrete     (* [x]E *)
#   |   Apply of concrete * concrete    (* (E1 E2) *);;
Type concrete defined
     Var : (string -> concrete)
   | Lambda : (string * concrete -> concrete)
   | Apply : (concrete * concrete -> concrete)
```

Thus every variable, whether it is free or bound, is implemented by a concrete string. Such representation needs complex substitution operations, in order to guarantee the preservation of the right binding relationships. Variable renamings (called α-conversion) may be necessary, and thus new names have to be coined. In order to avoid this naming problem, we define the type of *abstract* λ-terms:

```
#type lambda =
#       Ref of num             (* variables *)
#   |   Abs of lambda          (* lambda abstraction *)
#   |   App of lambda * lambda (* apply function L1 to argument L2 *);;
Type lambda defined
     Ref : (num -> lambda)
   | Abs : (lambda -> lambda)
   | App : (lambda * lambda -> lambda)
```

Now each variable occurrence is represented by its *binding depth*, i.e. the relative depth with respect to its binding abstraction operator. A term of type `lambda` containing n free variables is valid in a context of length n, according to:

```
#let rec valid n = function
#       Ref(m)     -> m <= n
#   |   Abs(l)     -> valid (n+1) l
#   |   App(l1,l2) -> valid n l1 & valid n l2;;
Value valid = <fun> : (num -> lambda -> bool)
```

Thus a *closed* λ-term is one which is valid in an empty context:

```
#let closed = valid 0;;
Value closed = <fun> : (lambda -> bool)
```

We write Λ_n for the set of lambdas M such that *valid n M*.

The recursion structure of algorithm `valid` is characteristic of computations on lambdas. The corresponding induction principle may be stated as:

Contextual Induction Principle. Let P_n be a property of lambdas, indexed by a natural number n, and satisfying the closure conditions:

- $P_n(M) \wedge P_n(N) \Rightarrow P_n(App(M, N))$

- $P_{n+1}(M) \Rightarrow P_n(Abs(M))$
- $0 < m \leq n \Rightarrow P_n(Ref(m))$.

Then $P_n(M)$ for every $M \in \Lambda_n$. Here is an algorithm for translating a concrete term to its abstract representation, given a context of free variables:

```
#exception Unbound;;
Exception Unbound defined

#let index_of id = search 1
#where rec search n = function
#       (name::names) -> if name=id then n
#                        else search (n+1) names
#      | [] -> raise Unbound;;
Value index_of = <fun> : ('a -> 'a list -> num)

#let rec parse_env lvars = abstract
#where rec abstract = function
#   Var(name)          -> Ref (index_of name lvars)
# | Lambda(id,conc)    -> Abs(parse_env (id::lvars) conc)
# | Apply(conc1,conc2) -> App(abstract conc1,abstract conc2);;
Value parse_env = <fun> :
      (string list -> concrete -> lambda)

#(* parsing closed lambdas *)
#let parser = parse_env [];;
Value parser = <fun> : (concrete -> lambda)
```

The abstract representation is not convenient for the human interface, and thus we assume we have declared a grammar for concrete syntax, as follows[44]:

```
#grammar concrete =
#rule entry Lambda = parse
#      Term x                         -> parser(x)
#and Term = parse
#      "("; Term_list x; Term y; ")" -> Apply(x,y)
#    | IDENT x                        -> Var(x)
#    | "["; Binder x; "]"; Term y     -> list_it (curry Lambda) x y
#    | "("; Term x; ")"               -> x
#and Term_list = parse
#      Term x                         -> x
#    | Term_list x; Term y            -> Apply(x,y)
#and Binder = parse
#      IDENT(x)                       -> [x]
#    | IDENT(x); ","; Binder y        -> x::y;;
Warning: variable(s)
   curry, Apply, list_it, prefix ::, Lambda, Var, parser
```

```
will be dynamically bound
Calling Yacc ... ..........................................
Value concrete = <fun> : (string -> Parsers)
Grammar concrete for programs defined
  entry Lambda
```

We may now give examples in concrete syntax:

```
#<<[x](x [y](x y))>>;;
(Abs (App ((Ref 1),(Abs (App ((Ref 2),(Ref 1))))))) :
 lambda
```

Remark that the two occurrences of x have different indexes 1 and 2, and that conversely the references `(Ref 1)` correspond to two distinct variables x and y.

The first fundamental algorithm concerns the recomputation of global references to global variables across n levels of binding. This is computed by the algorithm `lift` below.

```
#let lift n = lift_rec 1
#where rec lift_rec k = function
#    Ref(i)        -> if i<k then Ref(i)   (* bound variable : invariant *)
#                           else Ref(i+n) (* free variable  : relocated *)
#  | Abs(lam)      -> Abs(lift_rec (k+1) lam)
#  | App(lam,lam') -> App(lift_rec k lam,lift_rec k lam');;
Value lift = <fun> : (num -> lambda -> lambda)
```

Now we may program substitution, as follows.

```
#let subst lam =
#let lift_lam n = lift n lam in subst_lam 1
#where rec subst_lam n = function
#    Ref(k) -> if k=n then lift_lam (n-1)    (* the substituted variable *)
#              if k<n then Ref(k)            (* bound variables *)
#              else Ref(k-1)                 (* free variables *)
# |  Abs(lam') -> Abs(subst_lam (n+1) lam')
# |  App(lam1,lam2) -> App(subst_lam n lam1,subst_lam n lam2);;
Value subst = <fun> : (lambda -> lambda -> lambda)
```

For instance, we may now program the conversion of a solvable λ-term to its head normal form as:

```
#let rec hnf = function
#    Ref(n)         -> Ref(n)
# |  Abs(lam)       -> Abs(hnf lam)
# |  App(lam1,lam2) -> let lam'=hnf lam1 in match lam' with
#                      Abs(lam) -> hnf(subst lam2 lam)
#                 |  _        -> App(lam',lam2);;
Value hnf = <fun> : (lambda -> lambda)
```

These algorithms are satisfactory, as executable specifications. They are nor satisfactory as efficient programs. We see two main problems with this approach. The first one is that too much unnecessary copying of structures is effected. The second one is that the operation of λ-reduction is in some sense not elementary enough: we would like to substitute occurrence by occurrence, in a maximally lazy fashion. We shall not tackle the second problem here, but will try and share as much as possible during the computation, in order to remedy the first problem. It should be noted that this very basic problem of efficient computation on λ-terms is still largely open. The sharing of combinatory dags applies only to weak reduction (where one does not reduce inside abstractions). The same is true of abstract environment machines such as the SECD, FAM or CAM machines, which furthermore compute in applicative order (innermost) as opposed to the normal order (leftmost-outermost) corresponding to the standardization theorem. Wadsworth's method[43] does not avoid unnecessary duplications. Levy's sharing of redex families[31] has not yet been implemented in a computationnally efficient way. We do not claim that de Bruijn's abstract representation will lead to the best implementations of λ-calculus. However, the sharing method which we shall now present leads to an acceptable behaviour for our application to the constructive engine.

1.2 Sharing morphisms

Let us forget λ-calculus for the sake of the current discussion, and consider the simpler case of terms built up from free constructors. For instance, consider the following algebra, corresponding to the abstract syntax of terms for some simplified arithmetic:

```
#type term =
#    Var of string
#  | Plus of term * term
#  | Minus of term
#  | Constant of num;;
Type term defined
     Var : (string -> term)
   | Plus : (term * term -> term)
   | Minus : (term -> term)
   | Constant : (num -> term)
```

Now let us consider a simple-minded implementation of first-order substitution over these terms, with substitutions represented as association lists:

```
#let naive_subst sig = subst_rec
#where rec subst_rec = function
#    Var(name)   -> if mem_assoc name sig then assoc name sig else Var(name)
#  | Plus(t1,t2) -> Plus(subst_rec t1,subst_rec t2)
#  | Minus(t)    -> Minus(subst_rec t)
#  | Constant(n) -> Constant(n);;
Value naive_subst = <fun> :
      ((string * term) list -> term -> term)
```

This method is clearly computationally absurd, since a full copy of a term is effected, even when the substitution is empty. Worst, substitution will un-share terms naturally shared on the underlying dags, such as `let M=Constant(2) in Plus(M,M)`. This may be acceptable in certain situations, for instance if we want to rewrite the two occurrences of M above in distinct ways. But if we think of these abstract terms as applicative structures, we want to profit of the possible sharing to represent in a shared fashion the substitution to a non-linear term such as `Plus(Var(x),Var(x))`.

One solution to this problem has been proposed long time ago by Boyer and Moore[1], and this "structure sharing" representation is now one of the classical ways to implement PROLOG. In this method, we manipulate substitution closures rather than terms, and this has some undesirable effects. For instance, we do not have a direct access to the top of the term anymore, and this access gets more and more costly the further we substitute. After a while, the whole structure may become completely inverted. For this reason, the "structure copying" implementation is still retained in many applications.

At the other end of the spectrum, we find the representations which share through congruence closure[40, 20]. These methods are fine for *ground* terms (not containing free variables). In the case where we manipulate mostly terms with variables, congruence closure is not really applicable, and produces more overhead than it saves.

We shall now study how to share maximally in the standard representation of terms containing free variables. We shall not try and guess possible sharing through syntactic coincidences. We only care to preserve existing sharing, and to share as much as possible of a substituted version of a term with the original version. Let us first see how to implement this idea of sharing non-substituted portions of a term. Here is a simple recursive algorithm which reports, together with its result, a boolean value indicating whether sharing has been possible or not.

```
#let subst_and_share sig = fst o subst_rec
#where rec subst_rec x = match x with
#     Var(name)   -> if mem_assoc name sig then (assoc name sig,false)
#                                        else (x,true)
#   | Plus(t1,t2) -> let (t'1,b1)=subst_rec t1
#                    and (t'2,b2)=subst_rec t2
#                    in if b1&b2 then (x,true) else (Plus(t'1,t'2),false)
#   | Minus(t)    -> let (t',b)=subst_rec t
#                    in if b then (x,true) else (Minus(t'),false)
#   | Constant(n) -> (x,true);;
Value subst_and_share = <fun> :
      ((string * term) list -> term -> term)
```

Again, we have a correct specification of what we exactly mean by sharing, but the program above is very costly in storage, since the computation of the sharing relationship builds on the side a duplicate of the structure in the shape of a tree of booleans. If the aim is to avoid building un-necessary structure, we must clearly avoid these explicit boolean values.

The next idea is to replace the values `(true,x)` above by an exception, signaling that sharing is possible. Let us define a *sharing morphism* as a term morphism, which raises a special exception `Identity` to signal that its result is identical to its argument. If `f` is such a sharing morphism, it may be applied on an argument with possible sharing with:

```
exception Identity;;
```

```
let share f x = try f(x) with Identity -> x;;
```

and now we may implement a sharing substitution with:

```
#let sharing_subst sig = share subst_rec
#where rec subst_rec = function
#    Var(name)   -> if mem_assoc name sig then assoc name sig
#                                        else raise Identity
#  | Plus(t1,t2) -> (try Plus(subst_rec t1,share subst_rec t2)
#                     with Identity -> Plus(t1,subst_rec t2))
#  | Minus(t)    -> Minus(subst_rec t)
#  | Constant(n) -> raise Identity;;
Value sharing_subst = <fun> :
      ((string * term) list -> term -> term)
```

It is clear that we now have maximum sharing of the substituted term with its original pattern, without extra structure.

It may be objected however that the substitution algorithm is now cluttered with extra cases, which brings two problems: readability is questionable, and mistakes may occur in the transformation from a copying algorithm to its shared version. Note in particular that the case corresponding to an n-ary constructor splits into 2^{n-1} subcases. This objection may be overcome, by noticing that the transformation is systematic enough to be encapsulated in a macro.

More precisely, given two arguments, one which gives the signature of the constructors over which we want a sharing morphism effect, and the second which gives what happens on the others, we may write without difficulty a macro `morphism` which generates the full recursion. Without giving its actual code, which would be hard to understand for readers not familiar with CAML, here is the instance of its call in our example:

```
#pragma let term_signature = [("Plus",2);("Minus",1)]
        and replace_sig =
   <:CAML<function Var(x)   -> if mem_assoc x sig then assoc x sig
                               else raise Identity
              | Constant(_) -> raise Identity>>;;
(* Macro-generated version of sharing_subst above *)
let subst sig = share #(morphism term_signature replace_sig);;
```

Remark 1. The macro `replace_sig` above is enough to illustrate the example. It is unsatisfactory in that it assumes that the argument to the `substitute` algorithm is called `sig`. A more satisfactory version of the macro, parameterized in an adequate fashion, is easy to define with the mecanism of *anti-quotation* of CAML[44]:

```
#pragma let replace_sig sig =
   <:CAML<function Var(x)      -> if mem_assoc x {^sig^} then assoc x {^sig^}
                                  else raise Identity
                 | Constant(_) -> raise Identity>>;;
```

and now the identifier `sig` is generated in its proper scope:

```
let subst sig = let SIG = <:CAML<sig>> (* abstract syntax of sig *)
                in share #(morphism term_signature (replace_sig SIG));;
```

This remark applies to the other macros explained in the next section. However, we prefer to present the more readable "unsafe" versions.

Remark 2. The sharing mechanism we present uses ML's exceptions. Other mechanisms may be thought of. For instance, if one assumes the existence of a primitive `eq` which tests for *physical* identity, we may program a sharing substitution in the following manner:

```
...
match arg with
...
 | App(x,y) -> let x'=subst x and y'=subst y
               in if eq(x,x') and eq(y,y') then arg else App(x',y')
...
```

Using one mechanism or the other may be motivated by æsthetics or implementation reasons (availability of primitives, efficiency). Somehow the user should not have to worry about such low-level detail, but obviously current implementations of functional programming languages lack the proper memory management primitives needed to express naturally this economic use of data structures.

1.3 Binding morphisms with sharing

The ideas of the previous section may be extended to algebras where certain constructors are binding in some of their arguments. For instance, in `lambda` above, `Abs` is binding in its argument, whereas `App` is not. We indicate this information with the following *descriptors*:

```
#pragma let lambda_descr = [("Abs",[true]);("App",[false;false])];;
```

We now give the code for lifting `Ref` indexes over k levels of binding:

```
#pragma let lift_k =
  <:CAML<function Ref(i) -> if i<n then raise Identity (* bound var *)
                            else Ref(i+k) (* free var is shifted *)>>;;
```

and the general case is macro-generated with the help of a macro `binding_morphism` which generalizes the idea of sharing morphisms to binding constructors implemented with de Bruijn indexes. Without boring the reader with the actual details of the macro `binding_morphism`, let us just indicate an example of macro-expansion:

```
#pragma binding_morphism lambda_descr lift_k =
<:CAML<let rec f n = function
           Abs(x1)    -> Abs(f (n+1) x1)
          | App(x2,x1) -> App(try f n x2, (try f n x1 with Identity -> x1)
                              with Identity -> x2,f n x1)
          | Ref(i)     -> if i < n then raise Identity else Ref(i+k)
       in f 1>>
```

Now we get the lift function as:

```
#let lift k lam = if k=0 then lam
#                 else share #(binding_morphism lambda_descr lift_k) lam;;
Value lift = <fun> : (num -> lambda -> lambda)
```

Remark that we save exploring the term `lam` in the case $k = 0$, which we know beforehand to be an identity.

We now iterate this idea for substitution:

```
#pragma let subst_lam =
<:CAML<function Ref(k) -> if k=n then lift_lam (n-1) (* substituted var *)
                          if k<n then raise Identity (* bound var *)
                          else Ref(k-1)>>            (* free var *);;

#let subst lam =
#    let lift_lam n = lift n lam
#    in share #(binding_morphism lambda_descr subst_lam);;
Value subst = <fun> : (lambda -> lambda -> lambda)
```

This substitution algorithm is still far from perfect. First we should recognize the special case when we substitute a closed term, since in that case no lifting is necessary. Further, we should share all instances of `(lift n lam)` for a given n. These ideas are implemented in the following version:

```
#pragma let subst_closed_lam =
   <:CAML<function Ref(k) -> if k=n then lam         (* lam is shared *)
                             if k<n then raise Identity (* bound var *)
                             else Ref(k-1)>>            (* free var *);;

#let subst lam =
# if closed lam
#    then share #(binding_morphism lambda_descr subst_closed_lam) lam
# else let instances = ref [] in
#      let lift_lam n = (* local memo version *)
#          if n=0 then lam
#          else (assoc n !instances (* memo effect *)
#                ? let new_instance = lift n lam
#                  in (instances:=(n,new_instance)::!instances;
#                      new_instance))
#      in share #(binding_morphism lambda_descr subst_lam) lam;;
Value subst = <fun> : (lambda -> lambda)
```

Further optimizations are possible. For instance, the initial pass to check whether `lam` is closed or not could set up the lifting code with maximum sharing, for any relocating value n, since intuitively the shared areas are the same. The above version recomputes this information p times, where p is the number of distinct levels where the substituted variable occurs. This would not affect the quantity of sharing, it would only speed up its computation, and since the code would be significantly more complicated we shall not do it here.

Although in some sense we share maximally a term with its substituted versions, we fail to propagate this sharing, since recursive exploration of a term where certain nodes are shared does not recognize this sharing, and will therefore undo this sharing by further substitutions. In order to preserve sharing, we ought to remember in a table pairs of addresses (original-term-node, substituted-term-node), in the spirit of traditional graph-copying algorithms. We shall ignore this optimization here.

Finally, a smarter use of indexes may be imagined, where lifting and substitution are delayed as much as possible, by keeping relocation indexes in the expressions themselves. However, this idea is not as easy as it sounds, since such relocation annotations do not commute easily with abstraction, and very quickly we run into the problem of manipulating complex annotations of the form "lift by n_1 all variables greater the m_1, and ..." Clearly, we are not saying the last word on possible implementations of λ-calculus in the de Bruijn style.

2 Application to the Calculus of Constructions

2.1 Term and types

The Calculus of Constructions, originally described in Thierry Coquand's thesis[12], refers to a family of formalisms which permit to describe both objects structured with types, and natural deduction proofs of higher-order logic. We thus have naturally two levels of descriptions:

```
#type level = Object | Proof;;
Type level defined
     Object : level
   | Proof : level
```

The two levels are intermixed in uniform term structures of a typed λ-calculus, where types are themselves terms of the same nature. At the level of proofs, we use the Curry-Howard isomorphism: propositions are seen as the types of their proofs.

We thus have two kinds of judgement:

- `Judge(M,T,Object)` means "Object `M` has type `T`".
- `Judge(M,P,Proof)` means "Proof `M` proves proposition `P`".

Such judgements refer implicitly to a global context of variable declarations and constant definitions. That is, we have possibly non-closed terms of a λ-calculus with constants. We shall come back later to the structure of the global environment. We may at this point define our types of terms and judgements:

```
#type constr =
#    Rel of num                         (* bound variables *)
#  | Var of string * judgement          (* free variables *)
#  | Const of string * judgement        (* constants *)
#  | Prop                               (* type of propositions *)
#  | Type of num                      (* universes *)
#  | App of constr * constr           (* application  (M N) *)
#  | Lambda of constr * constr        (* abstraction  [x:T]M *)
#  | Prod of constr * constr          (* product      (x:T)M *)
#and judgement = Judge of constr * constr * level;;
Type constr defined
     Rel : (num -> constr)
   | Var : (string * judgement -> constr)
   | Const : (string * judgement -> constr)
   | Prop : constr
   | Type : (num -> constr)
   | App : (constr * constr -> constr)
   | Lambda : (constr * constr -> constr)
   | Prod : (constr * constr -> constr)
Type judgement defined
     Judge : (constr * constr * level -> judgement)
```

The `Rel` variables are local variables implemented with de Bruijn indexes. The `Var` and `Const` refer to respectively variable declarations and constant definitions. Global identifiers are recognized by the parser as known in the environment as variable declarations or constant definitions, and the corresponding piece of the environment is pointed to by the corresponding term constructors `Var` and `Const`. This has the advantage of avoiding environment searches during type-checking. It has the drawback that our term structures are complicated dags sharing with the environment all the necessary information.

Both `Prod` and `Lambda` are operators which are binding in their second argument, the first one being the *type* of the bound variable. `Lambda` is λ-abstraction, whereas `Prod` is product formation for types, and universal quantification for propositions. Note our ambiguous use of the word "type", since the type of a term is either a type, i.e. a term of type `Type(i)` for a certain `i`, or a proposition, i.e. a term of type `Prop`. In this last case, we use the analogy of propositions with types, identifying a proposition with the type of it proofs. We thus have a λ-calculus at two levels. At the level `Object` abstraction is used for expressing the functionality of the mathematical objects and proposition schemes, in the spirit of Church's type theory. At the level `Proof` it corresponds to (intuitionistic) implication introduction in natural deduction.

The following function checks that its argument is a kind, i.e. of the form `Prop` or `Type(_)`, and returns the level of terms whose types are of that kind.

```
#let level_of_kind = function
#   Type(_) -> Object
# | Prop    -> Proof
# | _       -> error "Not a proposition or a type";;
Value level_of_kind = <fun> : (constr -> level)
```

Note the similarity of our term structures with the linguistic structures of Automath. In the terminology of Automath, our structure extends Nederpelt's Λ. However, we use it with a restricted notion of level, i.e. we have a *regular* language[19]. We distinguish between `Prod` and `Lambda`, in the spirit of AutPi. This was not done in the original calculus of Coquand[12]. We prefer to make the distinction now, because this entails unicity of types. We have also constants and δ-rules, in the spirit of $\Lambda\Delta$[10]. More importantly, we have higher-order quantification, like in Girard's system $F\omega$, which makes the calculus significantly more powerful than both the Automath systems, and the Martin-Löf type theories. We may thus develop higher-order mathematics, as advocated by Scott[42]. In the jargon of proof theorists, the system is "non-predicative".

2.2 Substitution

Substitution is a straightforward generalization of the ideas given in section 1.3 above. The descriptor `lambda_descr` is simply replaced by the descriptor

```
let constr_descriptor =
    [("App",[false;false]);
     ("Lambda",[false;true]);
     ("Prod",[false;true])];;
```

which describes the binding effect of the term constructors. The functions `lift` and `subst` are appropriately adapted.

The main departure from pure λ-calculus is that we have a calculus with *constants*. We thus have several substitution functions for various uses:

```
subst1 : constr -> constr -> constr
(* (subst1 c1 c2) substitutes c1 for Rel(1) in c2 *)
subst2 : string -> constr -> constr
(* (subst2 str c) substitutes Rel(1) for Var(str,_) in c *)
subst_con : string -> constr -> constr -> constr
(* (subst_con str c1 c2) substitutes c1 for Const(str,_) in c2 *)
```

Each of these functions is programmed with the help of the macro `binding_morphism`.

2.3 Equality

Equality of terms means inter-convertibility by β-conversion. Because the calculus is confluent and nœtherian, this could be decided by comparing the normal forms of the two terms. However this is not what we want in a calculus with constants. We want to verify equivalence of terms with as little constant expansion as possible, i.e. to try intensional equality before computing unfolding the constant definitions.

For instance, consider the step of proof which consists in applying a lemma `L1` stating that every reflexive relation `R` is cyclic. We thus have, in the current context, a constant:

```
L1 : (T:Type)(R:T->T->Prop)(Reflexive R)->(Cyclic R)
```

Assume that the current context contains declarations for a type `T1`, a relation `R1` over `T1`, and a hypothesis `H1` that `R1` is reflexive. Now we want to be able to apply `(L1 T1 R1 H1)`. The last application will involve verifying equality of two instances of the proposition `(Reflexive R1)`. It would be absurd here to look up the definition of the concept `Reflexive`, and to expand to normal form the two propositions. We want to avoid going back to first principles, but to do as we usually do in first-order logic, when `Reflexive` is a non-analysed binary predicate. Of course, in last resort, we must allow for possible constant expansion.

This motivates the introduction of a data type of *approximations*, which are used to compute head normal forms progressively.

```
#(* The two kinds of variables *)
#type reference =
#     Local of num
#   | Global of string;;
Type reference defined
     Local : (num -> reference)
   | Global : (string -> reference)

#type approximation =
#     Abstraction of constr * constr
#   | Product of constr * constr
#   | Variable of reference * constr list
#   | Constant of (string * judgement) * constr list
#   | Propconst
#   | Typeconst of num;;
Type approximation defined
     Abstraction : (constr * constr -> approximation)
   | Product : (constr * constr -> approximation)
   | Variable : (reference * constr list -> approximation)
   | Constant :
     ((string * judgement) * constr list -> approximation)
   | Propconst : approximation
   | Typeconst : (num -> approximation)

#(* One step of approximation *)
#let rec approx stack = hnf
#where rec hnf = function
#     Rel(n)         -> Variable(Local(n),stack)
#   | Var(name,_)    -> Variable(Global(name),stack)
#   | Const(n_j)     -> Constant(n_j,stack)  (* No expansion! *)
#   | Prop           -> if stack=[] then Propconst
#                       else anomaly "Prop cannot be applied"
#   | Type(n)        -> if stack=[] then Typeconst(n)
#                       else anomaly "Type cannot be applied"
#   | App(c1,c2)     -> approx (c2::stack) c1
#   | Lambda(c,c')   -> (match stack with
#                          []         -> Abstraction(c,c')
#                        | arg1::rest -> approx rest (subst1 arg1 c'))
#   | Prod(c,c')     -> if stack=[] then Product(c,c')
#                       else anomaly "Product cannot be applied";;
Value approx = <fun> :
      (constr list -> constr -> approximation)
```

```
#let approxim c = approx [] c;;
Value approxim = <fun> : (constr -> approximation)
```

We are now ready to describe type equality, which proceeds by λ-conversion and constant expansion. This last operation is delayed as much as possible.

```
#(* Constant expansion *)
#let expand = function
#  Constant((_,Judge(c,_)),stack) -> approx stack c
#| _ -> anomaly "Trying to expand a non-constant";;
Value expand = <fun> : (approximation -> approximation)

#(* equality of terms modulo conversion *)
#let rec conv term1 term2 = eqappr (approxim term1,approxim term2)
#    where rec eqappr = function
#       (Abstraction(c1,c2),Abstraction(c'1,c'2)) -> conv c1 c'1
#                                                   & conv c2 c'2
#     | (Product(c1,c2),Product(c'1,c'2)) -> conv c1 c'1 & conv c2 c'2
#     | (Variable(n1,l1),Variable(n2,l2)) -> (n1=n2) &
#                (length l1 = length l2) & for_all2 conv l1 l2
#     | (Propconst,Propconst) -> true
#     | (Typeconst(u1),Typeconst(u2)) -> u1=u2
#     | ((Constant(s1,l1) as appr1),(Constant(s2,l2) as appr2)) ->
#         (* try first intensional equality *)
#         (eq(s1,s2) & (length(l1) = length(l2)) & (for_all2 conv l1 l2))
#         (* else expand the second occurrence (arbitrary heuristic) *)
#      or eqappr(appr1,expand appr2)
#     | ((Constant(_) as appr1),appr2) -> eqappr(expand appr1,appr2)
#     | (appr1,(Constant(_) as appr2)) -> eqappr(appr1,expand appr2)
#     | _ -> false;;
Value conv = <fun> : (constr -> constr -> bool)
```

The next function effects one step of head normal form leading to Prop, Type, or Prod. It is used for making explicit the type of a construction.

```
#let hnftype = apprec []
#where rec apprec stack = app_stack
#where rec app_stack = function
#     Rel(_)              -> error "Typing error 1"
#   | Var(_)              -> error "Typing error 2"
#   | Const(_,Judge(c,_)) -> app_stack(c)
#   | App(c1,c2)          -> apprec (c2::stack) c1
#   | Lambda(_,c)         -> (match stack with
#                                []       -> error "Typing error 3"
#                              | c'::rest -> apprec rest (subst1 c' c))
#     (* Prod/Prop/Type *)
#   | c                   -> if stack=[] then c
#                            else anomaly "Cannot be applied";;
Value hnftype = <fun> : (constr -> constr)
```

2.4 Structure of the environment

The environment contains variable declarations, constant definitions, values waiting to be applied, and types waiting to be used as coercions. We shall explain the last two kinds of items in the next section.

```
#type declaration =
#    Vardecl of string * judgement
#  | Constdecl of string * judgement
#  | Value of judgement
#  | Cast of judgement
#and context == declaration list;;
Type declaration defined
     Vardecl : (string * judgement -> declaration)
   | Constdecl : (string * judgement -> declaration)
   | Value : (judgement -> declaration)
   | Cast : (judgement -> declaration)
Type context abbreviates declaration list
```

We now declare the he initial state, consisting of the empty initial context, and the initial judgement, stating that `Prop` is an `Object` of type `Type(0)`.

```
#let ENV = ref ([]:context)
#and VAL = ref (Prop)          (* The current construction *)
#and TYP = ref (Type(0))       (* Its type *)
#and LEV = ref (Object)        (* Its level *);;
Value ENV = (ref []) : context ref
Value VAL = (ref Prop) : constr ref
Value TYP = (ref (Type 0)) : constr ref
Value LEV = (ref Object) : level ref
```

2.5 A digression on the meaning of inference rules

The constructive engine may be thought of as some kind of loom, where we weave back and forth between the environnement and the current value. The fabric is produced whenever we build a new component of the environment, by taking the current judgement to build a new variable declaration or a new constant definition. The fabric may be taken back by discharging variables as abstractions or products, forming arbitrary new patterns.

This "loom" implementation is in some way the spirit of natural deduction, as opposed to sequent calculus, in which proofs may be seen as produced by cooperating machines working in parallel. In some sense this is a deep distinction between natural deduction logical frameworks, and sequent calculus logical frameworks. For instance, compare and_intro:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \textit{natural}$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma \cup \Delta \vdash A \wedge B} \qquad \textit{sequent}$$

In the sequent formulation, Γ and Δ may be thought of as *sets* of propositions. In the natural formulation, Γ may be thought of as a list constructed progressively. However, the important distinction is not so much on the structural rules. It is that the two occurrences of Γ on the numerator of the natural rule do not denote two lists that somehow happen to be equal structurally: they denote the *same shared object*. Actually, the left top occurrence may be considered the binding occurrence of Γ, whereas the right top one, as well as the one in the denominator, may be considered residuals, or pointers to the data structure which was the state of the environment when applying the rule. Similarly, the occurrences of A and B in the denominator denote pointers to the propositions that have been bound to their respective binding occurrence on the numerator.

Thus we may regard inference rules of natural deduction formalisms as direct specifications of elementary machine operations, with registers for the environment, the current proof, and the currently proved proposition. This is the point of view we take for our implementation of the Calculus of Constructions.

It should be noticed that this point of view may not be adequate for theorem proving, as opposed to theorem checking. For synthesizing proofs, the sequent formulations may actually be preferable. The point we want to emphasize (and we believe has not been strongly stressed before) is first that natural versus sequent formulations of logic is not a choice based on æsthetic choice, but is important for its pragmatic use, and second that it suggests implementation techniques, since the meaning of meta-variables in the rules is quite different.

Once we notice that meta-variables in inference rules denote *shared* objects, it is natural to distinguish, in the various occurrences of such schematic variables, one *binding* occurrence, instanciated by pattern-matching, the others denoting *residuals* in the sense of λ-calculus, i.e. pointers to the corresponding substructure. For instance, consider the usual natural-deduction formulation of arrow-elim, i.e. application or modus ponens:

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B}$$

In order to make explicit the flow of information, when using this rule for proof-checking, it is obvious that the binding occurrences of Γ, M and N are in the denominator of the rule, whereas the binding occurrence of B is in the numerator. Concerning A, we have a problem. The one in the right argument of the rule may be considered binding: "Let A be the type of term N in environment Γ." The one in the left argument, also, could be considered binding in a system where unicity of types hold. Here, where types are only unique up to β-conversion, it could mean: "Convert the type of term M in environment Γ to its head normal form, check that it has the shape $A \rightarrow B$." But, in either case, we do not mean that the types of M and N should *share* a common substructure, but merely that the corresponding substructures ought to be *equal* (and here, actually, a recursive exploration to check this fact is meant). Thus, what we mean to specify is a rule that says: "Let C be the type of term M in environment Γ. Convert C to its head normal form. Check that it has the shape $A' \rightarrow B$. Check that A and A' are convertible." We are now close to a complete specification of the proof checker. Two details remain.

The first detail concerns the order in which M and N are actually typechecked in a sequential machine. There are two possible interpretations of application as a term constructor: the *left-sequential* one (first M and then N), and the *right-sequential* one (the opposite order). In either case, some explicit memorisation of the intermediate type shall be needed. We remark that this memorisation cannot be done simply in an auxiliary stack, since the proper scoping must be enforced. We are lead to stack the corresponding judgement as an item in the environment. In our implementation, we choose the right-sequential interpretation, and we are thus lead to an inference rule (where binding occurrences are underlined):

$$\frac{\Gamma \vdash N : \underline{A} \quad \Gamma \vdash M : \underline{C} \quad C \rhd \underline{A'} \to \underline{B} \quad A \equiv A'}{\underline{\Gamma} \vdash (\underline{M}\ \underline{N}) : B}$$

which is in turn implemented as the composition of two unary transition rules, with explicit stacking in the environment:

$$\Gamma \vdash N : B \quad \Longrightarrow \quad \Gamma; N : B \vdash \qquad\qquad \textit{stack\_value}$$

$$\Gamma; N : B \vdash M : C \quad \Longrightarrow (C \rhd A' \to B, A \equiv A') \ \ \Gamma \vdash (M\ \ N) : B \qquad\qquad \textit{pop\_apply}$$

In stack_value, the notation $\Gamma; N : B \vdash$ stands for stacking the current judgement in the environment without changing it. We are thus lead to the view of elementary machine operations (the conditional transition rules) corresponding to an explicit sequentialisation of inference rules. The memorisation insures that these elementary commands may be used independently of each other.

The second detail concerns conversion in the presence of the constant definitions, which are definitional equalities; in such systems the conversion depends on these equalities, and thus of the environment, in which they are declared. That is, in order to be completely rigorous we ought to write $\Gamma \vdash A \equiv A'$ above.

To a certain extent the situation may be compared to PROLOG. The inference rules, with suitable detail, correspond rather directly to PROLOG clauses. The relations $\rhd$ and $\equiv$ are executable predicates, and the underlining of binding occurrences correspond to input specifications à la Concurrent Prolog. This point of view is close to the one of the logical framework Typol[30]. The transition rules correspond to the basic instructions of a sequential PROLOG machine. One may argue that this low-level description is not really needed for the user of the logical environment. However, this amount of detail may be needed to derive useful meta-mathematical properties.

In the following, we shall describe our engine in terms of the CAML functions implementing its basic transition rules. The higher-level inference rules are given as comments.

2.6 Initialization

2.6.1 Resetting the current judgement

```
#let reset_current () =
#  VAL := Prop;
#  TYP := Type(0);
#  LEV := Object;;
Value reset_current = <fun> : (unit -> level)
```

2.6.2 Resetting, pushing and popping the environment

```
#let reset_env () =
#  ENV := []
#and push_env declaration =
#  ENV := declaration::!ENV
#and pop_env () = match !ENV with
#   [] -> error "Empty environment"
# | decl::env -> ENV := env; decl;;
Value reset_env = <fun> : (unit -> context)
Value push_env = <fun> : (declaration -> context)
Value pop_env = <fun> : (unit -> declaration)
```

2.6.3 Searching the environment

Note that we do not need to relocate the judgements stacked in the environment, since we do not use indexes, but identifiers for the globals.

```
#let search name = search_rec !ENV
#    where rec search_rec = function
#        [] -> error(name ^ " not declared")
#      | decl::rest -> match decl with
#              Vardecl(s,_)    -> if s=name then decl else search_rec rest
#            | Constdecl(s,_)  -> if s=name then decl else search_rec rest
#            | _               -> search_rec rest;;
Value search = <fun> : (string -> declaration)
```

2.7 The basic machine operations

There are few basic machine operations, partitioned into two kinds: the ones that build up the environment, and the ones that build up the current construction. For each such command, we give first the inference rule, then the CAML function which performs the corresponding machine instruction.

2.7.1 Introducing a new hypothesis

$$\frac{\Gamma \vdash M : Prop}{\Gamma; x : M \vdash}$$

$$\frac{\Gamma \vdash M : Type(i)}{\Gamma; x : M \vdash}$$

```
#let assume name =
#    let declaration =
#        let typ = hnftype !TYP
#        in let lev = level_of_kind typ
#           in Vardecl(name,Judge(!VAL,typ,lev))
#    in push_env declaration;;
Value assume = <fun> : (string -> context)
```

2.7.2 Introducing a new definition

$$\frac{\Gamma \vdash M : T}{\Gamma; x = M : T \vdash}$$

```
#let declare name =
#    let declaration = Constdecl(name,Judge(!VAL,!TYP,!LEV))
#    in push_env declaration;;
Value declare = <fun> : (string -> context)
```

2.7.3 Global Introduction

$$\frac{\Gamma = \Gamma_1; x : \Gamma_x; \Gamma_2}{\Gamma \vdash x : \Gamma_x} \qquad \frac{\Gamma = \Gamma_1; x = V : \Gamma_x; \Gamma_2}{\Gamma \vdash x : \Gamma_x}$$

```
#let consider name =
#    match (search name) with
#         Constdecl((_,Judge(_,typ,lev)) as constant) ->
#            (TYP:=typ;
#             LEV:=lev;
#             VAL:=Const(constant))
#       | Vardecl((_,Judge(typ,_,lev)) as variable) ->
#            (TYP:=typ;
#             LEV:=lev;
#             VAL:=Var(variable));;
Warning: 1 partial match in this phrase
Value consider = <fun> : (string -> constr)
```

2.7.4 Prop intro

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathit{Prop} : \mathit{Type}(1)}$$

```
#let proposition () = VAL:=Prop;TYP:=Type(1);LEV:=Object;;
Value proposition = <fun> : (unit -> level)
```

2.7.5 Type intro

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathit{Type}(i) : \mathit{Type}(i+1)}$$

```
#let universe num =
#  if num<1 then error "Illegal universe level"
#  else VAL:=Type(num);TYP:=Type(num+1);LEV:=Object;;
Value universe = <fun> : (num -> level)
```

2.7.6 App intro

$$\frac{\Gamma \vdash M : (x:B)C \quad \Gamma \vdash N : A \quad \Gamma \vdash A \equiv B}{\Gamma \vdash (M\ N) : [N/x]C}$$

As explained above, this rule splits into two transitions rules:

```
#(* Stacks a value for future application *)
#let stack_value () =
#    let declaration = Value(Judge(!VAL,!TYP,!LEV))
#    in push_env declaration;;
Value stack_value = <fun> : (unit -> context)

#(* Extracts the last pushed value in environment *)
#let unstack_value () =
#    match !ENV with
#          Value(j)::rest -> ENV:=rest; j
#        | _ -> error "Last item not a value";;
Value unstack_value = <fun> : (unit -> judgement)

#let pop_apply () =
#  match hnftype !TYP with
#     Prod(typarg,typres) ->
#       let (Judge(val,typ,lev)) = unstack_value()
#       in if conv typ typarg then (VAL:=App(!VAL,val);
#                                   TYP:=subst1 val typres)
#          else error "Application would violate typings"
#   | _ -> error "Non-functional construction";;
Value pop_apply = <fun> : (unit -> constr)
```

2.7.7 Discharging definitions

We allow forgetting a constant by replacing its value in the current judgement.

$$\frac{\Gamma; x = M : T \vdash N : A}{\Gamma \vdash [x/M]N : [x/M]A}$$

```
#let pop_const () =
#     match !ENV with
#        [] -> error "No constant in environment"
#      | Constdecl(name,Judge(val,_))::rest ->
#                 VAL:=subst_con name val !VAL;
#                 TYP:=subst_con name val !TYP;
#                 ENV:=rest
#      | _ -> error "Last item not a constant";;
Value pop_const = <fun> : (unit -> context)
```

Variables may be discharged in two ways: for forming abstractions and for forming products. Both rules use as an auxiliary:

```
#let unstack_var () =
#    match !ENV with
#      []                  -> error "No hypothesis in current context"
#    | Vardecl(var)::rest -> ENV:=rest; var
#    | _                  -> error "Last item not a variable";;
Value unstack_var = <fun> : (unit -> string * judgement)
```

2.7.8 Lambda Introduction

$$\frac{\Gamma; x:M \vdash N:P}{\Gamma \vdash [x:M]N:(x:M)P}$$

```
#let abs_var () =
#  let (name,Judge(typ,_)) = unstack_var()
#  in VAL:=Lambda(typ,subst2 name !VAL);
#     TYP:=Prod(typ,subst2 name !TYP);;
Value abs_var = <fun> : (unit -> constr)
```

2.7.9 Product Introduction

Here we have the union of 3 inference rules:

$$\frac{\Gamma; x:M \vdash N:\mathit{Prop}}{\Gamma \vdash (x:M)N:\mathit{Prop}}$$

$$\frac{\Gamma \vdash M:\mathit{Type}(j) \quad \Gamma; x:M \vdash N:\mathit{Type}(i)}{\Gamma \vdash (x:M)N:\mathit{Type}(\mathit{max}(i,j))}$$

$$\frac{\Gamma \vdash M:\mathit{Prop} \quad \Gamma; x:M \vdash N:\mathit{Type}(i)}{\Gamma \vdash (x:M)N:\mathit{Type}(i)}$$

The last two rules appear to be binary. However, they correspond to a unary transition, since the type of M is explicit in the judgement stored in the variable declaration. Here is an example where the rules of inference do not carry enough information: we do not need to use a meta theorem saying that if $\Gamma; x:M$ is a well-formed environment, then either $\Gamma \vdash M:\mathit{Type}(j)$ or $\Gamma \vdash M:\mathit{Prop}$. This fact is actually explicit from the way the variable declarations are structured in the machine.

```
#let gen_var () =
#  if !LEV = Proof then error "Proof objects can only be abstracted";
#  let (name,Judge(typ,kind,_)) = unstack_var()
#  in (TYP:=(match hnftype !TYP with
#         Prop -> Prop (* quantification : TYP stays Prop *)
#       | Type(n) -> (* product *)
#               Type(match kind with Prop -> n | Type(m) -> max m n)
#       | _ -> error "Product allowed only on types");
#      VAL:=Prod(typ,subst2 name !VAL));;
Warning: 1 partial match in this phrase
Value gen_var = <fun> : (unit -> constr)
```

2.7.10 Type conversion

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : Prop \quad \Gamma \vdash A \equiv B}{\Gamma \vdash M : B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : Type(i) \quad \Gamma \vdash A \equiv B}{\Gamma \vdash M : B}$$

As for application, this pair of rules splits into two transition rules:

```
#(* Stacking a type for future coercion *)
#let stack_cast () =
#    let typ = hnftype(!TYP)
#    in let lev = level_of_kind typ
#       in let declaration = Cast(Judge(!VAL,typ,lev))
#    in push_env declaration;;
Value stack_cast = <fun> : (unit -> context)

#let unstack_cast () =
#    match !ENV with
#          []            -> error "No cast in current context"
#        | Cast(j)::rest -> ENV:=rest;j
#        | _             -> error "Last item not a cast";;
Value unstack_cast = <fun> : (unit -> judgement)

#let cast () =
#   let (Judge(typ,_,lev)) = unstack_cast()
#   in if lev<>!LEV then error "Wrong level for coercion"
#      if conv !TYP typ then TYP:=typ
#      else error "Cannot coerce to intended type";;
Value cast = <fun> : (unit -> constr)
```

2.8 Higher-level commands

2.8.1 Sections

The machine is equipped of a hierarchical description mechanism which permits to declare variables and define constants within a local scope. This mechanism is operated through the opening and closing of named *sections*. Variables and constants are declared with a certain *strength* controlling their lifetime as follows. When n sections are opened, the possible strengths of notions are integers between 0 and n. A notion of strength 0 is local to the current section, a notion of greater strength will survive the closing of the current section. When a section is closed, the corresponding segment of the environment is popped and processed as follows. Stacked casts and values are forgotten, with a warning. Local constants disappear by substitution. Local variables are discharged, by abstraction in values, and by product/quantification in types. Thus the variables and constants declared with a positive strength will be kept, under the same name, but with added functionality.

We shall not discuss further this sections mechanism, which would require a more precise explanation, and whose design is not considered definitive yet. For the present paper, let us just consider this structuring mechanism as high-level commands, which could be implemented by macro-generation of the elementary operations of the engine, by keeping separately the strength information. In the current implementation, however, section headers are items in the environment.

2.8.2 Inductive definitions

It is possible to describe complex inductive definitions of types, constructors of those types, and induction/recursion principles on those types. We shall not describe further this facility, which is described in [39], and which ought to be superseded in next versions of the calculus by the addition, at the terms level, of (strongly positive) recursive type definitions.

3 The mathematical vernacular

The constructive engine is generally not operated directly. Its commands are usually compiled from a higher level notation for mathematics, called the *Mathematical Vernacular*. The idea of describing mathematical arguments in a mathematical vernacular language is due to de Bruijn[8].

3.1 A small vernacular syllabus

Our mathematical vernacular is very rudimentary at present. Here is a succinct description of its main constructs.

There are two sub-languages:

- a language of *expressions*, used to denote terms, propositions, types, proofs, etc...
- a language of *commands*, used to write mathematical definitions, postulate axioms and hypotheses, state and proof lemmas and theorems.

3.1.1 Expressions

Names are identifiers in the sense of CAML. Two special identifiers "Prop" and "Type" are reserved.

Expressions are defined recursively as follows:

- names are expressions.
- $Prop$ is an expression.
- $Type(n)$ is an expression, for $n > 0$. $Type$ is an abbreviation for $Type(1)$.
- $(M\ \ N)$ is an expression when M and N are expressions. It stands for "apply function M to its argument N". This notation is generalized to $(M\ \ N1\ \ N2\ \ ...\ \ Np)$.
- $(x : M)N_x$ is an expression if x is a name, and M and N_x are expressions. It stands for the logical proposition "for every x of type M we have N_x" when N_x stands for a proposition, and for the type "product of N_x indexed by M" when N_x stands for a type. This notation is extended to accomodate $(x1, x2, ..., xn : M)N$ as an

abbreviation for $(x1 : M)(x2 : M)...(xn : M)N$. When x does not occur in N, the expression $(x : M)N$ may also be written as $M \to N$.

- $\texttt{[}x : M\texttt{]}\,N_x$ is an expression if x is a name, and M and N_x are expressions. It stands for the function which associates to its argument x of type M the value N_x. This notation is extended to accomodate $\texttt{[}x1, x2, ..., xn : M\texttt{]}\,N$ as an abbreviation for $\texttt{[}x1 : M\texttt{]}\,\texttt{[}x2 : M\texttt{]}\,...\,\texttt{[}xn : M\texttt{]}\,N$.
- *let* $x = M$ *in* N_x is an expression if x is a name, M and N_x are expressions. It stands for the expression N_x, in which every occurrence of the name x is replaced by the expression M.

3.1.2 Commands

A mathematical text is seen as a list of commands to be executed. The type-checking of expressions that are arguments to commands ensures that the definitions make sense, and that the proofs prove the statements of theorems.

- Hypothesis x : T
  defines a new hypothesis x of type the expression T
- Variable x : T
  variant of the previous command
- Hypotheses x,y,z : T
  for multiple declarations sharing the type T
- Variables x,y,z : T
  idem
- Axiom x : A
  the proposition A is assumed as an axiom named x
- Type x
  Abbreviates Variable x : Type
- Proposition x
  Abbreviates Variable x : Proposition
- Definition x M
  Checks that M is well-typed, and defines it under name x
- Name x M
  Local version of Definition
- Global x : T = M
  Checks that M is well-typed consistently with type T, and defines it under name x
- Let x : T = M
  Local version of Global

- Theorem x P Proof M
  Checks that M is a proof of proposition P, and defines it under name x
- Lemma x P Proof M
  Local version of Theorem. These composite commands may be broken into parts, such as:
  Lemma x
  Statement P
  Proof M
  This permits to introduce definitions and facts local to the proof.
- Section x
  Opens a section named x
- End x
  Closes the section x

Other commands are useful for interactive use. We shall not list them here.

We do not explain in this note how the vernacular commands get parsed and translated as elementary commands of the machine. The details are relatively straightforward.

3.2 A full exemple: Newman's lemma

We give here the listing of a full exemple as written in the current system. Newman's lemma states that every nœtherian relation is confluent if it is locally confluent. The proof is the same as the one explained in [25], and given in an older version of the calculus in [15]. A discussion of the definition of induction principles in the Calculus of Constructions may be found in [27].

This section of vernacular assumes prior loading of a standard prelude, defining logical primitives.

We start with an auxiliary module of definitions concerning binary relations, where notions such as transitive-reflexive closure `Rstar` of relation `R` are defined.

```
  Variable A : Type.
  Variable R : A->A->Prop.

(* Definition of the reflexive-transitive closure R* of R *)
(* Smallest reflexive P containing R o P *)

Definition Rstar [x,y:A](P:A->A->Prop)
   ((u:A)(P u u))->((u:A)(v:A)(w:A)(R u v)->(P v w)->(P u w)) -> (P x y).

Theorem Rstar_reflexive  (x:A)(Rstar x x)
 Proof [x:A][P:A->A->Prop]
       [h1:(u:A)(P u u)][h2:(u:A)(v:A)(w:A)(R u v)->(P v w)->(P u w)]
       (h1 x).

Theorem Rstar_R (x:A)(y:A)(z:A)(R x y)->(Rstar y z)->(Rstar x z)
```

```
 Proof [x:A][y:A][z:A][t1:(R x y)][t2:(Rstar y z)]
       [P:A->A->Prop]
       [h1:(u:A)(P u u)][h2:(u:A)(v:A)(w:A)(R u v)->(P v w)->(P u w)]
       (h2 x y z t1 (t2 P h1 h2)).

(* We conclude with transitivity of Rstar : *)

Theorem  Rstar_transitive
         (x:A)(y:A)(z:A)(Rstar x y)->(Rstar y z)->(Rstar x z)
 Proof  [x:A][y:A][z:A][h:(Rstar x y)]
        (h ([u:A][v:A](Rstar v z)->(Rstar u z))
           ([u:A][t:(Rstar u z)]t)
           ([u:A][v:A][w:A][t1:(R u v)][t2:(Rstar w z)->(Rstar v z)]
            [t3:(Rstar w z)](Rstar_R u v z t1 (t2 t3)))).

(* Another characterization of R* *)
(* Smallest reflexive P containing R o R* *)

Definition Rstar' [x:A][y:A](P:A->A->Prop)
    ((P x x))->((u:A)(R x u)->(Rstar u y)->(P x y)) -> (P x y).

Theorem Rstar'_reflexive (x:A)(Rstar' x x)
 Proof  [x:A][P:A->A->Prop]
        [h1:(P x x)][h2:(u:A)(R x u)->(Rstar u x)->(P x x)]h1.

Theorem Rstar'_R (x:A)(y:A)(z:A)(R x z)->(Rstar z y)->(Rstar' x y)
 Proof  [x:A][y:A][z:A][t1:(R x z)][t2:(Rstar z y)]
        [P:A->A->Prop][h1:(P x x)]
        [h2:(u:A)(R x u)->(Rstar u y)->(P x y)](h2 z t1 t2).

(* Equivalence of the two definitions: *)

Theorem Rstar'_Rstar  (x:A)(y:A)(Rstar' x y)->(Rstar x y)
 Proof  [x:A][y:A][h:(Rstar' x y)]
        (h Rstar (Rstar_reflexive x) ([u:A](Rstar_R x u y))).

Theorem Rstar_Rstar'  (x:A)(y:A)(Rstar x y)->(Rstar' x y)
 Proof  [x:A][y:A][h:(Rstar x y)](h Rstar' ([u:A](Rstar'_reflexive u))
         ([u:A][v:A][w:A][h1:(R u v)][h2:(Rstar' v w)]
          (Rstar'_R u w v h1 (Rstar'_Rstar v w h2)))).
```

We are now ready to develop the standard notions from rewriting theory. `<A>Ex2(P,Q)>` means there exists `x:A` such that `(P x)` and `(Q x)`.

```
Definition coherence [x:A][y:A]<A>Ex2((Rstar x),(Rstar y)).

Theorem coherence_intro  (x:A)(y:A)(z:A)(Rstar x z)->(Rstar y z)->(coherence x y)
```

```
 Proof  [x:A][y:A][z:A][h1:(Rstar x z)][h2:(Rstar y z)]
        [C:Prop][h:(w:A)(Rstar x w)->(Rstar y w)->C](h z h1 h2).

(* A very simple case of coherence : *)

Lemma Rstar_coherence (x:A)(y:A)(Rstar x y)->(coherence x y)
 Proof  [x:A][y:A][h:(Rstar x y)]
        (coherence_intro x y y h (Rstar_reflexive y)).

(* coherence is symmetric *)
Lemma coherence_sym (x:A)(y:A)(coherence x y)->(coherence y x)
 Proof  [x:A][y:A][h:(coherence x y)]
         (h (coherence y x)
            ([w:A][h1:(Rstar x w)][h2:(Rstar y w)]
                  (coherence_intro y x w h2 h1))).

Definition confluence
     [x:A](y:A)(z:A)(Rstar x y)->(Rstar x z)->(coherence y z).

Definition local_confluence
     [x:A](y:A)(z:A)(R x y)->(R x z)->(coherence y z).

Definition noetherian
     (x:A)(P:A->Prop)((y:A)((z:A)(R y z)->(P z))->(P y))->(P x).

Section Newman.

(* The general hypotheses of the theorem *)

Hypothesis Hyp1:noetherian.
Hypothesis Hyp2:(x:A)(local_confluence x).

(* The induction hypothesis *)

Section Ind.
  Variable    x:A.
  Hypothesis  hyp_ind:(u:A)(R x u)->(confluence u).

(* Confluence in x *)

  Variables   y,z:A.
  Hypothesis  h1:(Rstar x y).
  Hypothesis  h2:(Rstar x z).

(* particular case x->u and u->*y   *)
Section Newman_.
```

```
  Variable    u:A.
  Hypothesis  t1:(R x u).
  Hypothesis  t2:(Rstar u y).

(* In the usual diagram, we assume also x->v and  v->*z   *)

Theorem Diagram.
  Variable v:A.
  Hypothesis u1:(R x v).
  Hypothesis u2:(Rstar v z).

Statement (coherence y z).
Proof     (* We draw the diagram ! *)
  (Hyp2 x u v t1 u1
        (coherence y z)              (* local confluence in x for u,v *)
                                     (* gives w, u->*w and v->*w *)
        ([w:A][s1:(Rstar u w)][s2:(Rstar v w)]
  (hyp_ind u t1 y w t2 s1            (* confluence in u => coherence(y,w) *)
        (coherence y z)              (* gives a, y->*a and z->*a *)
          ([a:A][v1:(Rstar y a)][v2:(Rstar w a)]
  (hyp_ind v u1 a z                  (* confluence in v => coherence(a,z) *)
      (Rstar_transitive v w a s2 v2) u2    (* gives b, a->*b and z->*b *)
        (coherence y z)
        ([b:A][w1:(Rstar a b)][w2:(Rstar z b)]
  (coherence_intro y z b (Rstar_transitive y a b v1 w1) w2))))))).

Theorem caseRxy (coherence y z)
Proof (Rstar_Rstar' x z h2
      ([v:A][w:A](coherence y w))
      (coherence_sym x y (Rstar_coherence x y h1))  (* case x=z *)
      Diagram).                                     (* case x->v->*z *)
End Newman_.

Theorem Ind_proof (coherence y z)
Proof (Rstar_Rstar' x y h1 ([u:A][v:A](coherence v z))
        (Rstar_coherence x z h2)                    (* case x=y*)
        caseRxy).                                   (* case x->u->*z *)
End Ind.

Theorem Newman (x:A)(confluence x)
Proof [x:A](Hyp1 x confluence Ind_proof).

End Newman.
```

The complete proof-checking of this example takes 4 seconds on a SUN 3-60 in the current version 4.9 of our implementation of the calculus, using CAML version 2.6.

We would like to stress the fact that this seemingly purely declarative piece of mathematics is actually compiled and run on the constructive engine. This is an extremely concrete illustration that "the mathematician builds the universe he lives in."

Conclusion

We have presented here only a facet of our implementation of the calculus. Other uses of the system are possible. For instance, an extractor computes the information contents of proofs with special annotations. A theorem-prover, based on the proper adaptation of LCF tactics, searches for proofs under the user's guidance. The incorporation of the tactics language in the vernacular would be a first step towards a completely formal description of mathematical theories in a language close to the usual mathematics practice. Extensions of the calculus are considered, principally recursive types.

We have given in this note a rather complete description of the basic bloc, the Constructive Engine. The algorithms presented are directly executable CAML programs, taken from the actual implementation. This is a first attempt at using CAML as a publication language.

References

[1] R.S. Boyer, J Moore. "The sharing of structure in theorem proving programs." Machine Intelligence **7** (1972) Edinburgh U. Press, 101–116.

[2] 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.

[3] 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.

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

[5] N.G. de Bruijn. "Some extensions of Automath: the AUT-4 family." Internal Automath memo M10 (Jan. 1974).

[6] N.G. de Bruijn. "A namefree lambda calculus with facilities for internal definition of expressions and segments." TH Report 78-WSK-03, Technological University Eindhoven, Aug. 1978.

[7] 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).

[8] N.G. de Bruijn. "Formalizing the Mathematical Vernacular". Unpublished memo, Technological University Eindhoven, July 1982.

[9] N.G. de Bruijn. "Formalization of constructivity in AUTOMATH." in: Papers dedicated to J.J. Seidel, Ed. P.J. de Doelder, J. de Graaf and J.H. van Lint (1984)

[10] N.G. de Bruijn. "Generalizing Automath by means of a lambda-typed lambda calculus." Proceedings, Maryland 1984-1985 Special Year in Mathematical Logic and Theoretical Computer Science.

[11] A. Church. "A formulation of the simple theory of types." Journal of Symbolic Logic **5,1** (1940) 56–68.

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

[13] Th. Coquand. "An analysis of Girard's paradox." First IEEE Symposium on Logic in Computer Science, Boston (June 1986), 227–236.

[14] Th. Coquand. "Metamathematical Investigations of a Calculus of Constructions." Submitted to Theoretical Computer Science.

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

[16] Th. Coquand and G. Huet. "The Calculus of Constructions." To appear, Information and Control.

[17] Th. Coquand and G. Huet. "Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions." Logic Colloquium, Orsay (July 85). To appear, North-Holland.

[18] G. Cousineau, G. Huet. "The CAML Primer, Version 2.6." Projet Formel, INRIA-ENS, March 1989.

[19] D. Van Daalen. "The language theory of Automath." Ph. D. Dissertation, Technological Univ. Eindhoven (1980).

[20] P. J. Downey, R. Sethi, R. Tarjan. "Variations on the common subexpression problem." JACM **27,4** (1980) 758–771.

[21] J.Y. Girard. "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure." Thèse d'Etat, Université Paris VII (1972).

[22] R. Harper, F. Honsell and G. Plotkin. "A Framework for Defining Logics." Second LICS, Ithaca, June 1987.

[23] R. Harper and R. Pollack. "Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions." CCIPL, TAPSOFT'89, Barcelona, March 1989.

[24] W. A. Howard. "The formulæ-as-types notion of construction." Unpublished manuscript (1969). Reprinted in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds J. P. Seldin and J. R. Hindley, Academic Press (1980).

[25] G. Huet. "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems." J. Assoc. Comp. Mach. **27,4** (1980) 797–821.

[26] G. Huet. "Formal Structures for Computation and Deduction." Course Notes, Carnegie-Mellon University, May 1986.

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

[28] G. Huet. "Extending the Calculus of Constructions with Type:Type." Unpublished notes, Feb. 1987.

[29] L.S. van Benthem Jutting. "The language theory of Λ_∞, a typed λ-calculus where terms are types." Unpublished manuscript (1984).

[30] G. Kahn. "Natural Semantics." In Programming of Future Generation Computers, eds K. Fuchi & M. Nivat, North-Holland 1988.

[31] J.J. Lévy. "Optimal Reductions in the Lambda Calculus." in To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press (1980), 159–191.

[32] Z. Luo. "ECC, an Extended Calculus of Constructions." Fourth LICS, Asilomar, June 1989.

[33] P. Martin-Löf. "A Theory of Types." Report 71-3, Dept. of Mathematics, University of Stockholm, Feb. 1971, revised (Oct. 1971).

[34] P. Martin-Löf. "An intuitionistic theory of types: predicative part." Logic Colloquium, North-Holland (1975).

[35] P. Martin-Löf. "Intuitionistic Type Theory." Studies in Proof Theory, Bibliopolis (1984).

[36] P. Martin-Löf. "Truth of a proposition, evidence of a judgement, validity of a proof." Transcript of talk at the workshop "Theories of Meaning", Centro Fiorentino di Storia e Filosofia della Scienza, Villa di Mondeggi, Florence (June 1985).

[37] C. Mohring. "Algorithm Development in the Calculus of Constructions." IEEE Symposium on Logic in Computer Science, Cambridge, Mass. (June 1986).

[38] C. Mohring. "Extracting F_ω'programs from proofs in the Calculus of Constructions." Sixteenth Symposium on Principles of Programming Languages, Austin, Jan. 89.

[39] C. Mohring. "Extraction de programmes dans le Calcul des Constructions." Thèse d'Informatique, Université Paris 7, Janv. 89.

[40] G. Nelson, D.C. Oppen. "Fast decision procedures based on congruence closure." JACM **27,2** (1980) 356–364.

[41] B. Russel and A.N. Whitehead. "Principia Mathematica." Volume 1,2,3 Cambridge University Press (1912).

[42] D. Scott. “Constructive validity.” Symposium on Automatic Demonstration, Springer-Verlag Lecture Notes in Mathematics, **125** (1970).

[43] C. Wadsworth. “Semantics and Pragmatics of the Lambda-Calculus”. Ph.D. thesis, Oxford University, Sept. 1971.

[44] P. Weis et al. “The CAML Reference Manual, Version 2.6.” Projet Formel, INRIA-ENS, March 1989.