The calculus of constructions. Documentation and users's guide. Version 4.10

Projet Formel

▶ To cite this version:

Projet Formel. The calculus of constructions. Documentation and users's guide. Version 4.10. RT-0110, INRIA. 1989, pp.203. inria-00070056

HAL Id: inria-00070056
https://inria.hal.science/inria-00070056

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports Techniques

N° 110

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

THE CALCULUS OF CONSTRUCTIONS

Documentation and users's guide

Version 4.10

Projet FORMEL

Août 1989

The Calculus of Constructions

Documentation and users's guide

Version 4.10

Projet Formel

INRIA-ENS

July 1st 1989

Second Printing July 1st, 1989

©INRIA 1989

Résumé

Ce rapport technique rassemble divers documents constituant la documentation du Calcul des Constructions, dans sa version 4.10.

Abstract

This document is a collection of papers gathered as a documentation guide for the Calculus of Constructions, in its version 4.10. It contains:

- An article "Meta-mathematical investigations of a Calculus of Constructions", by Thierry Coquand, which describes theoretical results about various versions of the calculus.
- An article "Extracting $F\omega$'s Programs from Proofs in the Calculus of Constructions", by Christine Paulin-Mohring, which describes the computational interpretation of the calculus, and its use as a program extractor.
- An article "The Constructive Engine", by Gérard Huet, which describes a simplified version of the implementation.
- A technical note "The Vernacular Syllabus", by Gilles Dowek, describing the language of the user interface.
- A technical note "Inductive Definitions in the Calculus of Constructions", by Christine Paulin-Mohring, describing how to define inductive notions in the vernacular.
- A technical note "The Tactics Theorem Prover", by Thierry Coquand, explaining how to develop proofs with computer assistance.
- A technical note "A short user's guide for the Constructions Version 4.10", by Gérard Huet.
- A library of commented examples.

How to read this document

If you are interested in the basic theory, read the first two articles. If you are interested in using the system, read first the short user guide, then the vernacular syllabus and the theorem-prover guide. Try out examples from the library, then develop your own. If you are interested in the implementation, read the constructive engine paper.

The Calculus of Constructions implementation is available free of charge for universities and public research centers. Send requests to huet@inria.inria.fr. In order to run the system, you need CAML version 2.6. Send requests for CAML to chenetier@ilog.fr.

Metamathematical Investigations of a Calculus of Constructions

Thierry Coquand

INRIA

Introduction

In this paper, we present a theory of constructions for higher-order intuitionistic logic. The original inspirations were Martin-Löf Intuitionistic Theory of Types, Girard's system $F\omega$, de Bruijn's Automath, and some ideas of Huet about a higher-order notion of Horn clauses. A first version of this calculus appeared in [17] and an early implementation, written by G. Huet in CAML, is presented in [20,21].

After general motivations, we give a formal presentation of the present system of constructions. We present a skectch of different semantics. The third part is a survey of the main properties of this calculus (and connections with other logics). Finally, we discuss some possible extensions.

1 Informal motivation

A first attempt at formulating a system aiming at a representation of the full intuitionistic theory of finite types was Martin-Löf's first version of his theory of types [58]. However, this theory was based on the strongly impredicative axiom that there is a type of all types, and this assumption was shown to be inconsistent by J.-Y. Girard [30]. This discovery led Martin-Löf to formulate theories that do not contain any more intuitionistic simple type theory. We will present here a version of type theory that expresses naturally Heyting's semantics of intuitionistic higher-order logic.

Let us recall first what were Martin-Löf's motivation for having a type of all types. Three principles entail this idea.

- we want quantification over predicates and propositions,
- Russell's doctrine of types: the range of significance of a propositional function forms a type,
- the identification of propositions and types.

Indeed, by the first and second points, the collection of all propositions must form a type. If propositions and types are identified, then this type is also the type of all types.

The idea here will be to keep the first and the second principles, but to restrict the third point. We retain only the idea that to each proposition corresponds a certain type, namely, the type of its proofs.

A proposition will be characterised by the way it can be proved. The proofs of a given proposition are to be considered as mathematical objects (cf. the "proof-as-objects" paradigm of [10]), and they must form a type, which is the type of proofs of the given proposition. To any proposition φ is thus associated the "type" of its proofs $\mathsf{T}(\varphi)$. Furthermore, the possibility of doing quantification over propositions, predicates, forces the introduction a type Prop of propositions. Heyting's semantics of the universal quantification can then be expressed by: if A is a type, φ a propositional function over A, that is an object of type $A \rightarrow \mathsf{Prop}$, to give a proof of the proposition $(\forall x : A)\varphi(x)$ is to give a function f such that for any object a of type A, $f(a)$ is a proof of $\varphi(a)$. Thus, $\mathsf{T}((\forall x : A)\varphi(x))$ has to be isomorphic to the product $(x : A)\mathsf{T}(\varphi(x))$ of the dependent family of type $\mathsf{T}(\varphi(x))$ over A.

Let us define $\varphi \Rightarrow \psi$ as $(\forall x : \mathsf{T}(\varphi))\psi$. A quick study of the type system we get shows that it contains the second-order calculus of Girard-Reynolds. For example, the generic identity will be the term $(\lambda A : \mathsf{Prop})(\lambda x : \mathsf{T}(A))x$ of type $(A{:}\mathsf{Prop})\mathsf{T}(A) \rightarrow \mathsf{T}(A)$. It contains also naturally an intuitionistic version of simple type theory [13].

The nice point is that we do not need to state any further axioms or deduction rules. The logic becomes simply the expression of the fact that a proposition is true if, and only if, its corresponding type of proofs is inhabited.

We have thus extended Howard's theory of construction to an intuitionistic version of higher-order logic [43,13]. In particular, this system leads to some new facts about the interpretation of the connector $\exists$, problem mentioned in [43], and will allow us to discuss the problem of semantics of evidence of classical second-order logic.

2 The system (terms and typing rules)

2.1 Formal presentation

The terms have the following inductive structure.

1. identifiers,
2. the special constant Prop,
3. typed abstractions $(\lambda x : M)N$ and hypothetical formation $(\Lambda x{:}M)N$,
4. product, that we shall write $(x : M)N$, where M and N are terms, and universal quantification $(\forall x{:}M)N$,
5. application $M(N)$ and instantiation $\mathsf{app}(M, N)$ where M and N are terms.
6. type formation $\mathsf{T}(M)$, where M is a term.

We adopt the following notations: we write $M(M_1, ..., M_p)$ for $M(M_1)...(M_p)$. If the variable x does not appear free in N, we write $(M)N$ for $(x{:}M)N$ [1].

[1] We have to say a word about the problem of bound variables. We propose the following solution: use De Bruijn indexes (cf. [8]) for bound variables, but keep identifiers for free variables as in [18].

An *assignment*, or *context* is a list of pairs (x,t) where t is a term, and x an identifier. We shall write $\mathcal{V}(B)$ for the set of identifiers which appear in an assignment B. We define the following ordering relation on assignments: $B \subset B'$ which means that if (x,t) appears in B, then it appears in B' (i.e. the set which corresponds to B is a subset of the one which corresponds to B').

We define simultaneously what is a valid assignment, and when a term M is well-formed in an assignment B of type N, relation which is written $M : N\ [B]$. If $x \in \mathcal{V}(B)$, we denote by B_x the term such that (x, B_x) appears in B, and that is the first occurrence in the list B of the form (x,t). It should be noted that the definition of the entailment relation that follows is an instance of an inductive definition. It is a calculus of derivations of judgements of the three possible forms B is valid, T type $[B]$ and $M : T\ [B]$.

2.2 Type Inference Rules

We have first the axiom Prop type $[B]$, if B is a valid assignment. The second axiom is that $x : B_x\ [B]$, if B is valid, and x is an identifier which appears in B. We have then the rule of context formation.

$$\frac{M \text{ type } [B]}{B, x : M \text{ is valid}}$$

In this rule, we must impose on x that it is an identifier which does not appear in B, and B may be empty.

$$\frac{M \text{ type } [B] \qquad N \text{ type } [B, x : M]}{(x{:}M)N \text{ type } [B]}$$

$$\frac{\varphi : \mathsf{Prop}\ [B]}{\mathsf{T}(\varphi) \text{ type } [B]}$$

$$\frac{P \text{ type } [B] \qquad Q \text{ type } [B, x : P] \qquad t : Q\ [B, x : P]}{(\lambda x : P)t : (x{:}P)Q\ [B]}$$

$$\frac{M \text{ type } [B] \qquad \varphi : \mathsf{Prop}\ [B, x : M]}{(\forall x : M)\varphi : \mathsf{Prop}\ [B]}$$

$$\frac{A \text{ type } [B] \qquad \varphi : \mathsf{Prop}\ [B, x : A] \qquad b : \mathsf{T}(\varphi)\ [B, x : A]}{\Lambda x : A.b : \mathsf{T}((\forall x : A)\varphi)\ [B]}$$

$$\frac{A \text{ type } [B] \qquad P \text{ type } [B, x : A] \qquad c : (x{:}A)P\ [B] \qquad a : A\ [B]}{c(a) : [a/x]P\ [B]}$$

$$\frac{A \text{ type } [B] \qquad \varphi : \mathsf{Prop}\ [B, x{:}A] \qquad t : \mathsf{T}((\forall x : A)\varphi)\ [B] \qquad a : A\ [B]}{\mathsf{app}(t, a) : \mathsf{T}([a/x]\varphi)\ [B]}$$

$$\frac{t : P\ [B] \qquad P = Q\ [B]}{t : Q\ [B]}$$

Here the equality used in $P = Q\ [B]$ is β-conversion, which can be defined directly at a syntactic level on the raw terms. The need for conversion rules is for instance explained in [61,83].

We can define as well β-reduction, and then prove it is Church-Rosser by Tait's method (see [58]). This is used in the proof of the next result.

Lemma: (uniqueness of products) if $(\forall x : A_1)\varphi_1 = (\forall x : A_2)\varphi_2 : \mathsf{Prop}\ [B]$ then $A_1 = A_2\ [B]$ and $\varphi_1 = \varphi_2 : \mathsf{Prop}\ [B, x : A_1]$.

The following propositions are proved by induction. The meta-variables $E, E', \ldots$ will be used to denote arbitrary judgements.

Proposition 1: If B, B' are valid, $B \subset B'$, $E\ [B]$ then $E\ [B']$.

This allows us to consider assignment as sets, and not as lists.

Proposition 2: If $M : P\ [B]$, then P type $[B]$.

Proposition 3: If $M : P\ [B]$, and $M = N : P\ [B]$, then $N : P\ [B]$.

Proposition 4: If $M : P\ [B]$, $M : Q\ [B]$ then $P = Q\ [B]$.

From this, we derive that any judgement has at most one derivation, if we identify terms up to β-conversion.

This expresses the "uniqueness" of types of a well-formed term (though this uniqueness is only modulo conversion).

Definition 3: Let B be a valid assignment, a *type* (of B) is a term M in B such that M **type** $[B]$. A *proposition* of B is a term φ such that $\varphi : \mathsf{Prop}\ [B]$. A *small type* is a type of proofs of some proposition, that is a type of the form $\mathsf{T}(\varphi)$, with $\varphi : \mathsf{Prop}$. If φ is a proposition of B, a *proof* of φ is a term M such that $M : \mathsf{T}(\varphi)\ [B]$, and we say that φ is *true* or *valid* in B if, and only if, φ has a proof.

One may think of Prop as a type of names of small types. Notice that the small types are intuitively closed by product over families of small types. If A is a type, and $B : (A)\mathsf{Prop}$, that is B is a family over A of names of small types, then the product $(x : A)\mathsf{T}(B(x))$ is intensionally isomorphic to the small type $\mathsf{T}(\forall x : A.B(x))$. It should then be clear from this remark in what way our system contains the second-order λ-calculus of Girard-Reynolds.

Let us give some examples to illustrate this point. We write Void for $(\forall\alpha : \mathsf{Prop})\alpha$, so that $\mathsf{Void} : \mathsf{Prop}$ (and we shall see that $\mathsf{T}(\mathsf{Void})$ is indeed empty). We define $\varphi \rightarrow \psi$ as being $(\forall x : \mathsf{T}(\varphi))\psi$, so that $\varphi \rightarrow \psi : \mathsf{Prop}$ if $\varphi : \mathsf{Prop}$ and $\psi : \mathsf{Prop}$. We write Unit for $(\forall\alpha : \mathsf{Prop})\alpha \rightarrow \alpha$, so that $\mathsf{Unit} : \mathsf{Prop}$ (and it is possible to show that the only element of type $\mathsf{T}(\mathsf{Unit})$ is $id = (\Lambda\alpha : \mathsf{Prop})(\Lambda x : \mathsf{T}(\alpha))x$, that is, the polymorphic identity).

Finally, the following notion is the expression of intuitionistic consistency.

Definition 4 We say that a valid context B is *consistent* if, and only if there exists a proposition φ in B which is not provable in B.

2.3 Some syntactic remarks

As we noticed already, our presentation is not minimal, and we have to relate it to the more concise presentations of [17,20] (see below). Mainly since the work of [86,78], we understand better the fact that we can present typed calculus in so many different ways. The important distinction is between *derivations* and *judgement.* A sound requirement for a calculus is that any judgement has at most one derivation.

This is the case here, provided we identify terms up to β-conversion, and this was also the case for the formalisms presented in [17,20].

Furthermore, the calculus presented here is more explicit than the one in [17,20], so that there is a "forgetting map" between derivations in the present version in derivations of [17,20]. It is then possible to define by induction on the derivations a "section" of this forgetting map which is an interpretation of the implicit calculi in the explicit system, and this defines as well an interpretation of implicit judgement in explicit judgements. The best way to give a semantics of the implicit calculi, is then first to define a syntactic interpretation of the implicit calculi in the explicit calculus, and then the semantics of the explicit calculus. See [86] for details. There are potential problems if we take untyped λ-abstraction (see the examples in [78]).

Notice that the Church-Rosser property plays a crucial role when we prove that any judgement has at most one derivation (in the uniqueness of products lemma above). All the troubles come because our syntax for terms is still not explicit enough. In general as explained in [86], the terms have to be still more explicit, i.e. we have to put explicitly types in the application. The syntax may even be made explicit in such a way that terms become identical to derivations, up to conversions [78]. Of course, the syntax becomes then more cumbersome, since the application becomes now something like $\mathsf{app}(A, \lambda x : A.B, t, u)$. However, for theoretical study, this calculus has to be preferred, and the Church-Rosser property is not needed any more at this level. It will be needed when we try to translate from the implicit system in the explicit one, since we give a translation of derivations, and the Church-Rosser property seems then crucial to establish that a given judgement has at most one derivation, see [86].

Whenever there is no possible confusion, we write φ for $\mathsf{T}(\varphi)$ if $\varphi : \mathsf{Prop}$, and we identify the two λ-abstractions and the two applications. We will use the alternative notation $A \to B$ for $(A)B$, with A, B types, and $\varphi \Rightarrow \psi$ for $(\forall x : \mathsf{T}(\varphi))\mathsf{T}(\psi)$, with $\varphi, \psi : \mathsf{Prop}$. We consider also informally that Prop is included in the collection of types, so that we can talk of a small type for something which is of type Prop. There is a similar "abus de langage" in category theory with the notion of small complete category (see [41]). A final remark: the work of [86,78] may be seen as an indication that all these "abus de langage" are more subtle than they seem, and need a careful meta-mathematical study. Essentially, the problem is that when we use such facilities, we are working in an implicit system in which the uniqueness property of derivations w.r.t. judgements may fail. We will however not carry this discussion any longer here, and limit ourselves to indicating this difficulty.

Thus, we suppose that we are working with the simplified system: first $B \vdash x : B_x$ and Prop type $[B]$ if B is valid, and

$$\frac{A \ \mathsf{type}\ [B]}{B, x : A \ \text{ is valid}} \qquad \frac{A : \mathsf{Prop}\ [B]}{B, x : A \ \text{ is valid}}$$

$$\frac{P \ \mathsf{type}\ [B, x : A]}{(x : A)P \ \mathsf{type}\ [B]} \qquad \frac{P : \mathsf{Prop}\ [B, x : A]}{(x : A)P : \mathsf{Prop}\ [B]}$$

$$\frac{M : P\ [B, x : A]}{\lambda x : A.M : (x : A)P\ [B]} \qquad \frac{N : (x : A)P\ [B] \quad M : A\ [B]}{N(M) : [M/x]P\ [B]}$$

$$\frac{M : P\ [B] \quad Q \text{ type } [B] \quad P{=_\beta}Q}{M : Q\ [B]} \qquad \frac{M : P\ [B] \quad Q : \mathsf{Prop}\ [B] \quad P{=_\beta}Q}{M : Q\ [B]}$$

As explained above, there is a syntactic translation of this calculus into the more explicit calculus, built by induction on the derivation. This justifies to use the more explicit version when we give the semantics, and to use (and implement) the more implicit version in the examples.

We have just seen that we can think of Prop as a collection of small types. The type Prop can thus be thought of as a "universe" of Martin-Löf type theory. The essential difference here is that this universe is closed under any products, not only products over small types. For instance, $(X : \mathsf{Prop})X$ is still a small type. The meaning of $(X : \mathsf{Prop})X$ involves a typical circularity: it is an object of type Prop but also defined by a quantification over Prop (see [76]). In particular, we cannot think of small types as sets ([75]).

Despite this fact, we have, by a suitable generalisation of Girard's argument [30,17,48].

Proposition 5: The calculus of constructions is strongly normalisable.

Corollary: the judgements B is valid, P type $[B]$, $M : P\ [B]$ are decidable.

See the arguments of [58] for a proof of the corollary.

It is possible to define a system in which we restrict the product formation only for a family of small types only over a small types. The axiom for $\forall$ becomes then

$$\frac{M : \mathsf{Prop}\ [B] \qquad \varphi : \mathsf{Prop}\ [B, x : M]}{(\forall x : \mathsf{T}(M))\varphi : \mathsf{Prop}\ [B]}$$

$$\frac{A : \mathsf{Prop}\ [B] \qquad \varphi : \mathsf{Prop}\ [B, x : \mathsf{T}(A)] \qquad b : (x{:}\mathsf{T}(A))\mathsf{T}(\varphi)\ [B]}{\Lambda(b) : \mathsf{T}((\forall x : \mathsf{T}(A))\varphi)\ [B]}$$

The system we get is then very close to some Automath systems [7,36].

We will need also the following examples (see [6,20]), given in the implicit syntax.

$$\begin{array}{rcl}
\bot &=& (A{:}\mathsf{Prop})A \\
\neg(A) &=& A{\to}\bot \quad [A : \mathsf{Prop}] \\
\mathsf{Bool} &=& (A{:}\mathsf{Prop})A{\to}A{\to}A \\
\mathsf{true} &=& \lambda A{:}\mathsf{Prop}.\lambda x, y{:}A.x \\
\mathsf{false} &=& \lambda A{:}\mathsf{Prop}.\lambda x, y{:}A.y \\
\mathsf{Nat} &=& (A{:}\mathsf{Prop})A{\to}(A{\to}A){\to}A \\
0 &=& \lambda A{:}\mathsf{Prop}.\lambda x{:}A.\lambda f{:}A{\to}A.x \\
S &=& \lambda n{:}\mathsf{Nat}.\lambda A{:}\mathsf{Prop}.\lambda x{:}A.\lambda f{:}A{\to}A.f(n(A,x,f)) \\
\mathsf{Eq}(A,x,y) &=& (P{:}A{\to}\mathsf{Prop})P(y){\Rightarrow}P(x) \quad [A \text{ type}, x, y : A] \\
A \times B &=& (C{:}\mathsf{Prop})(A{\to}B{\to}C){\to}C \quad [A, B : \mathsf{Prop}] \\
(x,y) &=& \lambda C{:}\mathsf{Prop}.\lambda z{:}A{\to}B{\to}C.z(C,x,y) \quad [A, B : \mathsf{Prop}, x : A, y : B] \\
\pi_1(z) &=& z(A, \lambda x{:}A.\lambda y{:}B.x) \quad [A, B : \mathsf{Prop}, z : A \times B] \\
\pi_2(z) &=& z(B, \lambda x{:}A.\lambda y{:}B.y) \quad [A, B : \mathsf{Prop}, z : A \times B]
\end{array}$$

2.4 Relationship with Automath languages

It turns out that the calculus presented here is mainly equivalent to the Automath version of [7] but where we allow quantification over (De Bruijn notion of) "type" as well, and this possibility was actually considered by De Bruijn (in the section 12.6). What follows shows the (maybe) surprising result that we still get a meaningful calculus by allowing this quantification (see also [9]).

A difference is that we did not take η-conversion. However, as noticed in [39], we have the following result:

Proposition: if $\lambda y : A.M(y) : T\ [B]$ and y is not free in M, then $M : T\ [B]$.

For a more precise comparison between all the different systems: Automath, LF, second-order λ-calculus, $\cdots$ we refer to [4].

3 Semantics

We present now a few semantics of the calculus. Besides proving consistency and suggesting some extensions, a semantics may be useful in order to establish independence results. These semantics are defined first for the system with the explicit syntax, and then, by composition of the translation from the implicit syntax to the explicit syntax.

3.1 Proof-irrelevance semantics

A first semantics consists in a classical reading of our rules, which forgets everything about proofs. Types are interpreted as Zermelo-Fraenkel sets, and Prop is interpreted as the two element set $\{0,1\}$. Let $\mathsf{T}(0)$ be the empty set, and $\mathsf{T}(1)$ be one singleton set. Furthermore, the product, the λ-abstraction, the application are interpreted as the ordinary set-theoretic product, function formation and application. Finally, if A is a type, and $\varphi : A \rightarrow \mathsf{Prop}$ $(\forall x : A)\varphi(x)$ is interpreted as 1 if, and only if, $[\![\varphi]\!](u)$ is 1 for every $u \in [\![A]\!]$. In this semantics, $\mathsf{T}(\forall x : A.\varphi(x))$ will not coincide in general with $(x : A)\mathsf{T}(\varphi(x))$. They are in general only sets in one-to-one correspondance (and this justifies the choice of the explicit system).

Notice a subtle point. When we define precisely this semantics, it is given by induction on the *derivation.* So the crucial property that judgements have at most one derivation is needed here. Notice that this argument does not work with a less explicit syntax, for instance in general if we use untyped λ-abstraction, or if we don't use the operators $\mathsf{T}(X)$ (see [78]). See [86] for a solution to the problem of the semantics of the present calculus with untyped abstraction.

This semantics, however simple it is (it can be seen as a higher-order generalisation of the truth-table methods), shows the consistency of our calculus. Indeed, the interpretation of $(\forall\alpha : \mathsf{Prop})\alpha$ is 0, and hence the type $(\alpha : \mathsf{Prop})\mathsf{T}(\alpha)$ cannot be inhabited. This was actually the method used by Gentzen [28] in order to establish the consistency of the simple theory of types. It is completely elementary, that is, it is "finitist" in the sense of Hilbert and can be carried out formally in a logically weak system (for instance, primitive recursive arithmetic). A similar semantics has been used by [84] in order to prove that we cannot derive all Peano axioms

in Martin-Löf intuitionistic theory of types without universes. In our case too, we can use this semantics in order to show that the proposition $\mathsf{Eq}(\mathsf{Bool},\mathsf{true},\mathsf{false})\Rightarrow\perp$ is not inhabited.

The fact that there exist elementary consistency proofs has to be contrasted with the normalisation theorem, which cannot have an elementary proof. This can be understood intuitively: the normalisation theorem will entail not only the consistency of the pure calculus, that is the consistency of the empty context, but also the consistency of non trivial contexts.

As an example, let us consider a context which expresses the "axiom of infinity". It is the context Inf declaring one small type $A : \mathsf{Prop}$, one constant and one unary operation $a : A, f : A \rightarrow A$, one binary relation transitive and antireflexive $R : A \rightarrow A \rightarrow \mathsf{Prop}, h_1 : (x : A)R(x,x) \Rightarrow \perp, h_2 : (x,y,z : A)R(x,y) \Rightarrow R(y,z) \Rightarrow R(x,z)$ with the hypothesis $h_3 : (x : A)R(x, f(x))$.

Proposition: Inf is consistent.

Proof: Using the normalisation theorem, a purely combinatorial argument can be given. We claim that in the context Inf, by induction on the size of the construction in normal form M, if M has for type a proposition $R(u,v)$, then $u = f^n(a)$,$v = f^p(a)$ with $n < p$, and if M is of type A, then M is of the form $f^n(a)$. Indeed, by induction, such a construction will never use h_1.

It results that in the context Inf, $R(f^n(a), f^p(a))$ is provable only if $n < p$ (conversely, if $n < p$, $R(f^n(a), f^p(a))$ is provable). In particular, at least one proposition is not provable and so Inf is consistent.

Heuristically at least, this says that in the context Inf, we can define an infinity of "provably" distinct elements that are $a, f(a), f(f(a)), \cdots$. Formally, we can follow the development of arithmetic as done in [77] where an integer is interpreted as a class of classes over A. It is possible to do such a construction over an arbitrary type, but we cannot in the general case get all Peano axioms, in particular not the axiom that zero differs from a successor, and the axiom that the successor operation is one-to-one. In the context Inf and over the type A, all Peano axioms are provable (see [3] where such a development is done in a classical framework, and from which is inspired the context Inf). This shows, in an elementary way, that the normalisation theorem implies the consistency of higher-order classical arithmetic, and so, by Gödel's theorem, that the normalisation theorem cannot be proved by means of higher-order arithmetic.

Remark: what we have used here (and is thus not elementary provable) is the result a priori weaker than normalisation that if a small type is inhabited, then it is inhabited by a term in normal form. As noticed in another framework by Kreisel [32], this last result may become elementary inside a context. For instance, in any context which contains $x : (A : \mathsf{Prop})A \rightarrow A$, one direct inductive argument shows that any inhabited small type is inhabited by a term in normal form. We can use the variable x to "freeze" the redexes.

3.2 Realisability semantics

A more informative semantics is the realisability semantics. The basic idea behind these models is quite natural: consider this language really as a programming language. At compile time, we forget any type information to get a untyped program. This suggests to start with an untyped universe of combinatory expressions (the "programs") and to interpret a type as a set of such programs.

All this can be formalised. We take for the universe of untyped programs the set U of untyped λ-terms. A small type is interpreted as a subset of U (closed under β-conversion), and an element of a small type is interpreted by the corresponding untyped λ-term we get by forgetting all type informations. The "big" products over a type of a family of small types is interpreted by intersection (here we use impredicativity of set theory: an arbitrary intersection of a family of subsets of a given set is still a subset of this set!). Finally, a product $(x:A)B(x)$ over a small set A is interpreted as the set of λ-terms t such that, if $u \in [\![A]\!]$ then $t(u) \in [\![B(x)]\!][x/u]$ (see [21] for the details, this can be seen as a realisability interpretation of the constructions). For a generic example, $\lambda A : \mathsf{Prop}.\lambda x : A.x$ is interpreted by $\lambda x.x$ and its type $(A:\mathsf{Prop})(A)A$ is interpreted as the set of λ-terms t such that, for any subset A of U (closed under β-conversion), if u is in A, then $t(u) \in A$. This is indeed the case for $t = \lambda x.x$.

This model can be used in order to show the consistency of some extensions of the original calculus. For instance, one may want to add a small type $n : \mathsf{Prop}$, define $N = \mathsf{T}(n)$, together with one constant $O : N$, and one unary operation $S : (N)N$. We also add a recursive operator $\mathsf{rec} : (P : (N)\mathsf{Prop})P(O) \rightarrow ((x : N)P(x) \rightarrow P(S(x))) \rightarrow (x:N)P(x)$ with the new conversion rules

$$\mathsf{rec}(P,a,f)(O) = a \qquad \mathsf{rec}(P,a,f)(S(x)) = f(x, \mathsf{rec}(P,a,f)(x)).$$

It is possible to interpret this extension in the realisability model (but this was already possible, in a trivial way, in the proof-irrelevance model). The simplest way to see it is to add primitive operations to the "untyped programming languages" (i.e. to extend U with appropriate constants and δ rules). Furthermore, in this model, we can check that all Peano axioms are satisfied (and this does not hold in the proof-irrelevance model, where for instance $\mathsf{Eq}(N,O,S(O)) \Rightarrow \perp$ is not provable). Hence, we have shown the consistency of the extension of the pure calculus with Peano axioms.

We get a "more mathematically civilised" presentation of this model by using the notion of D-set of E. Moggi. Intuitively, we start with an arbitrary combinatory algebra D instead of the combinatory algebra of untyped λ-terms, and we take partial equivalence relation instead of arbitrary subsets (see [86,25]).

This semantics has been generalised to universes in [55].

Finally, let us mention that interesting independence results have been obtained by T. Streicher using variations of the realisability models, for building, for instance, models without strong sums of families of small types over a small type [86].

3.3 Model in domain theory

Domain theory has been developped in order to give a semantics of simply typed λ-calculus with fixed-point [80] (and it was realised later that it could be used also

to give a semantics of untyped λ-calculus). It is then natural to try to extend this semantics to a non-standard model of a richer type theory, for instance the present theory of constructions, by interpreting a type as a domain. We get the interpretation of a family of types via the remark that there is a natural notion of "approximations" between domains: the embedding-projection pairs. We can thus define a "dependent domain" over a fixed domain D as a Scott-continuous (that is preserving directed colimits) functor from D, seen as a category, into the category Dom^{EP}, of domains with embedding-projection pairs as morphisms.

We get in this way an interpretation of type theory with products over dependent families. Notice that in this interpretation, any type is inhabited (at least by $\perp$). The main problem is the interpretation of Prop and $\forall$. An obvious candidate for Prop seems to be the "flat" domain $\mathsf{Bool}_\perp$. Over an infinite set α however the universal quantification $\forall : (\alpha \rightarrow \mathsf{Prop}) \rightarrow \mathsf{Prop}$ is not "continuous": the computation of $\forall(f)$ uses an infinity of values of f, and thus we cannot use $\mathsf{Bool}_\perp$ as an interpretation of Prop.

3.3.1 Domain of domains

A first solution to this problem has been given by D. Scott [63,82], using the notion of closures, following a suggestion of P. Hancock and P. Martin-Löf. Another solution using finitary projections has been found [1]. The rough idea is that everything is interpreted as a point of some "big" domain. The main drawback of these models is that they interpret a richer theory which is known to be "inconsistent" as a type theory, since there is a type of all types. Furthermore, these models are "not canonical" in the choice of the "big" domain to start with. However, direct generalisations are very convenient for showing the equational consistency of various extensions of the present calculus (strong sums, recursive types,$\cdots$ see [12]). This seems to indicate that there is nothing wrong with having a type of all types in a programming language which contains already a fixed-point operator [11]. However, what is missing here is an adequacy theorem that relates the operational and denotational semantics (like the one proved in [73] for the language PCF), which is a priori problematic, since the method used ordinary for proving these results is similar to the reducibility method used in the proof of normalisation, and we know that a calculus with a type of all types does not verify the normalisation property [30,64,18,45].

3.3.2 Category of domains

As discovered by J.Y. Girard [31], it is possible to interpret polymorphism in a model where the interpretation of a type is an arbitrary domain (and not as a point of a special "big" domain), so that we get really a "canonical" model of type theory in domain theory. Quite naturally, a dependent family of domains over all domains is interpreted as a continuous functor from Dom^{EP} to Dom^{EP}. Girard used in [31] a somewhat non-standard notion of domain and morphisms between these domains, but in [22], it is shown that his semantics of the product of a functor F is nothing but the domain of sections of the Grothendieck cofibration of F, that is the domain of all continuous (and stable) family (t_X) such that $t_X \in F(X)$ (see [22,65]). From this remark it is possible to develop a model similar to Girard's

one, but with ordinary Scott-domains and continuous maps [22]. What is crucial in these models is that any domain is a directed colimit of finite domains.

The general picture that emerges from the study of this family of models is then that a small type is interpreted as a domain, and a type in general will be interpreted as a category, which shares a lot of properties of domains, but at a categorical level [19]. Typically, the category of domains with embedding-projection pairs as morphisms is such a category (and is the interpretation Dom^{EP} of the large type Prop), but also the category of Scott-continuous functors (that is, functors that preserve filtered colimits). All these categories C will have the property that for any Scott-continuous functor F from C to Dom^{EP}, the collection of its continuous sections $t_X \in F(X)$ is a domain for the pointwise ordering. This has been done in details for $F\omega$ in [23], with complete algebraic lattices for small types and locally-finitely presented categories for general types. Some elements for a generalisation to Scott-domains or dI-domains are in [19] (see also [87]). One may object generally to these models that they are not faithful to the Curry-Howard notion of proposition as types, since any type is now inhabited (by $\perp$). However, it is possible to define a notion of total object (see the appendix of [31]) to remove $\perp$, and the notion of partial proofs suggested by these models may have some applications. We can also mention that there is no problem here to extend the adequacy theorem of [73] to the present framework (with the system of constructions instead of simple typed λ-calculus).

In [42], besides a careful analysis of what is a categorical model of dependent types, is presented a topos-theoretic formulation of the complete algebraic lattice model, that may be a key step for a conceptual understanding of the general picture. One starting point is that any domain can be seen as a locale, hence as a special kind of topos. As an application, it is shown that this model contains a type of all types (which corresponds to the locally finitely presented category of left-exact categories). One topos-theoretic version of Girard's model may be interesting (see [47] for a topos-theoretic interpretation of stability).

Finally, an important remark is that the constructions of all these domain models of polymorphic systems are all elementary (this elementary character is only lost when we try to define what are the total objects, see the appendix of [31]).

4 A few properties

4.1 Conservativity over $F\omega$

In [29], J.Y. Girard introduced a functional system which is a generalisation of Gödel system T, in order to extend the Dialectica interpretation to higher-order arithmetic (see [33]). It is also explained in [29] how this calculus can be seen as a system of natural deduction for intuitionistic higher-order propositional logic. It is thus not surprising that the language we have developed contains $F\omega$ as a subsystem.

In $F\omega$, $\Rightarrow$ is a primitive constant, and we need to introduce the clause that $M{\Rightarrow}N$ is a term if M and N are terms. We add the rules

$$\frac{M : \mathsf{Prop}\ [B] \qquad P : \mathsf{Prop}\ [B] \qquad N : P\ [B, x : M]}{(\Lambda x : M)N : M \Rightarrow P\ [B]}$$

$$\frac{M : \mathsf{Prop}\ [B] \qquad N : \mathsf{Prop}\ [B]}{M \Rightarrow N : \mathsf{Prop}\ [B]}$$

Furthermore we do not allow the rule of product formation

$$\frac{M\ \mathsf{type}\ [B] \qquad N\ \mathsf{type}\ [B, x : M]}{(x : M)N\ \mathsf{type}\ [B]}$$

if M is a small type. Typically, we cannot form a type like $(A : \mathsf{Prop})(P : A \to \mathsf{Prop})(x : A)P(x) \Rightarrow P(x)$.

A nice and simple result (noticed by V. Breazu-Tannen using a tool developped by Ch. Mohring) is that the calculus of constructions is a conservative extension of $F\omega$. This is shown by giving a "forgetting" map $\mathcal{E}$ from constructions to $F\omega$ (which can be seen as a modified realisability for construction into $F\omega$, see [68]). Intuitively, this map forgets all dependencies. For instance, $\mathcal{E}((\forall \alpha : \mathsf{Prop})(\forall P : \alpha \to \mathsf{Prop})(\forall x : \alpha)P(x))$ is $(\alpha, \beta : \mathsf{Prop})\alpha \to \beta$ (see [68,69]). This forgetting map is the identity on terms that are in $F\omega$, and preserves the typing: if $M : N$ in the system of constructions, then $\mathcal{E}(M) : \mathcal{E}(N)$ in $F\omega$. From this, we deduce

Proposition If M, N are terms in $F\omega$, then $M : N$ is provable in the calculus of constructions if, and only if, $M : N$ is provable in $F\omega$. Furthermore if M is a type of $F\omega$, then M is inhabited in $F\omega$ if, and only if it is inhabited in the calculus of constructions.

4.2 Connection with Higher-Order Logic

An elegant formal system is "minimal" higher-order logic (this is the system used in [70]). The types and the terms of this system are the same as the terms of Church's simply typed theory [13]. The types are built from the ground type Prop by the arrow operation, written here $\alpha \to \beta$. The terms are built by λ-abstraction $\lambda x : \alpha.M$, by application $M(N)$, and by typed variables. There is a constant $\Rightarrow : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}$ (we write $\varphi \Rightarrow \psi$ for $\Rightarrow(\varphi, \psi)$) and a "polymorphic" constant $\forall : (\alpha \to \mathsf{Prop}) \to \mathsf{Prop}$ (we write as usual $(\forall x : \alpha)\varphi$ for $\forall(\lambda x : \alpha.\varphi)$). A proposition is a term of type Prop.

We can define what is a "true" proposition relatively to a list of "hypotheses" Γ, relation that we write $\Gamma \vdash \varphi$ true, inductively as follows: first $\Gamma \vdash \varphi$ true if $\varphi \in \Gamma$, then

$$\frac{\Gamma, \varphi \vdash \psi\ \mathsf{true}}{\Gamma \vdash \varphi \Rightarrow \psi\ \mathsf{true}}$$

$$\frac{\Gamma \vdash \varphi \Rightarrow \psi\ \mathsf{true} \qquad \Gamma \vdash \varphi\ \mathsf{true}}{\Gamma \vdash \psi\ \mathsf{true}}$$

$$\frac{\Gamma \vdash \varphi\ \mathsf{true}}{\Gamma \vdash (\forall x : \alpha)\varphi\ \mathsf{true}} \qquad (*)$$

$$\frac{\Gamma \vdash (\forall x : \alpha)\varphi\ \mathsf{true}}{\Gamma \vdash [t/x]\varphi\ \mathsf{true}} \qquad (**)$$

In the rule (**), it is supposed that t is a term of type α, and in the rule (*), that the variable x does not occur free in Γ.

This logic is known as "minimal" higher-order logic. It may seem quite poor, but (as known already from Russell [76]), other intuitionistic connectives are then definable (see [30,21], the general principle is that the coding of a connective is the direct expression of its elimination rule). Let us give only here the representation of the existential quantification which will be used later: for a type A and a predicate φ over A we define $\exists x : A.\varphi(x)$ as the proposition $\forall p : \mathsf{Prop}.(\forall x : A.\varphi(x) \Rightarrow p) \Rightarrow p$. If a is an object of type A and b an object of type $\mathsf{T}(\varphi(a))$ then we can build as expected a proof $\mathsf{pair}(a, b)$ of $\exists x : A.\varphi(x)$. However, given an object x in $\mathsf{T}(\exists x : A.\varphi(x))$, there is no way to compute from x its components (this will be proved indirectly below by showing that the existence of the two projections contradicts the consistency result).

Furthermore, we can consider Leibniz' equality [77]: if x, y are two terms of type A, we define $\mathsf{Eq}(A, x, y)$ as $(P : A \rightarrow \mathsf{Prop})P(y) \Rightarrow P(x)$. This corresponds to the following rule: if $\mathsf{Eq}(A, x, y)$ and we want to prove the proposition $\varphi(x)$, then it's enough to prove $\varphi(y)$. We thus do not need to introduce a primitive notion of equality as in Martin-Löf type theory.

The system $F\omega$ can be seen as a language for expressing the proofs of this logic. We have seen how to embed $F\omega$ in the pure calculus of constructions. From this, we get a translation of intuitionistic higher-order logic in the system of constructions. From the conservativity result of the calculus of constructions over $F\omega$, we deduce also that this translation is conservative, that is a proposition φ is provable in intuitionistic propositional higher-order logic if, and only if, it is inhabited as a small type of the system of constructions. This last result can be seen as a (constructive) completeness theorem: a proposition is provable in higher-order logic if, and only if, its translation in the system of constructions (its "semantics") is true. By using the normalisation theorem, we can prove a similar result for intuitionistic first-order logic (see [58]).

Proposition The calculus of constructions is a conservative extension of minimal propositional higher-order logic.

Remark: We thus get another consistency proof for the system of constructions by (elementary) reduction to the consistency of higher-order logic.

Higher-order logic is ordinarily presented with extensionality. In this case, as explained in [2], it is better to take the (extensional) equality as primitive and it is then possible to define all other logical connectives from it (this is the solution chosen in [52] for instance, see also [35] that presents a proof-checker based on this logic). The connections between extensional higher-order logic and topos theory are explained in [52]. It is thus important to relate the intensional presentation of higher-order logic given above to an extensional one.

Such an interpretation was done by R. Gandy in [27]. But the spirit of this translation goes back to Principia [77]. The idea is to define for every type an extensional equality by induction. On the type Prop we take the logical equivalence as equality. On the type $\alpha \rightarrow \beta$ we take as extensional equality $\mathsf{Eq}_{\alpha\rightarrow\beta}(f, g) = \forall x, y : \alpha.\mathsf{Eq}_\alpha(x, y) \Rightarrow \mathsf{Eq}_\beta(f(x), g(x))$. We then say that an element x of type α is extensional if, and only if $\mathsf{Eq}_\alpha(x, x)$, and we have an interpretation of extensional

higher-order logic in intensional higher-order logic.

By composing these two interpretations: topos theory in extensional higher-order logic, and extensional higher-order logic in intensional higher-order logic, we thus get a translation of topos theory inside intensional higher-order logic. There is actually a direct way of building the free topos from intensional higher-order logic: an object of the free topos is defined as a pair of a type A, together with a partial equivalence relation on it $R : A \rightarrow A \rightarrow \mathsf{Prop}$. A morphism between (A, R) and (B, S) is a relation $f : A \rightarrow B \rightarrow \mathsf{Prop}$ which is "functional" with respect to R and S as in [52].

In [13], Church introduces a ground type ι of individuals. In the system of constructions described so far, we can introduce only "small" types $D : \mathsf{Prop}$. We can then translate faithfully any result of higher-order logic in the context with one proposition variable $D : \mathsf{Prop}$ (notice however that the conservativity is not clear, and that the previous realisability method will not help).

4.3 Categorical semantics

We put this section here, since categorical "semantics" are really more translation of a type-theoretic formalism in categorical terms than true semantics. We have seen the connection between higher-order logic and topos theory: extensional higher-order logic is the "internal language" of topos theory. This connection is sometimes used in topos theory in order to prove logically a categorical result. We know also that the calculus we consider is richer than type theory: each proposition gives rise to a "small" type, the type of its proofs.

This suggests in a natural way two questions: is there an extensional version of the calculus of constructions, and what is the categorical version of such a calculus? For the first question, it seems possible to introduce a primitive equality type, as in one version of Martin-Löf type theory (the one described in [62], and used in [15]). The system we get has not been studied yet. The categorical notion that generalises topos theory in that we have an explicit representation of proofs (the principle of "proof-irrelevance" holds in topos theory as well as in classical set theory) seems to be the following one: a locally cartesian closed category with a small complete category [41,25]. In [25] (where such categories are coined "dictos") an alternative description is given which emphasizes one interesting phenomenon [41]: if we have a small complete category C in a locally cartesian closed category E, then there is a reflection from E to C, so that, intuitively, in these models, each type has "a best approximation" in term of small types or propositions. This fact is also true in the algebraic lattices/ locally finitely presented category model [42]: given a locally finitely presented category its "reflected" complete algebraic lattice is the ideal completion of the canonical preorder associated to any small aubcategory of generators.

The categorical presentation of the intensional theory is described in [42,50,51, 25,86].

4.4 Representation of data types

There are representations of data structures in the system F (originated in [60], see for instance [6,53,20]) that we can use as types of individuals. We have given

already the coding of the type of booleans and natural numbers.

Let us give only some remark for the case of the natural number. We can define the iterator over small types $\lambda n:\mathsf{Nat}.\lambda A:\mathsf{Prop}.\lambda x:A.\lambda f:A\rightarrow A.n(A,x,f)$, which is reminiscent of the expression of a weak natural number object in category theory. One can then represent primitive recursive functions, via the usual coding of iterations and products [53,20]. For instance, for the predecessor function we define first the function $F(z)=(S(\pi_1(z)),\pi_1(z))$ of type $(\mathsf{Nat}\times\mathsf{Nat})\rightarrow(\mathsf{Nat}\times\mathsf{Nat})$ and then $\mathsf{pred}(x)=\lambda x:\mathsf{Nat}.\pi_2(x(\mathsf{Nat}\times\mathsf{Nat},F,(0,0)))$.

Let $\bar{n}$ be the canonical representation of the integer n in the system F, that is $\lambda A:\mathsf{Prop}.\lambda x:A.\lambda f:A\rightarrow A.f^n(x)$. Let us say that a function f from integers to integers is representable (in the system of constructions) if, and only if, there exists a term $M:\mathsf{Nat}\rightarrow\mathsf{Nat}$ such that, for any integer n, we have $M(\bar{n})=\overline{f(n)}$. Though it may seem difficult even to represent the predecessor function, the class of functions that we can define of type $\mathsf{Nat}\rightarrow\mathsf{Nat}$ is actually very large.

Proposition: A function is representable if, and only if, it is provably total in higher-order arithmetic.

Proof: The simplest proof is the one described in [68] (see [32] for the definition of provably total). By using the modified realisability of the system of constructions in $F\omega$ described above, we are reduced to a similar statement for $F\omega$ which was proved in [29] using the Dialectica interpretation and in [68] using a realisability method (from an idea of P. Martin-Löf [60]).

It should be said however that this representation of data types and recursors is not completely understood yet. As emphasised by J.L. Krivine, these representations theorems are *extensional*, and what matters as much is the *intensional* aspect. But the coding of primitive recursion via iteration and pairing (though all right from a denotational point of view) seems pretty bad intensionally. For instance, the computation of the predecessor of $\bar{n}$ will be in n β-reduction steps (for the normal order evaluation), and it is in one step with a system that has primitive recursion built-in (like Gödel's system T).

It should be noted also that, although we can state the induction principle

$$(P:\mathsf{Nat}\rightarrow\mathsf{Prop})P(0)\Rightarrow((x:\mathsf{Nat})P(x)\Rightarrow P(S(x)))\Rightarrow(x:\mathsf{Nat})P(x)$$

we have the following negative result:

Proposition: The induction principle over Nat is not provable.

Proof: By the normalisation theorem, we are reduced to show that in the context

$$P:\mathsf{Nat}\rightarrow\mathsf{Prop},h_1:P(0),h_2:(x:\mathsf{Nat})P(x)\Rightarrow P(S(x)),n:\mathsf{Nat}$$

there is no term in normal form of type $P(n)$. Indeed a term built from h_1 or h_2 will prove something of the form $P(0)$ or $P(S(x))$ and the *variable* n is not convertible to 0 or to a term of the form $S(x)$ (here we use the Church-Rosser property).

All these facts are strong motivations for the extension of the system with inductively defined types as primitive (see below, and [72,24]).

4.5 Inconsistent extensions

In our proof of consistency, the possibility of interpreting a type as a set (in Zermelo-Fraenkel sense) was crucial. Roughly speaking, this is the only known consistency criterion. That is, whenever a functional/logical system does not have such a property, it is likely that some form of Russell's paradox, or Burali-Forti paradox will apply and show the inconsistency.

4.5.1 The system U

The first example is Girard's system U (see [30]), that I will adapt to the present formalism. Following [18], we explain it first for logical systems, and then we derive what it means in term of constructions.

The type structure of minimal higher-order logic is the simply typed λ-calculus (and simply typed λ-calculus was historically created in [13] for that). As noticed in [59], simply typed structure presents unnatural restriction, since we cannot formulate a statement for an arbitrary type. A possible attempt to overcome this restriction is to start with the second-order λ-calculus instead of the simply typed λ-calculus.

We thus enlarge the types of higher-order logic with type variables and product over type variables $\Pi\alpha.T(\alpha)$ where $T(\alpha)$ is a type expression where α may occur as a variable. At the level of terms, we have to introduce an abstraction over type variables, so that $\lambda\alpha.M$ is of type $\Pi\alpha.T(\alpha)$ if M is of type $T(\alpha)$ (with the usual restriction that α cannot occur free in the type of the free variables of M), and instantiation of a term to a given type, so that $M(\tau)$ is of type $[\tau/\alpha]T$ if τ is a type and M a term of type $\Pi\alpha.T(\alpha)$. Notice that the constant $\forall$ is now a constant of type $\Pi\alpha.(\alpha\rightarrow\mathsf{Prop})\rightarrow\mathsf{Prop}$.

But we want also to state (and prove) generic statement. For that, we introduce a new constant $\bigwedge$ which is of type $(\Pi\alpha.\mathsf{Prop})\rightarrow\mathsf{Prop}$ (and we write $\bigwedge\alpha.\varphi$ for $\bigwedge(\lambda\alpha.\varphi)$). We express thus universal quantification over all types. Another view of this system would have been to consider it as an extension of the second order λ-calculus, with one special type Prop, and special constants for implications and quantifications.

We add then the new rules of inference:

$$\frac{\Gamma\vdash\varphi\ \mathsf{true}}{\Gamma\vdash\bigwedge\alpha.\varphi\ \mathsf{true}}\qquad(*)$$

$$\frac{\Gamma\vdash\bigwedge\alpha.\varphi\ \mathsf{true}}{\Gamma\vdash[\alpha/\tau]\varphi\ \mathsf{true}}\qquad(**)$$

In the rule (**), it is supposed that τ is a type, and in the rule (*), that the variable α doesn't occur free in Γ.

If we think of our types as sets, it seems clear that we get an inconsistent system: we cannot form an arbitrary product over all sets and get a set (in general it will be "too big"). We have to be careful however since it is not always obvious to translate a set-theoretical result in term of types (for instance, Russell's paradox shows at once that there cannot be a set of all sets, but it's not so clear that there cannot be a type of all types). This intuition is however correct: we can derive a paradox similar to Burali-Forti paradox [30,18,45].

It is also possible to derive a paradox similar to Cantor's paradox by translating in type theoretic terms the argument of Reynolds [75] that there is no set-theoretic model of polymorphism. The trick is that the construction of an initial T-algebra (see [75]) which uses in set-theory a "big" equalizer becomes in type theory the expression of an induction predicate over the type which is a weak initial T-algebra (see [24]).

A striking corollary of the inconsistency of the system U is that a type system with a type of all types contains non-normalisable terms [30,18]. Indeed, it is possible to interpret the system U in such a type system in such a way that propositions are interpreted by types, and that a proposition is provable in U if, and only if, its corresponding type is inhabited. In particular, the type $(X:\mathsf{Type})X$ is inhabited. We know that β-reduction preserves typing, and a closed term in $(X:\mathsf{Type})X$ cannot be in head-normal form. Hence, a closed term of type $(X:\mathsf{Type})X$ is not normalisable.

Intuitively, we cannot say that all types are (isomorphic to) small types in a consistent way.

4.5.2 Representation of existence

We have seen in the representation of existential quantification that from a given proof of $\exists x : A.\varphi(x)$, we have "no access to the two components of this proof". We know however (by using the normalisation theorem) that any closed proof of this proposition reduces to one term of the form $\mathsf{pair}(a,b)$ where a is of type A and b a proof of $\varphi(a)$. We can thus try to internalise this remark and to add to the system a "choice operator" which extracts the two components of the proof of an existential statement [43]. Notice that this conflicts with the intuition that we get from the proof-irrelevance (or the realisability, or the domain model). Indeed, the existential type $\exists x : A.\varphi(x)$ becomes them a name for the type $\Sigma x : A.\mathsf{T}(\varphi(x))$, but this type is not "small" in general. For instance, in the domain model, it will be in general a large category, and not at all a domain. There is nothing wrong a priori with having these choice operators if A is itself small.

From these remarks, it is not surprising that we can prove a contradiction from the introduction of the two projections for existential types. This can be done directly as in [18]. It has also been noticed [44,38] that we can interpret $Type : Type$ in this system by considering Prop as a type of all types for the new typing relation $M \in P$ if, and only if $\mathsf{pair}(M,id) : \mathsf{T}(\exists x : P.\mathsf{Unit})$ (where id is the canonical proof of $\mathsf{Unit} = (\forall\alpha : \mathsf{Prop})\alpha\rightarrow\alpha$). This shows that for an impredicative theory of constructions, we cannot have an existential quantification with choice operators.

This property can be interpreted by saying that the use of the existential type "hides" some information in such a way that it is impossible to get it back. It has been suggested by J. Mitchell and G. Plotkin [66] to use existential types for the representation of abstract data types.

4.5.3 Semantics of evidence for classical logic

It seems clear that we can extend the interpretation of propositions as sets for intuitionistic propositional logic to classical logic simply by adding a special element

$?_A : A + \neg A$. The goal of this section is to show that such an addition for an impredicative logic will trivialise any explicit considerations of proofs, so that the proof-irrelevance principle follows from the assumption that the logic is classical and impredicative.

The intuition behind this result is the following. It has been noticed by Spector [85] that if we add to predicative analysis the axiom of choice and the law of excluded middle, then we get a logic as strong as impredicative analysis. We can thus expect to get two levels of impredicativity if we add the magic element $?_A : A + \neg A$, and we can then apply the inconsistency of the system U to ge the inconsistency of this extension (with the "propositions-as-types" meaning, that is all types become inhabited).

Lemma: The following context

$$B : \mathsf{Prop},\ E : B \rightarrow \mathsf{Prop},\ \epsilon : \mathsf{Prop} \rightarrow B,\ H : (A : \mathsf{Prop}) A \Leftrightarrow E(\epsilon(A)),$$

is inconsistent.

Intuitively, this says that there cannot be any "reflection" from the "category" of small types into one small type.

Proof: The proof of this lemma is by a direct interpretation of the system U in that context. The small type B is used for the representation of the type of truth-value of the system U and in general the types of the system U are interpreted by small types. We define $\Rightarrow_1 : B \rightarrow B \rightarrow B$, $\forall_1 : (A : \mathsf{Prop})(A \rightarrow B) \rightarrow B$, and $\bigwedge : (\mathsf{Prop} \rightarrow B) \rightarrow B$.

$$\begin{array}{rcl} \Rightarrow_1 & = & (\lambda p, q : B)\epsilon(E(p) \rightarrow E(q)) \\ \forall_1 & = & (\lambda A : \mathsf{Prop})(\lambda f : A \rightarrow B)\epsilon((x : A) f(x)) \\ \bigwedge & = & (\lambda F : \mathsf{Prop} \rightarrow B)\epsilon((A : \mathsf{Prop}) F(A)). \end{array}$$

This gives an interpretion of the system U. We thus get that for any $p : B$, $E(p)$ is inhabited. In particular, if $A : \mathsf{Prop}$, $E(\epsilon(A))$ is inhabited, and so A is inhabited. This means that the given context is inconsistent.

In order even to state the next result, we need to extend the calculus of constructions with disjoint sums à la Martin-Löf. This means that we add a new type forming operation $A + B$ type, if A type and B type, with the rules that $i(M) : A + B$ if $M : A$, $j(N) : A + B$ if $N : B$. Finally, if A type, B type, $C(z)$ type $[z : A + B]$, then

$$\frac{M(x) : C(i(x))[x : A] \quad N(y) : C(j(y))[y : B] \quad P : A + B}{\mathsf{if}(P, M, N) : C(P)}$$

with the conversions rules that $\mathsf{if}(i(x), M, N) = M(x) : C(i(x))\ [x : A]$ and $\mathsf{if}(j(y), M, N) = N(y) : C(j(y))\ [y : B]$.

Proposition: In the calculus of construction with disjoint sums, the existence of a classical operator $? : (p : \mathsf{Prop})\mathsf{T}(p) + \mathsf{T}(\neg(p))$ implies that any small type has at most one element for Leibniz equality.

Proof: We consider the small type of booleans $\mathsf{Bool} = (A : \mathsf{Prop}) A \rightarrow A \rightarrow A$, with $\mathsf{true} = (\lambda A : \mathsf{Prop})(\lambda x, y : A)x : \mathsf{Bool}$ and $\mathsf{false} = (\lambda A : \mathsf{Prop})(\lambda x, y : A)y : \mathsf{Bool}$. It is enough to

show that $\mathsf{Eq}(\mathsf{Bool},\mathsf{true},\mathsf{false})$, where the equality used is Leibniz equality, since for any small type A and any $a, b : A$, we can define $f : \mathsf{Bool}{\rightarrow}A$ such that $f(\mathsf{true})$ is convertible to a, and $f(\mathsf{false})$ is convertible to b.

If we have $? : (p{:}\mathsf{Prop}).\mathsf{T}(p) + \mathsf{T}(\neg(p))$, we are in classical logic and we can reason by cases. Hence, we can suppose that $\neg(\mathsf{Eq}(\mathsf{Bool},\mathsf{true},\mathsf{false}))$. We then show $\perp$ by reduction to the previous lemma.

We define the map $E : \mathsf{Bool}{\rightarrow}\mathsf{Prop}$ by $E(p) = \mathsf{Eq}(\mathsf{Bool},p,\mathsf{true})$, for $p : \mathsf{Bool}$. For the map $\epsilon : (\mathsf{Prop})\mathsf{Bool}$, we consider, for $p : \mathsf{Prop}$, $?(p) : \mathsf{T}(p) + \mathsf{T}(\neg(p))$, and we take

$$\epsilon(p) = \mathsf{if}(?(p), (\lambda x : \mathsf{T}(p))\mathsf{true}, (\lambda x : \mathsf{T}(\neg(p)))\mathsf{false}).$$

We can then show, by using the hypothesis $\neg(\mathsf{Eq}(\mathsf{Bool},\mathsf{true},\mathsf{false}))$, that, for an arbitrary $A : \mathsf{Prop}$, A is equivalent to $E(\epsilon(A))$, hence the result by the lemma.

This implies that the only "model" of such a theory is the "proof-irrelevance" model, where we interpret Prop as a set with two elements, and each type is either empty, or with only one element. In particular, a non-trivial "small complete category" [41] cannot satisfy the choice principle. This says also intuitively that there cannot be any realisability like interpretation of impredicative classical logic.

5 Some possible extensions

Does the constructive proof of Gödel's Incompleteness Theorem suggest any reflection principles that could be added to the theory preserving its constructive character? Would this be a way in which an "abstract" theory of proofs might become interesting again? There seems to be quite a number of things to think about this area, and the theory of constructions, in this form or another, gives us a way to making the questions and answers precise (D. Scott [81]).

5.1 Addition of universes

As pointed out in [59], the simple theory of types, although proof theoretically quite strong, has some unnatural limitations: it does not talk about arbitrary set, but instead, talks about individuals or sets of individuals, or sets of sets of individuals, $\cdots$. In [58], the assumption that there is a type of all types which was formulated avoids this problem, but, as we have seen, this is inconsistent with the proof-as-objects idea. There is however another possible solution to this difficulty, which is to use a reflection principle (as in [61]).

For the expression of this reflection principle, we introduce a special type U, called a universe, together with the assumptions that U is closed under products: if A is a type in U and $B : A{\rightarrow}U$, then $(x{:}A)B(x)$ is a type in U, and $\mathsf{Prop} : U$. With this addition, we can introduce type variables $X : U$, then state, and prove, theorems in a generic way in X, like the fact that the inclusion relation between predicate over X is transitive. The consistency of this extension is clear since we can extend the proof-irrelevance semantics to this new calculus, by taking for $[\![U]\!]$ the set V_ω of all hereditarily finite sets. Notice however that this consistency proof is no longer elementary, and indeed, by considering the type of "predicative" Church numerals $(X{:}U)(X{\rightarrow}X){\rightarrow}X{\rightarrow}X$, it is possible to interpret higher-order arithmetic

in the empty context, so that there cannot be any elementary consistency proof any more. In [55], the realisability model (and in [56,5], the normalisation proof) is extended to the system with universes.

An interesting problem is to characterize the proof theoretic strength of higher-order logic together with the reflection principle. It seems likely that we can interpret Zermelo set theory by a direct extension of the usual representation of Zermelo set theory with the bounded comprehension axiom in higher-order logic (see [34] for details), to a representation of the full Zermelo set theory in higher-order logic with the reflection principle.

In practice, it seems that we do not need to use more than one or two universe levels. One natural question however is to relate this kind of reflection principle to the one studied by Turing-Feferman [26] (see [46] for examples of the use of the reflection principle). One is tempted to iterate this "universe principle", at first over the integers [62,18,55] (in particular notice the slight improvement in the formulation of [55], which introduces explicitely a subtyping relation that expresses really the cumulativity of the universes). G. Huet has proposed a notion of "universe polymorphism" [40,37]. We then get a system which combines impredicative formations (at the level of propositions), and predicative logic (at the level of types), and we know furthermore from Girard's paradox that such an extension is, in some way "the best possible" [18]. It may be interesting to illustrate ideas from [54] in the system with universe polymorphism, and to try to develop a mechanical study of predicative reasoning (as started in the introduction of the second edition of Principia [77]).

5.2 Strong sums

We have seen that we cannot add strong elimination rules for the existential quantification, so that existential quantification becomes a "strong sum". However, the proof-irrelevance semantics suggests that it is possible to add a strong sum for the types. We add the terms $\Sigma(M,N)$, the pairing operation $\mathsf{pair}_A(M,N)$ and the rules

$$\frac{A \ \mathsf{type} \qquad P \ \mathsf{type} \ [x : A]}{\Sigma(A,P) \ \mathsf{type}}$$

In the following three rules, we assume implicit the hypotheses:
A type and P type $[x : A]$.

$$\frac{M : A \qquad N : P \ [x : A]}{\mathsf{pair}_{\Sigma(A,P)}(M,N) : \Sigma(A,P)}$$

$$\frac{Q \ \mathsf{type} \ [z : \Sigma(A,P)] \qquad M : (x{:}A)(y{:}P)Q[\mathsf{pair}_{\Sigma(A,P)}(x,y)]}{\mathsf{Elim}(M) : (z : \Sigma(A,P))Q}$$

$$\frac{Q \ \mathsf{type} \ [z : \Sigma(A,P)] \qquad M : (x{:}A)(y{:}P)Q[\mathsf{pair}_{\Sigma(A,P)}(x,y)] \qquad a : A \qquad b : P[a]}{\mathsf{Elim}(M)(\mathsf{pair}_{\Sigma(A,P)}(a,b)) = M(a,b) : Q[\mathsf{pair}_{\Sigma(A,P)}(a,b)]}$$

As in [62], from these operators, we can define the two projections. Since we have an interpretation of these rules in the proof-irrelevance model, we know that

this extension is consistent. We write $\Sigma x : A.B(x)$ for $\Sigma(A, B)$ with A type, and $B : A \rightarrow \text{Prop}$.

One interesting application of strong sums is the representation of mathematical theories, and this is particularly powerful when combined with universes [71,57]. A theory will be defined by its carrier part which is a type, and its axiomatisation, which is a predicate over this type. For instance, the theory of reflexive relation is defined by the carrier part $T = \Sigma(U, \lambda A : U.A \rightarrow A \rightarrow \text{Prop})$, and the axiomatisation is the predicate $\psi = \text{Elim}(\lambda A : U.\lambda R : A \rightarrow A \rightarrow \text{Prop}.(x : A)R(x, x))$ which is of type $T \rightarrow \text{Prop}$. A reflexive relation can then be seen as an object of type $\Sigma(T, \psi)$. A way to think of this last type is as the "subset" of objects in T that satisfies ψ, or as the type of models of the theory defined by T and ψ.

We can now consider the notion of morphisms, or interpretations, between theories. For instance, we can consider the notion of converse of a relation. We define first a function $\text{conv} : T \rightarrow T$ by $\text{conv} = \text{Elim}(\lambda A : U.\lambda R : A \rightarrow A \rightarrow \text{Prop}.\text{pair}_T(A, \lambda x, y : A.R(y, x)))$. We can then remark that we have a proof of $(x : T)\psi(x) \Rightarrow \psi(\text{conv}(x))$. This interpretation has thus two parts: one part is defined purely at the level of support, and the other part is purely logical.

This seems to be a general phenomenon: an interpretation between two theories $\Sigma(T_1, \psi_1)$ and $\Sigma(T_2, \psi_2)$ will have one carrier part $f : T_1 \rightarrow T_2$, and one logical part in $(x : T_1)\psi_1(x) \Rightarrow \psi_2(f(x))$.

We can think of $\Sigma(T, \psi)$ as the type of models of the theory defined by the pair (T, ψ). Interpretations between two theories (T_1, ψ_1) and (T_2, ψ_2) are the maps $f : T_1 \rightarrow T_2$ such that $(x : T_1)(\psi_1(x))\psi_2(f(x))$ holds, so that they are themselves elements of the type of models of the theory defined by $(T_1 \rightarrow T_2, \lambda f : T_1 \rightarrow T_2.(x : T_1)\psi_1(x) \Rightarrow \psi_2(f(x)))$.

If we want to internalize this discussion, we have to use two levels of universes. We define the type $\text{Theory} : U_2$ as $\Sigma(U_1, \lambda A : U_1.A \rightarrow \text{Prop})$. We have an evaluation mapping $\text{Mod} : \text{Theory} \rightarrow U_1$ defined by $\text{Mod}(Th) = \text{Elim}(Th)(\lambda A : U_1.\lambda\psi : A \rightarrow \text{Prop}.\Sigma(A, \psi))$. We can now define what is a morphism between two theories $\text{MOR} : \text{Theory} \rightarrow \text{Theory} \rightarrow \text{Theory}$ by defining

$$\text{mor}(T_1, \psi_1, T_2, \psi_2) = \text{pair}_{\text{Theory}}(T_1 \rightarrow T_2, \lambda f : T_1 \rightarrow T_2.(x : T_1)\psi_1(x) \Rightarrow \psi_2(f(x)))$$

and $\text{MOR}(Th_1, Th_2) = \text{Elim}(Th_1)(\lambda T_1 : U_1.\lambda\psi_1 : T_1 \rightarrow \text{Prop}.$
$\text{Elim}(Th_2)(\lambda T_2 : U_1.\lambda\psi_2 : T_2 \rightarrow \text{Prop}.\text{mor}(T_1, \psi_1, T_2, \psi_2)))$.

If F is an object in $\text{Mod}(\text{MOR}(Th_1, Th_2))$, and S_1 an object in $\text{Mod}(Th_1)$, then we can apply F to S_1, yielding an object in $\text{Mod}(Th_2)$. That is, we have an object application in

$$(Th_1, Th_2 : \text{Theory})\text{Mod}(\text{MOR}(Th_1, Th_2)) \rightarrow \text{Mod}(Th_1) \rightarrow \text{Mod}(Th_2).$$

The remark that it was possible to internalize the representation of theories in the calculus itself was explained to me by R. Pollack (from discussions with Z. Luo).

From what we just said, there is no object in

$$(Th_1, Th_2 : \text{Theory})(\text{Mod}(Th_1) \rightarrow \text{Mod}(Th_2)) \rightarrow \text{Mod}(\text{MOR}(Th_1, Th_2)),$$

so that, a priori, there is a problem to interpret the operation of simple type λ-calculus with theories. However, it is quite possible to define a product operation $\times$: Theory $\rightarrow$ Theory $\rightarrow$ Theory, and to interpret the equational presentation of cartesian closed category [52] in this framework.

There are three (at least formal) analogies that seem quite interesting: the first is the distinction between runtime and compile time that is discussed in [12] as an argument for avoiding dependent types (for compilation), the other is E. Moggi's notion of modules for SML [67], the last one is the interpretation of the implication in realisability (see [69]).

5.3 Inductive Types

As we noticed before, there are some problems in the impredicative representation of inductive types. It is thus natural to try to extend the core calculus with a primitive notion of inductively defined types. We shall not give here the full theory (see [72,24]), but only illustrate some points about such an extension.

For the natural number object, we want to introduce one type Nat (a priori, it is not clear whether we want this type to be small or not) with one constant $0 : \mathsf{Nat}$ and one successor operation $S : \mathsf{Nat} \rightarrow \mathsf{Nat}$. We then introduce one elimination operator rec with the rule

$$\frac{P(x)\ \mathsf{type}\ [x : \mathsf{Nat}] \quad a : P(0) \quad f : (x{:}\mathsf{Nat})(P(x))P(S(x))}{\mathsf{rec}(a, f) : (x{:}\mathsf{Nat})P(x)}$$

We add the new conversion rules that $\mathsf{rec}(a, f)(0) = a$ and $\mathsf{rec}(a, f)(S(x)) = f(a, \mathsf{rec}(a, f)(x))$, for $x : \mathsf{Nat}$.

With the operator rec, we have simultaneously the possibility of building terms and of proving properties by induction over Nat. Furthermore, since Prop is a type, we can define by induction propositions, predicates,... For instance, we can define the property Z of being equal to 0 by $\mathsf{rec}(\mathsf{true}, \lambda x : \mathsf{Nat}.\lambda y : \mathsf{Prop}.\mathsf{false})$ with $\mathsf{true} = (A{:}\mathsf{Prop})(A)A$ (actually any true proposition will do) and $\mathsf{false} = (A{:}\mathsf{Prop})A$. We can also define the predecessor function as $\mathsf{rec}(0, \lambda x, y : \mathsf{Nat}.x)$. From this, we can deduce the other Peano axioms: the axiom $(x{:}\mathsf{Nat})\neg\mathsf{Eq}(\mathsf{Nat}, S(x), 0)$ follows from the existence of the property Z, and the axiom $(x, y : \mathsf{Nat})\mathsf{Eq}(\mathsf{Nat}, S(x), S(y)) \Rightarrow \mathsf{Eq}(\mathsf{Nat}, x, y)$ follows from the existence of the predecessor function.

If we do not ask Nat to be a small type, then the proof irrelevance semantics gives a consistency proof. As expected, this semantics is not elementary, since we need an infinite set in order to interpret Nat. If we want Nat as a *small* type, then the semantics has to be more subtle than the proof irrelevance semantics, since in this semantics, small types are interpreted as sets with at most one element. It turns out that the realisability interpretation works here, by interpreting Nat as the set of untyped λ-term of the form $\lambda f.\lambda x.f^n(x)$ (or by adding special constants $0, S, \mathsf{rec}$ to the untyped λ-calculus with the new conversion rules $\mathsf{rec}(a, f, 0) = a$ and $\mathsf{rec}(a, f, S(x)) = f(a, \mathsf{rec}(a, f, x))$).

With one universe U, we can internalize the recursion operator as a constant of type $(P : \mathsf{Nat} \rightarrow U)P(0) \rightarrow ((x : Nat)P(x) \rightarrow P(S(x))) \rightarrow (x : \mathsf{Nat})P(x)$. With this extension, we can also (as in [61]) define by induction families like $P(n) = A^n$, with A type, and we can define, for instance the notion of sums of a n-tuple of integers as a term of type $(n : \mathsf{Nat})\mathsf{Nat}^n \rightarrow \mathsf{Nat}$, which could be defined only at

the meta-level before. The combination of universes and inductive types is thus extremely powerful for the internalisation of meta-arguments (see [46]).

Conclusion

We have presented a possible expression of Heyting's semantics for intuitionistic higher-order logic, and extended for this purpose Howard's theory of constructions. The notion of construction developped here is however still too crude for being suitable for the interpretation of intuitionistic mathematics (in particular we have not considered at all the difficult problem of the representation of choice sequences). The goal of our approach was only to show that a type-theoretic presentation of intuitionistic mathematics is possible (in opposition to a categorical or set-theoretical setting). We hope that it will be interesting to mechanize and illustrate proof-theoretic results (like the realisability method or the Dialectica interpretation) in such a framework and that it will help towards the understanding of the mystery of impredicativity.

Acknowledgments

Serge Yoccoz suggested to me the problem of the semantics of evidence for classical impredicative analysis.

References

[1] R. Amadio, K. Bruce, and G. Longo. "The finitary projection model for second order lambda calculus and solutions to higher order domain equations." LICS Boston, 1986.

[2] P. Andrews. "General Models and Extensionality." JSL, vol 37, No 2, 1972.

[3] P. Andrews. "An introduction to Mathematical Logic and Type Theory: To Truth through Proofs." Academic Press, 1986.

[4] H. Barendregt. "The forest of lambda calculi with types." Talk given at the Workshop of Lambda Calculus and Category Theory, CMU, 1988.

[5] S. Berardi. "A subsystem λ_Z of λ_U without Girard's paradox." Draft, 1988.

[6] C. Böhm and A. Berarducci. "Automatic synthesis of typed λ-programs on term algebras." Theoretical Computer Science 39, 1985.

[7] 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.

[8] 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.

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

[10] 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).

[11] R. Burstall and B.Lampson. "Pebble, a Kernel Language for Abstract Data Types and Modules." Information and Computation, Volume 76, 1989.

[12] L. Cardelli. "A Polymorphic λ-calculus with Type:Type." Private communication (1986).

[13] A. Church. "A formulation of the simple theory of types." Journal of Symbolic Logic 5, 1940.

[14] R. Constable "On the semantics of evidence." Manuscript, Marktoberdorf summer school, 1988.

[15] R. Constable & alii. "Implementing mathematics with the Nuprl Proof Deevlopment System." Prentice-Hall edition, 1986.

[16] 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).

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

[18] T. Coquand. "An Analysis of Girard's Paradox." LICS, Boston, 1986.

[19] T. Coquand. "Categories of Embeddings." LICS, Edimburgh, 1988.

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

[21] T. Coquand, G. Huet. "The Calculus of Constructions." Information and Computation, Volume 76, 1988.

[22] T. Coquand, C. Gunter, G. Winskel. "Domain Theoretic Models of Polymorphism." Cambridge Technical Report No 116 (1987), to appear in Information and Computation.

[23] T. Coquand, T. Ehrhard. "An equational presentation of Higher-Order Logic." Proceedings of Category theory and computer science, Edimburgh, Springer Lecture Notes in Computer Science, vol. 283, 1987.

[24] T. Coquand, Ch. Paulin-Mohring. "About inductively defined types."' To appear in the proceedings of the Tallin conference, 1988.

[25] Th. Ehrhard. "Une sémantique catégorique des types dépendants." Thèse, Université Paris VII, 1988.

[26] S. Feferman. "Turing in the land of O(z)" Stanford University, 1987.

[27] R.O. Gandy. "On the axiom of extensionality-Part I." J.S.L. 21, 1956.

[28] G. Gentzen. Collected Works. Edited by Szabo, North Holland, 1969.

[29] J.Y. Girard. "Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et dans la théorie des types." Proc. 2nd. Scand. Log. Symp., North Holland, 1971.

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

[31] J.Y. Girard. "System F: fifteen years later" Theoretical Computer Science, vol. 45, 1986.

[32] J.Y. Girard. "Proof Theory and Logical Complexity, Part I" Bibliopolis, 1988.

[33] K. Gödel. "On a Hitherto Unexploited Extension of the Finitary Standpoint." Translation by W. Hodges, Journal of Philossphical Logic 9, 1980.

[34] R. Goldblatt. "Topoi: the Categorical Analysis of Logic." North-Holland, 1979.

[35] M. Gordon. "HOL A Machine Oriented Formulation of Higher Order Logic." Cambridge Technical Report, no. 68.

[36] R. Harper, F. Honsell, G. Plotkin. "A Framework for defining Logics." LICS Ithaca, 1987.

[37] B. Harper and R. Pollack. "Type-Checking, Universe Polymorphism, and Typical Ambiguity in the Calculus of Constructions." To appear in the proceedings of CCIPL, 1989.

[38] B. Harper and J. Mitchell. "The Essence of ML." ACM POPL 1988.

[39] L. Hehmink and R. Ahn. "Goal Directed Proof Construction in Type Theory." Draft, Philips Research Laboratory, 1988.

[40] G. Huet. "A calculus with Type:Type." Unpublished manuscript, 1987.

[41] M. Hyland. "A Small Complete Category." In proceedings of Church's Thesis Conference, to appear in Ann. Pure. Appl. Logic, 1986.

[42] J.M.E. Hyland, and A.M. Pitts. "Categorical model of the theory of constructions." in Proceedings of the Boulder Conference, Contemporary Mathematics, AMS, Providence RI, 1988

[43] W. A. Howard. "The formulæ-as-types notion of construction." In Hindley, Seldin To H.B. Curry, Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, 1980.

[44] J. Hook and D. Howe. "Impredicative strong existential equivalent to type:type." Technical Report, Cornell University, 1986.

[45] D.J. Howe. "The Computational Behaviour of Girard's Paradox." LICS 87 (1987).

[46] D. Howe. "Automatic Reasoning in an Implementation of Constructive Type Theory." Ph. D. thesis, 1988.

[47] P. Johnstone. "A Topos-Theoretic Look at Dilators." Unpublished manuscript, Cambridge 1988.

[48] B. Jutting. "Normalisation in the system of constructions." Unpublished Manuscript, 1986.

[49] François Lamarche. "A model of the theory of constructions." Comptes Rendus Soc. Roy. Canada, Vol X, No 2 (1988) 89-94.

[50] François Lamarche. "A simple model of the theory of constructions." in Proceedings of the Boulder Conference, Contemporary Mathematics, AMS, Providence RI, 1988

[51] François Lamarche. "Modelling Polymorphism with Categories." Ph. D. thesis, McGill University 1988

[52] J. Lambek and P. J. Scott. "Introduction to higher order categorical logic" Cambridge studies in advanced mathematics, Cambridge University Press, 1986.

[53] D. Leivant. "Reasoning about functional programs and complexity classes associated with type disciplines." 24th FOCS, 1983.

[54] D. Leivant. "Stratified Polymorphism." Unpublished Manuscript, CMU, 1988.

[55] Z. Luo. "ECC, an Extended Calculus of Constructions." To appear in LICS 89.

[56] Z. Luo. "$CC^{\infty}_{\sqsubset}$ and its Meta Theory." LFCS report ECS-LFCS-88-58, 1988.

[57] D.B. MacQueen. "Using Dependent Types to Express Modular Structure." ACM POPL, 1986.

[58] P. Martin-Löf. "A Theory of Types." Unpublished manuscript.

[59] P. Martin-Löf. "On the strength of intuitionistic reasoning." International Congress for Logic, Methodology and Philosophy of Science, 1971.

[60] P. Martin-Löf. "A construction of the provable wellorderings of the theory of species." Unpublished manuscript.

[61] P. Martin-Löf. "An Intuitionistic Theory of Types: predicative part." Logic Colloquium '73, North-Holland, 1975, 73-118.

[62] P. Martin-Löf. "Intuitionistic Type Theory." Bibliopolis, 1980.

[63] N. McCracken. "An investigation of a programming language with a polymorphic type structure." Ph.D. Dissertation, Syracuse University (1979).

[64] A. Meyer and M. Reinholt. "Type is not a type" POPL 86

[65] J. Messeguer. "Relating Models of Polymorphism." Proceeding of POPL 89.

[66] J. Mitchell and G. Plotkin. "Abstract types have existential types." ACM POPL 1985.

[67] E. Moggi. Personal Communication.

[68] Ch. Paulin-Mohring. "Extraction de programmes dans le Calcul des Constructions." Thèse, Université Paris 7, 1989.

[69] Ch. Paulin-Mohring. "Extracting Fω programs from proofs in the calculus of construction." Proceedings of POPL 89.

[70] L. Paulson "The Representation of Logic in Higher-Order Logic." Cambridge Technical Report, 113, 1987.

[71] B. Nordström and K. Peterson. "The Semantics of Module Specifications in Martin-Löf's Type Theory." Chalmers Technical Report 36, 1985.

[72] F. Pfenning. "Inductively defined types for the constructions." To appear in MFPLS, 1989.

[73] G. Plotkin. "LCF as a programming language." Theoretical Computer Science 5, 1976.

[74] J. C. Reynolds. "Towards a Theory of Type Structure." Programming Symposium, Paris. Springer Verlag LNCS **19** (1974) 408–425.

[75] J.C. Reynolds. "Polymorphism is not Set-Theoretic." Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[76] B. Russell. "The Principles of Mathematics." 1903.

[77] B. Russell and A.N. Whitehead. "Principia Mathematica." Volume 1,2,3 Cambridge University Press, 1912.

[78] A. Salvesen. "Polymorphism and Monomorphim in Martin-Löf's Type Theory." To appear in Logic Colloquium' 88, Padova, 1988.

[79] A. Scedrov. "A guide to polymorphic types." To appear in Logic and Computer Science, edited by P. Odifreddi, Academic Press, 1989.

[80] D. Scott. "A Type-Theoretical Alternative to CUCH, ISWIM, OWHY." Unpublished manuscript, Oxford, 1969.

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

[82] D. Scott. "Data Types as Lattices." SIAM Journal of Computing **5** (1976) 522–587.

[83] J.P. Seldin. "Progress report on generalized functionality." Ann. Math. Logic. 17 (1979).

[84] J. Smith. "Non derivability of Peano Axioms in Type Theory without Universes." To appear in Journal of Symbolic Logic, 1986.

[85] C. Spector "Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles formulated in current intuitionistic mathematics." Recursive function theory, Proc. Symp. Pure Math., vol. V, A.M.S. Providence, 1962.

[86] Th. Streicher. "Correctness and Completeness of a Categorical Semantics of the Calculus of Constructions." Ph. D. Thesis, Passau, 1988.

[87] P. Taylor. "The Trace Factorisation and Cartesian Closure for Stable Categories." Unpublished manuscript, 1989.

Extracting F_ω's Programs from Proofs in the Calculus of Constructions

Christine Paulin-Mohring

INRIA and LIENS
URA CNRS 1327
Ecole Normale Supérieure
45 Rue d'Ulm
75230 PARIS Cedex 05
FRANCE

We define in this paper a notion of realizability for the Calculus of Constructions. The extracted programs are terms of the Calculus that do not contain dependent types. We introduce a distinction between informative and non-informative propositions. This distinction allows the removal of the "logical" part in the development of a program. We show also how to use our notion of realizability in order to interpret various axioms like the axiom of choice or the induction on integers. A practical example of development of program is given in the appendix.

Introduction

Several systems have been developed in order to formalize and check mathematical reasoning. Among these is the Calculus of Constructions. It is an impredicative higher-order λ-calculus with dependent types [5, 4, 8, 7]. Via the "Curry-Howard" correspondence [11], if t is a well-formed term of type M, then we can read t as an higher-order proof presented in a natural deduction way of some proposition represented by M.

Given a formalized intuitionistic proof of a proposition, it is natural to look at the computational contents of this proof. We are also interested in deriving correct algorithms as proofs of existential formulae universally quantified $\forall x.\exists y.P(x,y)$, where P is the specification of the program. Examples of derivations of algorithms in this formalism may be found in [13].

A first attempt to obtain the computational meaning of a proof was the "stripping" process [6] that removes all types from the proof. But this term still contains informations that are useless for computation. For example, some part of the proof may disappear during reduction or is known to always reduce to some given term. This problem appears also in systems like Nuprl or Martin-Löf's Type Theory where a subset type is introduced in order to hide part of the computational information. In the system PX of Hayashi, a syntactic class of non-informative propositions is recognized by the process of extraction.

[1]This paper appeared in the proceedings of POPL '89

We propose a way to hide computational information in the Calculus of Constructions. We introduce two constants *Spec* and *Prop* in order to do a syntactic distinction between the propositions that have a "computational informative" contents and the propositions that only have a "logical" contents. We guide the process of extraction by marking in the propositions some parts that are useless for computation. The system checks the correctness of the marking. We extract programs from proofs by removing the "logical" information and also the type's dependencies.

For example, let t be a proof of the existential type $\exists x : A.P(x)$. If P is "informative" the extraction will forget about the dependency of P with respect to x. We shall extract from t a pair of type the conjonction of the types extracted from A and P. If P is marked to be "non-informative" then this proof will give us an element of type the type extracted from A.

Now if a term t is extracted from a proof of A, then it certainly satisfies some extra properties (depending on A). For example it will be convenient to be able to prove something like : If P is a "non-informative" proposition then the term extracted from a proof of $\forall x.\exists y.P(x, y)$ is a fonction f such that $P(x, f(x))$ is "true". The definition of a notion of realizability gives a precise and systematic way to internalize the computational meaning of the intuitionistic propositions. With each proposition A of the Calculus, we associate a predicate of realizability $\lambda x.\mathcal{R}(A, x)$. We prove that if the judgement $t \in A$ is derivable in the system then there is a proof of the proposition $\mathcal{R}(A, t')$ with t' the term extracted from t.

It is possible to interpret the proposition A as the set of terms u such that the proposition $\mathcal{R}(A, u)$ may be proved. Let assume that there exists a term u such that $\mathcal{R}(A, u)$ is provable. Then it is possible to prove that A is consistent with the theory. If we add A as an axiom axA then axA may occur in the extracted programs and we have to say how to reduce it. In this case we shall know that it is correct to replace axA by u in the extracted programs. We will show several examples of propositions that are not provable but that are realizable in our system.

In the first section, we present the Calculus of Constructions with Realizations that is a variant of the Calculus of Constructions with explicit distinction between proofs and programs, and between informative and non-informative propositions. In the second section we define our notion of realizability for this Calculus and state its properties. The third section gives examples of application of this notion. In the appendix is a complete example of development of a program that find the minimum of a non empty list.

1 The Calculus of Constructions with Realizations

In the original Calculus of Constructions proofs are identified with programs and each proof is intuitionistic and represents a program. In the Calculus of Constructions with Realizations there will be a syntactic distinction between pure programs and proofs. And among the proofs we separate "logical" proofs and proofs from which we will extract programs.

1.1 The system F_ω as a programming language

The Calculus of Constructions contains Girard's F_ω system. The system F_ω is the part of the Calculus of Constructions where types are not dependent of proofs. It is worth noting that each pure λ-term typed in the Calculus of Constructions is also typed in the system F_ω. The dependent types add an extra logical information about the term. We may see a proof of the Calculus of Constructions as a development of a program in the system F_ω.

We briefly recall some features of the system F_ω. In the system F_ω there are three levels of terms : the orders, the data types schemes and the programs. We call Λ_D the set of terms of F_ω. This set contains a set V_D of variables and is defined as follows.

Orders *Data* is an order and if A and B are orders so is $A \Rightarrow B$.

Data types schemes Let Γ be a sequence of binding of variables to orders. We define the judgement $\Gamma \vdash_{F_\omega} \sigma \in A$, with A an order.

Variables :

$$\frac{X : A \text{ occurs in } \Gamma}{\Gamma \vdash_{F_\omega} X \in A}$$

Products :

$$\frac{\Gamma \vdash_{F_\omega} \sigma \in Data \qquad \Gamma \vdash_{F_\omega} \tau \in Data}{\Gamma \vdash_{F_\omega} \sigma \Rightarrow \tau \in Data}$$

$$\frac{\Gamma, X : A \vdash_{F_\omega} \sigma \in Data}{\Gamma \vdash_{F_\omega} (X : A)\sigma \in Data}$$

Abstraction :

$$\frac{\Gamma, X : A \vdash_{F_\omega} \sigma \in B}{\Gamma \vdash_{F_\omega} [X : A]\sigma \in A \Rightarrow B}$$

Application :

$$\frac{\Gamma \vdash_{F_\omega} \sigma \in A \Rightarrow B \qquad \Gamma \vdash_{F_\omega} \tau \in A}{\Gamma \vdash_{F_\omega} (\sigma\ \tau) \in B}$$

If there exists Γ and an order A such that $\Gamma \vdash_{F_\omega} \sigma \in A$ is derivable by the above rules then σ is called a *data type scheme* and a *data type* if $A = Data$.

We introduce a congruence relation on data types schemes generated by :

$$([X : A]\sigma\ \tau) =_\beta \sigma[X/\tau]$$

with $\sigma[X/\tau]$ the result of substituting τ to free occurrences of X in σ.

Environment An environment is a sequence of binding of variables to orders or data types. If Γ is an environment then Γ_O denotes the subsequence of Γ with only the variables bound to orders. We define the notion of well-formed environment.

- The empty environment is well-formed
- If Γ is well-formed and A is an order and X does not occur in Γ then $\Gamma, X : A$ is well-formed.

- If Γ is well-formed, if $\Gamma_O \vdash_{F_\omega} \sigma \in Data$ and if x does not occur Γ then $\Gamma, x : \sigma$ is well-formed.

If Γ is an environment then we introduce the judgement $\Gamma \vdash_{F_\omega} M \in Type$. This judgement is correct if Γ is well-formed and M is an order.

Programs Let Γ be a well formed environment. We define the judgement $\Gamma \vdash_{F_\omega} t \in \sigma$ with σ a data type.

Variables :

$$\frac{x : \sigma \text{ occurs in} \Gamma}{\Gamma \vdash_{F_\omega} x \in \sigma}$$

Abstractions :

$$\frac{\Gamma, X : A \vdash_{F_\omega} t \in \sigma}{\Gamma \vdash_{F_\omega} [X : A]t \in (X : A)\sigma}$$

$$\frac{\Gamma, x : \sigma \vdash_{F_\omega} t \in \tau}{\Gamma \vdash_{F_\omega} [x : \sigma]t \in \sigma \Rightarrow \tau}$$

Application :

$$\frac{\Gamma \vdash_{F_\omega} t \in (X : A)\sigma \qquad \Gamma_O \vdash_{F_\omega} \tau \in A}{\Gamma \vdash_{F_\omega} (t\ \tau) \in \sigma[X/\tau]}$$

$$\frac{\Gamma \vdash_{F_\omega} t \in \sigma' \Rightarrow \tau \qquad \Gamma \vdash_{F_\omega} u \in \sigma \qquad \sigma =_\beta \sigma'}{\Gamma \vdash_{F_\omega} (t\ u) \in \tau}$$

In F_ω (unlike the Calculus of Constructions) there is no "type's dependence". It means that data type schemes cannot occur in orders and programs cannot occur in data types.

If there exists Γ and a data type σ such that

$$\Gamma \vdash_{F_\omega} t \in \sigma$$

is derivable by the above rules then t is called a *program*.

The system F_ω fulfills the following properties.

- A well-typed term is strongly normalizable.
- The usual concrete types (for example natural numbers or product) are defined using second-order quantification (cf [2, 7].
- There is no general fixpoint. However it is possible to code (extensionally) all recursive functions provably total in higher order arithmetic using primitive recursion on functional types.

The system F_ω is included in the Calculus of Constructions by an identification of *Data* with *Prop*.

Now let see the system F_ω as a programming language. It is possible to express and prove properties of these programs in the Calculus of Constructions. But it is then convenient to not identify *Data* with *Prop*. This will allow for example to

add one "exceptional" element in each data type without getting an inconsistent proof system.

The Calculus of Constructions with Realizations will contain the possibility of doing proofs of terms in Λ_D.

1.2 Useless proofs

Some proofs are not needed for the computation. We distinguish two kinds of such proofs.

In some cases we want just to hide some part of the proof. For example when developing a program as a proof of an existential formula $\exists x.P(x)$, we are just interested in the computational contents of the witness and not in the one of the proof of correctness.

For some propositions, we know beforehand a program that interprets their computational meaning. We want to use this program instead of interpreting an arbitrary proof. These formulae are called self-realizing in Beeson's book [1]. Examples of self-realizing propositions are Harrop's formulae and "type zero" propositions in Hayashi's system PX [10]. This replacement of proofs leads to an optimization of the extracted code.

The minimum we want is to be able to say to the system "extract or not something from this proof". This may be done with a very simple modification of the Calculus of Constructions. With just this "hiding" facility, we will be able to internally optimize the extracted code by replacing some proofs by a given program.

We now give a precise definition of our system.

1.3 The proof language

We define the language Λ_R of the proof-system. We follow the presentation of [4]. We will just consider a Calculus of Constructions with three levels (proofs, propositions and propositional types). But there will be two kinds of propositions, the informative ones of type *Spec* and the non-informative ones of type *Prop*.

As in the original calculus, a set V of variables is given and the constructors are product, abstraction and application. We want propositions that express properties of programs of F_ω. So there will be new constructors for abstraction, application and product that will mix a term of Λ_D and a term of Λ_R.

Definition 1 *The set of terms Λ_R is the smallest set containing the following constructions :*

- *Constants : Prop, Spec and Type.*
- *Variables :* $V \subset \Lambda_R$
- *Application :* $(M\ N)$ *with* $M, N \in \Lambda_R$
  $({}_D M\ N)$ *with* $M \in \Lambda_R$ *and* $N \in \Lambda_D$

- *Abstraction :* $[x : M]N$ *with* $x \in V$ *and* $M, N \in \Lambda_R$
  $[x :_D M]N$ *with* $x \in V_D$, $N \in \Lambda_R$ *and* $M \in \Lambda_D$

- *Product :* $(x : M)N$ *with* $x \in V$ *and* $M, N \in \Lambda_R$
  $(x :_D M)N$ *with* $x \in V_D$, $N \in \Lambda_R$ *and* $M \in \Lambda_D$

Notations Let x be in V and N in Λ_R or x in V_D and N in Λ_D. The term $M[x/N]$ denotes the term M where N has been substituted for free occurrences of x in M.

If x does not occur in B then the term $(x : A)B$ will be written $A \rightarrow B$ or $A \Rightarrow B$.

The rules of inference of the Calculus of Constructions with Realizations are described in the following section.

1.4 Type inference system

We extend the notion of environment in a natural way. An environment binds variables of either the proof or the programming system. If Γ is an environment and x is a variable, $x \notin \Gamma$ means that the variable x does not occur in Γ. The type inference system is the natural extension of the classical system to this new language.

There are two kinds of judgement. Let Γ be an environment and M and N two terms of Λ_R, the judgement are Γ *is Valid* and $\Gamma \vdash M \in N$.

Let $\mathcal{K}$ be one of *Prop*, *Spec* or *Type* and $\mathcal{D}$ be one of *Data* or *Type*. With each environment Γ we associate an environment Γ_D of F_ω that contains only the programming bindings $x : N$ with $x \in V_D$ and $N \in \Lambda_D$.

Environments :

$$\frac{}{[]\ \textit{is Valid}}$$

$$\frac{\Gamma \vdash M \in \mathcal{K} \qquad x \in V \qquad x \notin \Gamma}{\Gamma, x : M\ \textit{is Valid}}$$

$$\frac{\Gamma_D \vdash_{F_\omega} M \in \mathcal{D} \qquad x \in V_D \qquad x \notin \Gamma \qquad \Gamma\ \textit{is Valid}}{\Gamma, x :_D M\ \textit{is Valid}}$$

Hypothesis :

$$\frac{\Gamma\ \textit{is Valid} \qquad x : N \text{ occurs in } \Gamma \quad (N \in \Lambda_R)}{\Gamma \vdash x \in N}$$

Propositional types :

$$\frac{\Gamma\ \textit{is Valid}}{\Gamma \vdash \textit{Prop} \in \textit{Type}} \qquad \frac{\Gamma\ \textit{is Valid}}{\Gamma \vdash \textit{Spec} \in \textit{Type}}$$

Product :

$$\frac{\Gamma, x : P \vdash M \in \mathcal{K}}{\Gamma \vdash (x : P)M \in \mathcal{K}} \qquad \frac{\Gamma, x :_D P \vdash M \in \mathcal{K}}{\Gamma \vdash (x :_D P)M \in \mathcal{K}}$$

Abstraction :

$$\frac{\Gamma, x : P \vdash m \in N \qquad \Gamma, x : P \vdash N \in \mathcal{K}}{\Gamma \vdash [x : P]m \in (x : P)N}$$

$$\frac{\Gamma, x :_D P \vdash m \in N \qquad \Gamma, x :_D P \vdash N \in \mathcal{K}}{\Gamma \vdash [x :_D P]m \in (x :_D P)N}$$

Application :

$$\frac{\Gamma \vdash m \in (x : P)N \qquad \Gamma \vdash r \in Q \qquad P =_\beta Q}{\Gamma \vdash (m\ r) \in N[x/r]}$$

$$\frac{\Gamma \vdash M \in (x :_D P)N \qquad \Gamma_D \vdash_{F_\omega} r \in Q \qquad P =_\beta Q}{\Gamma \vdash (_D m\ r) \in N[x/r]}$$

1.5 Properties

We shall say that a term M is well-formed in Λ_R if and only if there exists Γ and N such that $\Gamma \vdash M \in N$. We shall call N the type of M. Terms of type *Prop* or *Spec* are called propositions and terms of type *Type* are called *propositional types.* A *type* is either a proposition or a propositional type. There are three levels of terms : the propositional types, the propositional schemes (terms of type a propositional type) and proofs (terms of type a proposition).

This system is a syntactic variant of the original Calculus of Constructions. We find again the original Calculus by an identification of *Spec* with *Prop* and either by an identification of *Data* with *Prop* or just by removing all terms of Λ_D. In particular it is straighforward to verify that all terms are strongly normalizable and that the system is consistent.

1.6 Non-informative Terms

We now distinguish between two syntactic classes of terms : terms with a positive informative contents (we shall call them informative terms) and terms with a null informative contents (we shall call them non-informative terms).

Definition 2 *Let M be a term of Λ_R. Then M is a* non-informative term *if one of the following properties holds :*

- $M = Prop$
- *M is a variable of type a non-informative term.*
- *$M = (x : P)N$ or $M = (x :_D P)N$ with N a non-informative term.*
- *$M = [x : P]N$ or $M = [x :_D P]N$ with N a non-informative term.*
- *$M = (N\ P)$ or $M = (_D N\ P)$ with N a non-informative term.*

Otherwise a term is said to be informative.

There is another caracterisation of non-informative terms.

Proposition 1 *Let M be a well-formed term in Λ_R of type N.*

If $N = Type$ then N is non-informative if and only if $N = (x_1 : A_1)\ldots(x_n : A_n)Prop$.

If $N \neq Type$ then M is non-informative if and only if N is.

An important property is that substitution does not change the informative or non-informative nature of a term.

Proposition 2 *Let M and R be terms such that M and $M[x/R]$ are well-formed. Then M is a non-informative term if and only if $M[x/R]$ is a non-informative term.*

Examples It is possible to combine *Prop* and *Spec* in several ways. For example if A is of type *Data* and P of type $A \rightarrow Prop$. It is possible to define an informative proposition whose meaning is "there exists x of type A such that $(P\ x)$" in the following way :

$$\Sigma_A P \ \equiv\ (C : Spec)((x : A)(P\ x) \rightarrow C) \rightarrow C$$

Or if A is an informative proposition (A of type *Spec*) then it is possible to hide its informative contents by considering :

$$A' \ \equiv\ (C : Prop)(A \rightarrow C) \rightarrow C$$

2 Realizability

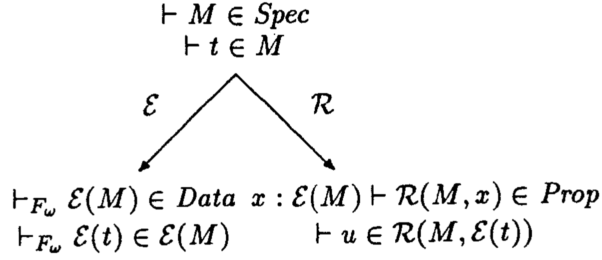
2.1 Introduction

Notions of realizability have been defined in order to capture the computational meaning of intuitionistic proofs. With each formula A of the theory is associated another formula "x realizes A". The soundness theorem establishes that if A is provable then we can find some object t such that "t realizes A".

There are many different notions of realizability [17]. In particular the relation "x realizes A" may or not be a formula of the initial theory (with x a new variable). In the relation of realizability, the object x may or not be restricted to belong to a language with only total objects.

Our notion of realizability may be viewed as an extension to the Calculus of Constructions of the modified realizability of HA_ω. Its main characteristics are that the realizations are typed in the system F_ω (and then are strongly normalizable) and that the formula "x realizes M" (that we shall write $\mathcal{R}(M, x)$) is a formula of the Calculus.

Remark We are going to define on terms the function of extraction $\mathcal{E}$ and the function of realizability $\mathcal{R}$. We show on a picture the relationship between, *Data*, *Prop*, *Spec* extraction and realizability.



2.2 Extraction

We define a function $\mathcal{E}$ of extraction for informative terms, inductively on the structure of the term. This fonction is defined for the three levels of terms of the Calculus. The function $\mathcal{E}$ removes the non-informative part of the term and the type's dependences in order to get a F_ω-term. At each variable x is associated by extraction a new variable $\overline{x}$. We do not write the $_D$ subscripts in the terms and take as a notation that if $M \in \Lambda_D$ and $x \in V_D$ then $\mathcal{E}(M) = M$ and $\overline{x} = x$.

Definition 3 (Extraction)
Propositional types *Let M be an informative propositional type.*

- *If $M = Spec$ then $\mathcal{E}(M) = Data$.*
- *If $M = (X : A)B$ then B is an informative propositional type.*
  If A is an informative propositional type or an order then $\mathcal{E}(M) = \mathcal{E}(A) \Rightarrow \mathcal{E}(B)$ else $\mathcal{E}(M) = \mathcal{E}(B)$

Propositional schemes *Let M be an informative propositional scheme.*

- *If M is a variable X then $\mathcal{E}(M) = \overline{X}$.*
- *If $M = (x : A)B$ then B is an informative propositional scheme.*
  If A is informative or in Λ_D then $\mathcal{E}(M) = (\overline{x} : \mathcal{E}(A))\mathcal{E}(B)$ else $\mathcal{E}(M) = \mathcal{E}(B)$.
- *If $M = [x : A]B$ then B is an informative propositional scheme.*
  If A is an informative propositional type or an order then $\mathcal{E}(M) = [\overline{x} : \mathcal{E}(A)]\mathcal{E}(B)$ else $\mathcal{E}(M) = \mathcal{E}(B)$.
- *If $M = (A\ B)$ then A is an informative propositional scheme.*
  If B is an informative propositional scheme or a data type scheme then $\mathcal{E}(M) = (\mathcal{E}(A)\ \mathcal{E}(B))$ else $\mathcal{E}(M) = \mathcal{E}(A)$.

Proofs *Let M be an informative proof.*

- *If M is a variable x then $\mathcal{E}(M) = \overline{x}$.*
- *If $M = [x : A]B$ then B is an informative proof.*
  If A is informative or in Λ_D then $\mathcal{E}(M) = [\overline{x} : \mathcal{E}(A)]\mathcal{E}(B)$ else $\mathcal{E}(M) = \mathcal{E}(B)$.

- *If $M = (A\ B)$ then A is an informative proof.*

  If B is informative or in Λ_D then $\mathcal{E}(M) = (\mathcal{E}(A)\ \mathcal{E}(B))$ else $\mathcal{E}(M) = \mathcal{E}(A)$.

The function $\mathcal{E}$ is extended to the environments. The main result about extraction is :

Theorem 1 *If $\Gamma \vdash M \in N$ with M an informative term then $\mathcal{E}(\Gamma) \vdash_{F_\omega} \mathcal{E}(M) \in \mathcal{E}(N)$*

The proof proceeds by induction on the length of the derivation. The main point is the commutation of extraction with respect to substitution, namely if M and N are informative terms then

$$\mathcal{E}(M[x/N]) = \mathcal{E}(M)[x/\mathcal{E}(N)].$$

2.3 Definition and properties of realizability

In general the notion of realizability is defined for propositions. But in the Calculus of Constructions, in a proposition may appear variables, propositional types and propositional schemes.

The general scheme for the definition of the formula f **realizes** $A \Rightarrow B$ is :

$$\forall x.x \textbf{ realizes } A \Rightarrow f(x) \textbf{ realizes } B.$$

This definition is extended to dependent products and quantification over propositional types in the Calculus of Constructions.

Let us see how to define the notion of realizability for variables of type a propositional type, for example *Spec*. Let M be of type *Spec*, then $\mathcal{E}(M)$ is of type *Data*, a term t realizes M, if it is of type $\mathcal{E}(M)$ and if there exists a proof of the formula $\mathcal{R}(M,t)$. With realizability, we interpret M as a subset of terms of type $\mathcal{E}(M)$. In order to interpret a variable X of type *Spec*, we need two new variables. A variable C of type *Data* (the type of extracted objects of proofs of X) and a variable P of type $C \rightarrow$ *Prop* (the predicate satisfied by extracted objects of proofs of X). This treatment of higher-order quantification is analogous with the one of J. -L. Krivine in the system AF_2 [12] but we are in a typed framework.

The notion of realizability is defined for types (that is propositions or propositional types).

If M is an informative proposition or a propositional type, we define $\mathcal{R}(M,r)$ (to be read as r realizes M), with r a free variable of type $\mathcal{E}(M)$ not occurring in M. The formula $\mathcal{R}(M,r)$ will be a non-informative formula. If t is a term then we write $\mathcal{R}(M,t)$ for $\mathcal{R}(M,r)[r/t]$.

This is the general case. But we need also to define a notion of realizability for non-informative types. If M is a non informative type, the analogous of the predicate $\mathcal{R}(M,r)$ is a proposition or propositional type $\mathcal{R}(M)$. If M is a propositional scheme then we define a new term $\mathcal{R}(M)$. If M is an informative proposition then M is also a propositional scheme and we take as a definition $\mathcal{R}(M,r) = (\mathcal{R}(M)\ r)$.

We give some examples of the definition of $\mathcal{R}$.

Spec Let r be a variable of type *Data* then $\mathcal{R}(Spec, r) = r \rightarrow Prop$.

Variable If X is a variable of type M then $\overline{X}$ is a variable of type $\mathcal{E}(M)$ and we define $\mathcal{R}(X)$ to be a variable (still of name X) of type $\mathcal{R}(M, \overline{X})$.

Product Let $P = (x : M)N$ with N an informative proposition and r be a variable of type $\mathcal{E}(P)$.

- If M is informative then $\mathcal{R}(P, r) = (\overline{x} : \mathcal{E}(M))(x : \mathcal{R}(M, \overline{x}))\mathcal{R}(N, (r\ \overline{x}))$
- If M is non-informative then $\mathcal{R}(P, r) = (x : \mathcal{R}(M))\mathcal{R}(N, r)$
- If M is in Λ_D then $\mathcal{R}(P, r) = (x : M)\mathcal{R}(N, (r\ \overline{x}))$

Abstraction Let $P = [x : M]N$ with N an informative propositional scheme.

- If M is an informative propositional type then $\mathcal{R}(P) = [\overline{x} : \mathcal{E}(M)][x : \mathcal{R}(M, \overline{x})]\mathcal{R}(N)$
- If M is a non-informative propositional type then $\mathcal{R}(P) = [x : \mathcal{R}(M)]\mathcal{R}(N)$
- If M is an informative proposition then $\mathcal{R}(P) = [\overline{x} : \mathcal{E}(M)]\mathcal{R}(N)$
- If M is a non-informative proposition then $\mathcal{R}(P) = \mathcal{R}(N)$

The other cases may be deduced easily. The fonction $\mathcal{R}$ is extended to the environments in a natural way.

It is possible to see the case of non-informative proposition and data types as particular cases of informative propositions. Let T be a data type inhabited by one element t. If M is a non-informative proposition, it will be possible to define $\mathcal{E}(M) = T$ and $\mathcal{R}(M, r) = (r =_T t) \wedge \mathcal{R}(M)$. Let *True* be a non-informative tautology. If M is a data type then it will be possible to define $\mathcal{R}(M, r) = True$.

Definition 4 *A term t* realizes *(is a* realization *of) an informative type P (in an environment Γ) if the type $\mathcal{R}(P, t)$ has a proof (in Γ).*

An informative type P is realizable *(in an environment Γ) if there exists a term that realizes it (in Γ).*

Our notion of realizability is sound.

Theorem 2 (soundness) *If t is a proof of the informative proposition M in an environment Γ then $\mathcal{E}(t)$ realizes M in $\mathcal{R}(\Gamma)$.*

The proof is by induction on the length of the derivation of $\Gamma \vdash t \in M$.

Remarks In formulas $\mathcal{R}(M, r)$, the types only depend on programming objects and never on proof-objects. We have $\mathcal{R}(M) = M$, if M is a non-informative proposition that contains neither informative subterms nor sub-propositions that depends on proofs (they may depend on programs). In any case, if M is non-informative and provable then $\mathcal{R}(M)$ is also provable.

3 Examples

In this section we show how some propositions are realized. In order to be more readable, we do not write the $_D$ indexes in terms, there is no ambiguity to do that. We also adopt more convenient notations instead of $\overline{x}$ for extracted bound variables.

We study the following examples. Leibniz's equality is a proposition that is realized by a fixed term independently of its proof. We show the realization of the existential type and prove the consistency of the axiom of choice. We present also the predicate of realizability of disjunctive formulae, and the example of the natural numbers with the predicate of induction.

Remark An application of a notion of realizability is to prove the consistency of some axioms. Let A be a proposition, then if there exists a term that realizes A then A is consistent with the theory. Suppose that there exists a derivation of $A \vdash \bot$ then because A is realizable, $\bot$ is also realizable, this implies that it is provable.

The notion of realizability is not only useful to extract programs from proofs but also to add consistent axioms that will be used in the development of the proof. In this section, we shall show several axioms that may be added consistently to the theory. Some of them becomes tautology if we identify *Prop* and *Spec.* But they are necessary in order to optimize the extracted code.

3.1 About *Prop* and *Spec*

The original constant *Prop* of the Calculus of Constructions is now the constant *Spec.* In the syntax, these two constants are not related. But in a certain sense we have the following relation $Prop \subset Spec$. This relation may be explicited. We define a term S of type $Prop \to Spec$.

$$S \equiv [A : Prop](C : Spec)(A \to C) \to C$$

It is easy to find a proof of

$$(A : Prop)A \to (S\ A).$$

It is not possible to prove the non-informative proposition :

$$\Pi = (A : Prop)(S\ A) \to A$$

but this proposition is realizable. The proposition $\mathcal{R}(\Pi)$ is equal to :

$$(A : Prop)((C : Prop)(\mathcal{R}(A) \to C) \to C) \to \mathcal{R}(A)$$

that is provable because $\mathcal{R}(A)$ is of type *Prop.* trivially true. So Π may be added consistently as an axiom. So S is of type $Prop \to Spec$ and may be viewed as

an injection of the class of non-infomative propositions in the class of informative propositions.

When we define a second order type like conjonction or disjunction, it is possible to define it by quantification over a propositional variable of type *Prop* or *Spec*. For example with A and B, two propositions we have two kinds of disjonction :

$$A \vee B \equiv (C : Prop)(A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$$

$$A + B = (C : Spec)(A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$$

The proposition $A \vee B$ is non-informative while $A + B$ is informative. Using S it is possible to prove

$$A + B \Rightarrow A \vee B$$

and a similar proposition for every second order type.

We may wonder whether the converse is realizable. In general, it is false. For example

$$A \vee B \Rightarrow A + B$$

is not realizable.

But it is true for the conjonction of two non-informative propositions A and B. We define

$$A \wedge B \equiv (C : Prop)(A \rightarrow B \rightarrow C) \rightarrow C$$

$$A \times B \equiv (C : Spec)(A \rightarrow B \rightarrow C) \rightarrow C$$

The proposition

$$A \wedge B \Rightarrow A \times B$$

is provable using the two projections of the non-informative conjonction.

This corresponds to the notion of "self-realizing" formula (cf Beeson [1]). A formula M is self-realizing if there exists a term ϕ such that ϕ realizes M as soon as M is provable. We want to replace the term extracted from an arbitrary proof of M by ϕ. A way to implement this is to introduce a non-informative version M' of M (by quantification over *Prop*) and to add an axiom Φ of type $M' \rightarrow M$ that will be realized by ϕ. When we need a proof of M, we shall use $(\Phi\ m)$ with m a non-informative proof of M'. The extracted term of this proof of M will be exactly ϕ. The class of propositions that may be realized independently of their proofs can be dynamically increased.

3.2 A non-informative proposition : equality

Leibniz's equality is a simple example of "self-realizing" proposition. We are in the environment :

$$A : Spec,\ a : A,\ b : B$$

We define :

$$Eq \equiv (P : A \rightarrow Spec)(P\ a) \rightarrow (P\ b)$$

and

$$a =_A b \equiv (P : A \rightarrow Prop)(P\ a) \rightarrow (P\ b)$$

We have :

$$\mathcal{E}(Eq) = (C : Data)C \rightarrow C$$

and if r is a variable of type $\mathcal{E}(Eq)$:

$$\mathcal{R}(Eq, r) = (C : Data)(P : \mathcal{E}(A) \rightarrow C \rightarrow Prop)(x : C)(P\ \mathcal{E}(a)\ x) \rightarrow (P\ \mathcal{E}(b)\ (r\ C\ x))$$

The proposition $a =_A b$ is non-informative and

$$\mathcal{R}(a =_A b) = (P : \mathcal{E}(A) \rightarrow Prop)(P\ \mathcal{E}(a)) \rightarrow (P\ \mathcal{E}(b))$$

We shall use the same notations for equality of elements of type B of type *Data* and shall write :

$$\mathcal{R}(a =_A b) = (\mathcal{E}(a) =_{\mathcal{E}(A)} \mathcal{E}(b))$$

We remark that it is consistent to assume $a =_A b$ if $\mathcal{E}(a)$ and $\mathcal{E}(b)$ are equal. This allow to forget about equality of proofs terms.

It is easy to prove the following properties :

$$\mathcal{E}(a) =_{\mathcal{E}(A)} \mathcal{E}(b) \rightarrow \mathcal{R}(Eq, [C : Data][x : C]x)$$

$$\mathcal{R}(Eq, r) \rightarrow \mathcal{E}(a) =_{\mathcal{E}(A)} \mathcal{E}(b)$$

Each proof of equality can be replaced in the extracted program by the identity. So we do not need a computational proof of equality but only a logical one. We will use non-informative equality during the program's development process. We take as an (informative) axiom *eqspec* of type :

$$a =_A b \quad \rightarrow \quad (P : A \rightarrow Spec)(P\ a) \rightarrow (P\ b)$$

The variable $\overline{eqspec}$ may appear in the extracted programs but it is correct to replace it by the identity. So that the final program will be valid in an environment without any occurrence of *eqspec*.

3.3 The type absurdity

It is natural to want a proposition "absurdity" that is non-informative. We define :

$$\bot = (C : Prop)C$$

We then need the property :

$$except : (C : Spec)\bot \rightarrow C$$

The proposition $\mathcal{R}(except, r)$ is equal to :

$$(C : Data)(P : C \rightarrow Prop)\bot \rightarrow (P\ (r\ C))$$

This proposition is provable whatever r is. The problem is r has to be typed of type $(C : Data)C$. It is not possible to find a closed term of this type. We will keep the variable $\overline{except}$ in the extraction environment. Because the separation between *Prop* and *Data*, the system is still consistent. It is also possible to prove (as a meta result) that a term extracted in a consistent environment cannot be of the form $(\overline{except}\ u_1\ \dots\ u_p)$.

3.4 Existential proposition

We show that realizing the proposition $\exists x\ : A.P(x)$ with P a non-informative proposition gives an object t of type A such that $P(t)$ is realizable. We are in the environment :

$$A : Data,\ \Phi : A \rightarrow Prop$$

The existential intuitionistic quantification is defined by :

$$\Sigma_A(\Phi)\ \equiv\ (C : Spec)((x : A)(\Phi\ x) \rightarrow C) \rightarrow C.$$

3.4.1 Realizability

The extracted type is $(C : Data)(A \rightarrow C) \rightarrow C$ that we write $\widetilde{A}$. This is not exactly A but a type similar to A. More precisely we call ι_A the injection from A into $\widetilde{A}$ and π_A the projection from $\widetilde{A}$ on A. Formally :

$$\iota_A\ \equiv\ [a : A][C : Data][h : A \rightarrow C](h\ a)$$

$$\pi_A\ \equiv\ [a : \widetilde{A}](a\ A\ [x : A]x)$$

It is not exactly an isomorphism because

$$(\pi_A\ (\iota_A\ x)) =_A x$$

is provable but not

$$(\iota_A\ (\pi_A\ x)) =_{\widetilde{A}} x.$$

Actually there are examples of functional types A and closed terms x such that $(\iota_A\ (\pi_A\ x))$ and x are not convertible. But this is not a problem for what we want to prove.

Let r be a variable of type $\widetilde{A}$ then $\mathcal{R}(\Sigma_A(\Phi), r)$ is the following proposition :

$$(C : Data)(P : C \rightarrow Prop)(f : A \rightarrow C)((x : A)\mathcal{R}((\Phi\ x)) \rightarrow (P\ (f\ x))) \rightarrow (P\ (r\ C\ f))$$

It is easy to prove the following propositions :

$$(a : A)\mathcal{R}((\Phi\ a)) \rightarrow \mathcal{R}(\Sigma_A(\Phi), (\iota_A\ a))$$

$$(a : \widetilde{A})\mathcal{R}(\Sigma_A(\Phi), a) \rightarrow \mathcal{R}((\Phi\ (\pi_A\ a)))$$

Proposition 3 *The proposition $\Sigma_A(\Phi)$ is realizable if and only if there exists some element a of type A such that $(\Phi\ a)$ is realizable.*

3.5 A subset type

Using existential quantification and non-informative propositions, we get a construction analogous with the subset type in some versions of Martin-Löf's type systems.

It is also possible to build a real subset type, it means a type $\{A|P\}$ that is realized by objects of type A that realizes P. Assume that A is of type *Data* and that P is of type $A \to$ *Prop*. We want the following properties :

$$
\begin{array}{lll}
\{A|P\} & : & Spec \\
subintro & : & (x:A)(P\ x) \to \{A|P\} \\
subelim & : & \{A|P\} \to \ (C:Spec)((x:A)(P\ x) \to C) \to C \\
subreduce & : & (x:A)(h:(P\ x))(C:Spec)(f:(x:A)(P\ x) \to C) \\
 & & (subelim\ (subintro\ x\ h)\ C\ f) =_C (f\ x\ h) \\
subind & : & (t:\{A|P\})(Q:\{A|P\} \to Prop) \\
 & & ((x:A)(h:(P\ x)) \to (Q\ (subintro\ x\ h))) \to (Q\ t)
\end{array}
$$

We show that it is possible to realize all these axioms. In order to realize $\{A|P\}$ we need a type C and a predicate Q of type $C \to$ *Prop*. We take $C = A$ and $Q = \mathcal{R}(P)$. In this interpretation a realizator of $\{A|P\}$ is exactly an element x of type A such that $(P\ x)$ is realizable.

The axiom *subintro* is realized by a term f of type $A \to A$ such that

$$(x:A)(\mathcal{R}(P)\ x) \to (\mathcal{R}(P)\ (r\ x))$$

We take $f = [x:A]x$.

The axiom *subelim* is of type $\{A|P\} \to \Sigma_A P$. It is realized by a term g of type $A \to \widetilde{A}$ such that

$$(x:A)(\mathcal{R}(P)\ x) \to \mathcal{R}(\Sigma_A P, (g\ x))$$

We have already shown that the term $g = \iota_A$ works.

The axiom *subreduce* is realizable because (by definition of ι_A) the following proposition is provable :

$$(x:A)(C:Data)(f:A \to C)(\iota_A\ x\ C\ f) =_C (f\ x)$$

Finally $\mathrm{R}(subind)$ is the following (trivially provable) proposition :

$$(t:A)(\mathcal{R}(P)\ t) \to (Q:A \to Prop)((x:A)(\mathcal{R}(P)\ x) \to (Q\ x)) \to (Q\ t)$$

It is very easy to prove the equivalence between $\Sigma_A P$ and $\{A|P\}$. The subset type avoid the use of injection and projection in the extracted programs. We shall write $\{x:A|P\}$ with x free in P instead of $\{A|[x:A]P\}$.

We may internalize the equivalence between the provability of an informative proposition A and the existence of a program in $\mathcal{E}(A)$ that realizes A.

Proposition 4 *Let A be an informative proposition (A of type Spec), the following propositions are realizable :*

$$A \Rightarrow \{r:\mathcal{E}(A)|\mathcal{R}(A,r)\} \qquad \{r:\mathcal{E}(A)|\mathcal{R}(A,r)\} \Rightarrow A$$

The axiom of choice is just a particular case of this property.

3.5.1 Axiom of choice

Using the previous results it is possible to realize the axiom of choice. In the environment :

$$A : \mathit{Data},\ B : \mathit{Data}$$

$$\Psi : A \rightarrow \mathit{Prop},\ \Phi : A \rightarrow B \rightarrow \mathit{Prop}$$

The axiom of choice can be stated as follows :

$$((x : A)(\Psi\ x) \rightarrow \{y : B|(\Phi\ x\ y)\}) \rightarrow\ \{f : A \rightarrow B|(x : A)(\Psi\ x) \rightarrow (\Phi\ x\ (f\ x))\}$$

It is easy to realize it by the identity function on $A \rightarrow B$.

Remark This shows an example of proposition that is not provable in the Calculus but that may be realized. It is essential for Ψ to be a non-informative proposition and for B to be a data type.

3.6 Disjunctive propositions

Let A and B be non-informative propositions. Let $A + B$ be the intuitionistic second order disjunction then :

$$\begin{aligned} A + B &= (C : \mathit{Spec})(A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C \\ \mathcal{E}(A + B) &= (C : \mathit{Data})C \rightarrow C \rightarrow C \\ &= \mathit{bool} \end{aligned}$$

If u proves an intuitionistic disjunction, then the extracted term will be a boolean. It is not possible to know what boolean it will be without looking at the proof of $A + B$.

But there is a variant of the disjunction of two propositions introduced by S. Hayashi and which may be taken as a non-informative proposition. Let A and B be two non-informative propositions, and e be a boolean. Then we define the proposition "If e then A else B". More precisely we defined $\mathit{cond}(e, A, B)$ to be the following non-informative formula :

$$(P : \mathit{bool} \rightarrow \mathit{Prop})(A \rightarrow (P\ \mathit{true})) \rightarrow (B \rightarrow (P\ \mathit{false})) \rightarrow (P\ e)$$

Let COND be the following informative proposition :

$$\mathit{cond}(e, A, B) \rightarrow (P : \mathit{bool} \rightarrow \mathit{Spec})(A \rightarrow (P\ \mathit{true})) \rightarrow (B \rightarrow (P\ \mathit{false})) \rightarrow (P\ e)$$

And we would like to take COND as an axiom. We need to find a realization of COND. We get easily that $\mathcal{E}(\mathit{COND}) = \mathit{bool}$. And now with r a new variable of type bool, $\mathcal{R}(\mathit{COND}, r)$ is the following proposition :

$$\begin{aligned} \mathit{cond}(e, \mathcal{R}(A), \mathcal{R}(B)) &\rightarrow (C : \mathit{Data})(P : \mathit{bool} \rightarrow C \rightarrow \mathit{Prop}) \\ &\quad ((x_1 : C)\mathcal{R}(A) \rightarrow (P\ \mathit{true}\ x_1)) \\ &\quad \rightarrow ((x_2 : C)\mathcal{R}(B) \rightarrow (P\ \mathit{false}\ x_2)) \\ &\rightarrow (P\ e\ (r\ C\ x_1\ x_2)) \end{aligned}$$

It is very easy to prove the proposition $\mathcal{R}(COND, e)$. In this case, the realization of the proposition appears already inside the proposition and we do not need the proof in order to find it.

It is possible, using the axiom *COND* to prove the following proposition :

$$(A + B) \Rightarrow \{e : bool | (cond\ e\ A\ B)\}$$

3.7 The natural numbers

The type of the natural numbers as iterators is :

$$nat \equiv (C : Data)C \to (C \to C) \to C.$$

We define the two constructors zero (o) and the successor function (s). It is well known that there exists an operator of primitive recursion *rec* of type

$$nat \to (C : Data)C \to (nat \to C \to C) \to C$$

Let $\mathcal{N}$ be the induction principle on natural numbers, namely :

$$[n : nat](P : nat \to Prop)(P\ o) \to ((u : nat)(P\ u) \to (P\ (s\ u))) \to (P\ n).$$

Let C be of type *Data*, c of type C and f of type $nat \to C \to C$. It is possible to prove the following results :

$$(rec\ o\ C\ c\ f) =_C c$$

$$(n : nat)(\mathcal{N}\ n) \to (rec\ (s\ n)\ C\ c\ f) =_C (f\ n\ (rec\ n\ C\ c\ f))$$

It is not possible to prove $(n : nat)(\mathcal{N}\ n)$ in the Calculus of Constructions neither to realize it. We need the fact that $(n\ nat\ O\ S)$ is equal to n. This fact may be proved for closed terms. A similar relation may be false for other second order types.

We propose a solution that will work for every second order type defined by a specification of its constructors. We introduce using our subset type, a type NAT, of objects of type *nat* that satisfy $\mathcal{N}$.

$$NAT \equiv \{n : nat | (\mathcal{N}\ n)\}$$

Because we have proofs of

$$(\mathcal{N}\ o) \quad \text{and} \quad (u : nat)(\mathcal{N}\ u) \to (\mathcal{N}\ (s\ u))$$

it is easy to find a term O of type NAT, such that $\mathcal{E}(O) = o$ and a term S of type $NAT \to NAT$ such that $\mathcal{E}(S) = s$.

The terms of type NAT behaves like terms of type *nat* but they contain an extra logical information. Let *Peano* be the following proposition :

$$(n : NAT)(P : NAT \to Prop)(P\ O) \to ((u : nat)(P\ u) \to (P\ (S\ u))) \to (P\ n).$$

Then *Peano* is realizable because :

$$\mathcal{R}(Peano) = (n : nat)(\mathcal{N}\ n) \rightarrow (\mathcal{N}\ n)$$

We are often more interested in realizing the following proposition :

$$(n : NAT)(P : NAT \rightarrow Spec)(P\ O) \rightarrow ((u : nat)(P\ u) \rightarrow (P\ (S\ u))) \rightarrow (P\ n).$$

This proposition is realized by the operator of primitive recursion *rec*.

4 Related works

There are many relations between this work and PX [10], Nuprl [3], ITT (Martin-Löf's system [15]) or AF_2 [12, 16].

The idea of using a syntactic notion of non-informative propositions in order to optimize the realizability comes from the system *PX*. This system has also a clear distinction between proofs and programs, and among the proofs between informative and non-informative ones (non-informative propositions are called "type zero"). In this system the realizations may not terminate. There is a predicate E in the logic whose meaning is termination. The definition of realizability is such that x **realizes** $A \Rightarrow E(x)$ is always provable.

In Nuprl [3], the extra information is hidden using the subset type $\{x : A|P(x)\}$. In this system if a is of type $\{x : A|P(x)\}$ then a is also of type A. So given a, the type of a is undecidable. In our system, there is a difference between the proof of $\{x : A|P(x)\}$ and the extracted term of type A. In a judgement a **in** A in Nuprl, a may be an arbitrary term of the λ-calculus. In particular it may contains non-terminating subterms. The fact that this term is normalizable comes from the provability of the judgement. The judgement a **in** A has to be read "a **realizes** A". The notion of realizability satisfies the fact that a realization terminates.

The idea of separating types (what we call *Spec*) and propositions has been proposed for Martin-Löf's systems [14]. It seems to be the right way to express the subset type. In Nuprl and ITT, there are no distinctions between proofs and programs. This distinction becomes necessary if we want to realize more axioms without getting an inconsistent environment.

Our notion of realizability is related to AF_2 system. The system AF_2 is a second order predicate logic. One difference is that in AF_2, r realizes $\forall x.P$ is the proposition $\forall x.$ (r realizes P) (and not $\forall x.((r\ x)$ realizes P)). To find this notion of realizability in (the extension to the hierarchy of type of) our system, it is sufficient to say that the first-order objects are in some class U of type *Type*. But the main difference is the way this realizability is used. To develop a program p in AF_2, you first give an equational specification of this program. Suppose for example that p is a function on natural numbers, you have to realize the formula $\forall x.\mathcal{N}(x) \rightarrow \mathcal{N}(p\ x)$, where $\mathcal{N}$ is (for example) the induction principle on natural numbers. The realization gives you a λ-term that computes p.

Conclusion

We have defined and proved sound an internal notion of realizability for a syntactic variant of the Calculus of Constructions. The critical point in such a notion is to have a definition that commutes with substitution.

We think that the Calculus of Constructions with Realizations is an adequate "Programming Logic". In particular it is necessary to distinguish between the programming and the proof language, and to distinguish inside the proof language between informative and non-informative propositions. Because the proposition that says when a program realizes a specification is a formula of the Calculus of Constructions, we may use the system in order to prove that some term is a realization of some proposition.

An experimental implementation of the system has been done, using the current implementation of the Calculus of Constructions realized at INRIA by G. Huet in the language CAML. There has been no difficulty to adapt our previous examples of development of programs (Quicksort for lists, a formating program ...[13]) to this system. It is now possible to really get functional programs extracted from these proofs and to execute them.

The system presented in this paper extracts programs typed in the system F_ω which contains only terminating terms. But our notion of realizability is quite independent of the programming language. In particular we may extend it with a fixpoint. Then the proof of termination will no more appear inside the program but will be a consequence of the realizability proposition. Such a treatment is related to P. Dybjer's work [9] and to J.-L. Krivine's AF_2 [12, 16]. We will explain this in a forthcoming paper.

Acknowledgements

I am grateful to Gérard Huet for supervising this work and to Thierry Coquand for many helpful discussions.

A A proved example

We give the complete development of an algorithm for the search of the smallest element in a non-empty list in the Calculus of Constructions with Realizations. This is a session for the machine. Comments are written in italics. The command **Inductive** is a "macro" that builds simultaneously a type and its constructors.
Absurdity and negation :

```
Definition void (C:Prop)C.
Syntax void "{}".

Axiom except : (C:Spec){}->C.

Definition not [A:Prop]A->{}.
```

```
Syntax not "~_".
```

Informative disjunction of two non-informative propositions :

```
Syntax sumbool "{_}+{_}".
Inductive sumbool [A,B:Prop] : Spec
    = left : A ->({A}+{B})
    | right : B->({A}+{B}).
```

Non informative conjonction of two non-informative propositions :

```
Syntax and "_/\\_".
Inductive and [A,B:Prop] : Prop  = conj : A ->B ->(A/\B).
```

Existential quantification :

```
Syntax sig "<_>Sig(_)".
Inductive sig [A:Data;P:A->Prop] : Spec
       = exist : (x:A)(P x)->(<A>Sig(P)).
```

Equality :

```
Syntax eqd "<_>_=_".
Inductive eqd   [A:Data;x:A] : A->Prop = refl_equal : <A>x=x.

Axiom eq_spec : (A:Data)(x,y:A)(P:A->Spec)(<A>x=y)->(P x)->(P y).
```

Let R be a total, reflexive and transitive relation on a data type A.

```
Variable A : Data.
Variable R : A->A->Prop.
Axiom Rtot : (x,y:A){(R x y)}+{(R y x)}.
Axiom Rrefl : (a:A)(R a a).
Axiom Rtrans : (a,b,c:A)(R a b)->(R b c)->(R a c).
```

The data type of lists of elements of A :

```
Inductive List : Data
    = nil : List
    | cons : A->List->List.

Axiom List_ind : (l:List)(P:List->Spec)
      (P nil)->((a:A)(m:List)(P m)->(P (cons a m)))->(P l).

Axiom nil_cons : (a:A)(m:List)~(<List>nil=(cons a m)).
```

Definition of the predicate inL ($(inL\ a\ l) \equiv a \in l$): the smallest predicate such that $a \in a.m$ and $a \in m \;\Rightarrow\; a \in b.m$.

```
Inductive inL [a:A] : List->Prop
    = inL_tl : (b:A)(m:List)(inL a m)->(inL a (cons b m))
    | inL_hd : (m:List)(inL a (cons a m)).
```

Definition of the predicate infL ((infL a l) $\equiv a \ll l$): the smallest predicate such that $a \ll nil$ and $a \ll m \ \wedge (R\ a\ b) \Rightarrow \ a \ll b.m$.

```
Inductive infL [a:A] : List->Prop
    = infL_nil : (infL a nil)
    | infL_cons : (b:A)(m:List)
                  (R a b)->(infL a m)->(infL a (cons b m)).
```

Lemma : A list is or not equal to nil.

```
Lemma nil_or_not (l:List){<List>nil=l}+{~<List>nil=l}
Proof
   [l:List]
    (List_ind l [x:List]{<List>nil=x}+{~<List>nil=x}
       (left <List>nil=nil ~<List>nil=nil (refl_equal List nil))
       [a:A][m:List][H:{<List>nil=m}+{~<List>nil=m}]
         (right <List>nil=(cons a m) ~<List>nil=(cons a m)
                (nil_cons a m))).
```

We do not write the explicit proofs of the following non informative lemmas

```
Name Min [l:List][a:A]((infL a l)/\(inL a l)).

Lemma unit (a:A)(Min (cons a nil) a)
Proof ...

Lemma cons1 (a,x:A)(m:List)(Min m x)->(R a x)->(Min (cons a m) a)
Proof ....

Lemma cons2 (a,x:A)(m:List)(Min m x)->(R x a)->(Min (cons a m) x)
Proof ....
```

To prove : $\forall l : List.\ l \neq nil \ \Rightarrow \ \exists a : A.(a \ll l \wedge a \in l)$

```
Name ExMin [l:List](~<List>nil=l)-><A>Sig(Min l).

Theorem min_search  (l:List)(ExMin l)
Proof [l:List]
   (List_ind l ExMin
        [H:~<List>nil=nil]
        (except <A>Sig(Min nil) (H (refl_equal List nil)))
        [a:A][m:List][H:(ExMin m)][H0:~<List>nil=(cons a m)]
         (nil_or_not m <A>Sig(Min (cons a m))
```

```
[H1:<List>nil=m]
  (eq_spec List nil m [x:List]<A>Sig(Min (cons a x))
           H1 (exist A (Min (cons a nil)) a (unit a)))
[H1:~<List>nil=m]
  (H H1 <A>Sig(Min (cons a m))
     [x:A][H2:(Min m x)]
      (Rtot a x <A>Sig(Min (cons a m))
           [H3:(R a x)]
    (exist A (Min (cons a m)) a (cons1 a x m H2 H3))
          [H3:(R x a)]
    (exist A (Min (cons a m)) x (cons2 a x m H2 H3)))))).
```

The automatically extracted context is :

```
* [except :(C:Data)C]
sumbool  ==> (C:Data)C->C->C
left     ==> [C:Data][c1:C][c2:C]c1
right    ==> [C:Data][c1:C][c2:C]c2
sig      ==> [A:Data](C:Data)(A->C)->C
exist    ==> [A:Data][x':A][C:Data][c1:A->C](c1 x')
* [eq_spec :(A:Data)A->A->(P:Data)P->P]
* [A :Data]
* [Rtot :A->A->sumbool]
List = (C:Data)C->(A->C->C)->C          : Data
nil  = [C:Data][c1:C][c2:A->C->C]c1     : List
cons = [a:A][l:List][C:Data][c1:C][c2:A->C->C](c2 a (l C c1 c2))
     : A->List->List
* [List_ind : List->(P:Data)P->(A->List->P->P)->P]
nil_or_not ==>
  [l:List]
  (List_ind l sumbool left [a:A][m:List][H:sumbool]right)
ExMin      ==> (sig A)
min_search ==>
  [l:List](List_ind l ExMin (except (sig A))
            [a:A][m:List][H:ExMin]
             (nil_or_not m (sig A)
               (eq_spec List nil m (sig A) (exist A a))
               (H (sig A)
                  [x:A](Rtot a x (sig A) (exist A a) (exist A x)))))
```

We may replace eqspec by `[A:Data][x,y:A][C:Data][x:C]x` and get the following program :

```
[l:List](List_ind l ExMin (except (sig A))
           [a:A][m:List][H:ExMin]
             (nil_or_not m (sig A) (exist A a)
```

```
(H (sig A)
   [x:A](Rtot a x (sig A) (exist A a) (exist A x)))))
```

Relation with an ML program There is an analogy between second order types and ML's concrete types. For example the type `sig` is a type with one unary constructor.

```
type 'A sig  = exist of 'A;;
```

In the second order λ-calculus it is possible to apply a term of type `(sig A)` to a type `C` and a fonction of type `A->C`. This corresponds to the ML construction `match`. This analogy holds also for lists and booleans. We remark that the operation of matching a boolean is just the construction `if`. We get explicitely the following program of type `'A list-> 'A sig`

```
let min_search  =
  function [] -> failwith "except"
      |a::m -> if m = nil then (exist a)
               else let H = (min_search m) in
        (match H with
            (exist x)-> if R a x then exist a else exist x);;
```

References

[1] M.J. Beeson. *Foundations of Constructive Mathematics, Methamathematical Studies.* Springer-Verlag, 1985.

[2] C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39, 1985.

[3] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[4] Th. Coquand. Metamathematical investigations of a calculus of constructions, part I: syntax. 1987. To appear as INRIA Technical Report.

[5] Th. Coquand. *Une Théorie des Constructions.* Thèse de 3$^{\text{ème}}$ cycle, Université Paris 7, 1985.

[6] Th. Coquand and G. Huet. The calculus of constructions. *Information and Control*, 76, 1988.

[7] Th. Coquand and G. Huet. Concepts mathématiques et informatiques formalisés dans le calcul des constructions. In The Paris Logic Group, editor, *Logic Colloquium '85*, North-Holland, 1987.

[8] Th. Coquand and G. Huet. Constructions: a higher order proof system for mechanizing mathematics. In *EUROCAL85*, Springer-Verlag, Linz, 1985. LNCS 203.

[9] P. Dybjer. *Program Verification in a Logical Theory of Constructions.* Technical Report, Programming Methodology Group, Chalmers University of Technology and University of Göteborg, 1986.

[10] S. Hayashi and H. Nakano. *PX, a Computational Logic.* Technical Report, Research Institute for Mathematical Sciences, Kyoto University, 1987.

[11] W.A. Howard. The formulae-as-types notion of constructions. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism.*, Academic Press, 1980. Unpublished 1969 Manuscript.

[12] J.-L. Krivine and M. Parigot. Programming with proofs. 1987. Preprint, presented at 6th symposium on Computation Theory, Wendish-Rietz, Germany.

[13] C. Mohring. Algorithm development in the calculus of constructions. In *Symposium on Logic in Computer Science*, IEEE Computer Society Press, Cambridge, MA, 1986.

[14] B. Nordström and K. Petersson. Types and specifications. In R.E.A. Mason, editor, *Information Processing 83*, North-Holland, 1983.

[15] P. Martin-Löf. *Intuitionistic Type Theory. Studies in Proof Theory*, Bibliopolis, 1984.

[16] M. Parigot. Programming with proofs : a second order type theory. In *ESOP'88*, Springer-Verlag, Nancy, 1988. LNCS.

[17] A.S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis. LNM 344*, Springer-Verlag, 1973.

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 4.10 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] 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 4.10 implementation.

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 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  (* function application *);;
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(l) -> Abs(lift_rec (k+1) l)
# | App(l,l') -> App(lift_rec k l,lift_rec k l');;
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) (* 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 computationally 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 any more, 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 mechanism 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 CAML'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 `binding_morphism` macro 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 `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-analyzed 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(t1,c1),Abstraction(t2,c2)) -> conv t1 t2
#                                                  & conv c1 c2
#      | (Product(t1,c1),Product(t2,c2)) -> conv t1 t2 & conv t2 c2
#      | (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 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 environment 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, instantiated 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 memorization of the intermediate type shall be needed. We remark that this memorization 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):

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

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

$$\Gamma \vdash N : A \implies \Gamma ; N : A \vdash \qquad \textit{stack_value}$$

$$\Gamma ; N : A \vdash M : C \implies (C \rhd A' \rightarrow B, A \equiv A') \ \ \Gamma \vdash (M\ N) : B \qquad \textit{pop_apply}$$

In rule *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 memorization 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 $\triangleright$ 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

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

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

2.7.5 Type intro

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

```
#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
#        Constdecl(name,Judge(val,_))::rest ->
#                 VAL:=subst_con name val !VAL;
#                 TYP:=subst_con name val !TYP;
#                 ENV:=rest
#      | [] -> error "No constant in environment"
#      | _ -> 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
#      Vardecl(var)::rest -> ENV:=rest; var
#    | [] -> error "No hypothesis in current context"
#    | _  -> 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:

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

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

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

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 $\Gamma \vdash M : Type(j)$ or $\Gamma \vdash M : Prop$. This fact is actually explicit from the way the variable declarations are structured in the machine.

```
#let gen_var () =
#  if !LEV = Proof then error "Proofs 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 $Type(1)$.
- $(M \quad N)$ is an expression when M and N are expressions. It stands for "apply function M to its argument N". This notation is generalized to $(M \quad N1 \quad N2 \ \dots \ Np)$.
- $(x : M)N_x$ 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 accommodate

$(x1, x2, ..., xn : M)N$ 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 $M \rightarrow N$.

- $[x : M]N_x$ 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 accommodate $[x1, x2, ..., xn : M]N$ as an abbreviation for $[x1 : M][x2 : M]...[xn : M]N$.

- $let\ x = M\ in\ N_x$ 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, discharging local variables

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 4.10 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.

A Vernacular Syllabus

Gilles Dowek

INRIA

DRAFT

June 30, 1989

1 Introduction

This document is a manual for the proofs description language implemented in the version 4.10 of the Calculus of Constructions (this language is called the Constructions Vernacular).

The purpose of implementing a proof description language is to make this language as close as possible to the language in which mathematical books are written (we will call this language: the Mathematical Vernacular) in order to make easy user's work. Of course designing such a language is a long endeavor and many problems are still unsolved. What we are presenting here is a preliminary version of the Constructions Vernacular.

A first approach to vernacular is natural deduction. But natural deduction (whatever its name can be) is still quite far from the formalism in which mathematical books are written.

The languages studied by logicians (natural deduction, sequent calculus, etc.) are always idealizations of mathematical vernacular. Logicians never intend to write proofs of non obvious theorems in one of these languages. Their use is different: they are tools to prove meta-theorems (theorems about theorems).

Mathematicians are used to believe that everything which is written in mathematical vernacular could be translated (with enough time and paper) in natural deduction for instance, but when we want to write a proof description language this belief does not help since it gives no algorithm.

Nevertheless, natural deduction seems to be a good kernel for vernacular. The main problem with it is that the elementary operations are too simple. The vernacular of the Calculus of Constructions can be seen as a higher level language for writing natural deduction proofs.

In mathematical books proofs are not always fully written, in many theorems authors only gives indications and leave the reader complete the proof. This possibility exists in the Calculus of Constructions, so we have to distinguish two kinds of statements in vernacular: some statements are high level abbreviations for writing proofs, others are indications given to the theorem prover in order to guide its non deterministic search. We are presenting here the first kind of statements, the others are described in the user's guide for the Tactic's Theorem Prover by Thierry Coquand.

The Constructions Vernacular is a rich language. We are going to present it in several steps : in the first one (sections 2 and 3) we are going to see how to write first order logic with the symbols $\rightarrow$ and $\forall$ and we will be able to write a non obvious example. Then we will see other features of the language : in section 4 we are going to extend the language with all the usual logical symbols, in section 5 we are going to extend first order logic to higher order logic. Then in section 6 to 9 we are going to see some features which make the proofs easier to be written.

2 Classification of symbols and sentences

In this section we will have to define notions such as proposition, axiom, theorem, etc. These definitions should not be expected to be interesting from a logical or mathematical point of view, their only object is to explain what these notions are *in the implementation of the vernacular* and what is allowed and what is not in this language.

2.1 Symbols

In vernacular we distinguish two kinds of symbols: the term building symbols and the proposition building symbols.

2.1.1 Universal algebra

Trees The first order term building symbols are: individual symbols, function symbols and variables. With these symbols we can build first order terms.

These terms are trees in which non-terminal-nodes are function symbols and leaves individual symbols and variables. For instance in arithmetics, we have the symbols 0, S, $+$ and $*$. The trees 0, $(S\ 0)$, $(*\ (S\ (S\ 0))\ (S\ x))$ are first order terms.

Types Usually, a model of a theory is a set and individual symbols and variables denote elements of this set. In the Calculus of Constructions it is possible to define individual symbols and variables belonging to different sets. So we associate to each individual symbol and variable a type which indicates to which set the element belongs. For instance the type of natural numbers is Nat and 0 is of type Nat. We write this $0 : Nat$. If the variable x denotes an integer we write $x : Nat$.

Usually, function symbols are characterized by their arities, for instance the arity of S is 1 and the arity of $+$ and $*$ is 2. Since individual symbols and variable are typed we have also to type function symbols in order to avoid building senseless terms, for instance we have to make sure that S is always applied to terms of type Nat. So that we give a type to S, $+$ and $*$, $S : Nat \rightarrow Nat$, $+ : Nat \rightarrow Nat \rightarrow Nat$, $* : Nat \rightarrow Nat \rightarrow Nat$. The types of function symbols are build with atomic types and the arrow and are called functional types.

Thus, first order terms are well typed trees.

Types as terms The types are trees where non-terminal-nodes are always the arrow and leaves are atomic types. We can consider these trees as terms. In order to distinguish types from the other terms we give them the type $Type$. So $Type$ is the type of all the types, the arrow is a function symbol of type $Type \rightarrow Type \rightarrow Type$ and atomic types are individual symbols of type $Type$.

2.1.2 Propositions

The first order propositions building symbols are: predicates, connectors, quantifiers.

Predicates The basic way to form a proposition is to apply a predicate to a sequence of terms. For instance, if P is a predicate over one element of type T, and t of term of type T. Then $(P\ t)$ is a proposition. Propositions built that way are trees with one non-terminal-symbol which is a predicate and leaves which are terms.

Propositions as terms In the Calculus of Constructions, these trees can be considered as terms. In order to distinguish propositions from other terms we give them the type $Prop$. So a predicate of n places of type $t_1 ... t_n$ is a object of type $t_1 \rightarrow ... \rightarrow t_n \rightarrow Prop$.

Quantifiers When we have a proposition P where a free variable x of type T may occur, it is possible to quantify P over x. In mathematical vernacular we write this $\forall x : T \quad P$. In the vernacular of the Calculus of Constructions, we write this $(x : T)P$. The variable x may or may be not free in P, if it is, it is not free any more in $(x : T)P$.

So we extend the propositions structure by adding universal quantification. The rule for typing is

if $x : T$ allow to type $P : Prop$, then $(x : T)P : Prop$.

Existential quantification will be studied in the section: Logical symbols.

Connectors The last way to form propositions is to use connectors, as $\wedge, \vee, \rightarrow, \leftrightarrow$ and $\neg$. Let us look for instance at the arrow: $\rightarrow$. If have two propositions P and Q we can form the proposition $P \rightarrow Q$. So the arrow appears like an object of type $Prop \rightarrow Prop \rightarrow Prop$. This arrow (implication) must not be confused with the arrow used to form types, which is of type $Type \rightarrow Type \rightarrow Type$.

Other connectors will be studied in section: Logical symbols.

2.2 Sentences

2.2.1 Classification

In vernacular there are four kinds of sentences.

- Axioms which are assumed propositions,

- Theorems which are pairs of propositions and proofs,

- Constant definitions which are introducing a new symbol for a term (a constant). For instance: "let 1 be (S 0)".

- Declarations which are introducing a new variable standing for every term of its type. For instance: "let n be an integer".

Each of these sentences defines a new name which may be used in the following:

- an axiom associates to this a name, a proposition,
- a theorem associates to this name a proof and a proposition,
- a defined constant associates to this name a term and a type,
- a declared variable associates to this name a type.

A fifth kind of words can be used: they are key-words such as $Prop$ and $Type$, and the arrow $\rightarrow$: $Type \rightarrow Type \rightarrow Type$.

2.2.2 Terms building symbols and propositions building symbols

In vernacular each word is of one of these five kinds. So terms building symbols and propositions building symbols are also of one of these kinds.

Free variable in terms are of course variables (i.e. symbols introduced by a variable declaration). The only connector introduced till here is the arrow, and it is a key-word. The only quantifier is the forall, we can consider it as a key-word. (Actually, since we do not explicitly write this quantifier it is more or less a key-word internalized in the term structure.)

So remains individual symbols, functions symbol, predicates and atomic types. For instance in arithmetics they are 0, S, $+$, $*$, $=$, $<$ and Nat. They cannot be key-word since they belong to one particular theory. They are not axioms or theorems of course. So they can be defined constants or declared variables.

In axiomatic theories they are implicitly considered as variables.

It may seem strange to declare 0 as a variable, since a variable represents any element of its type. And 0 is not just any natural number.

The difference between 0 and an arbitrary x is that if we want to define entirely arithmetics we will have to assume axioms which involve 0 and no axiom which involve x. So 0 is a variable standing for every natural number which verifies the axioms. And what we will prove for 0 could be proved for any natural number that satisfies Peano's axioms.

So, 0, S, $+$, $*$, $=$, $<$ and Nat are variables.

$Nat : Type$

$0 : Nat$

$S : Nat \rightarrow Nat$

$+ : Nat \rightarrow Nat \rightarrow Nat$

$* : Nat \rightarrow Nat \rightarrow Nat$

$=: Nat \rightarrow Nat \rightarrow Prop$

$<: Nat \rightarrow Nat \rightarrow Prop$

The same thing stands for instance for theory of ordered sets. We consider a type A of the elements of the ordered set, and two predicates: R which is the order

relation and Eq which is the equality. A, R and Eq are variables.

Another way to consider individual and function symbols and predicates is to define them as constants, i.e. to find a model of the theory in the lambda-calculus (for instance Church's integer's for Peano's axioms). Doing so we do not work on any model of the theory, but on the chosen model, so we lose generality, except if we prove that every model of the theory is isomorphic to the model chosen in lambda-calculus.

In fact in many cases, as in arithmetics, this loss of generality is not regretable since we are only intersested in the standard model.

2.2.3 Context

The *context of a sentense* is the set of all the sentenses written before that sentense. This context defines the symbols that are allowed to be used in the sentense.

For instance in the empty context, it is possible to define a propositional variable P (i.e. a variable of name P and of type *Prop*). Then in the context formed with that declaration, it is possible to assume an axiom u of statement P, etc.

In the empty context the assumption of an axiom of statement P does not make any sense since P is an undefined symbol.

2.2.4 Typing terms and proving theorems

In a context Γ we can form a term t of type T if all the free variables of t are declared in Γ and if all the types are consistent.

A proposition P can be formed if all its free variables are declared in the context Γ and if types are consistent.

A well formed proposition can be proved using the axioms of Γ and the rules of natural deduction.

Since we restricted the set of connectors and quantifiers we can restrict to five the number of rules of natural deduction used to prove theorems:

- Rule 1. If there is in the current context an axiom of statement P then it is possible to write a theorem of statement P. (Axiom link)
- Rule 2. If there is in the current context an axiom or a theorem of statement $P \to Q$ and an axiom or a theorem of statement P then we can deduce Q [1]. ($\to$-elim or Modus Ponens)
- Rule 3. If it is possible to prove Q [2] in the current context grown with the an axiom of statement P then we can deduce $P \to Q$ in the current context. ($\to$-intro)

[1] i.e. write a theorem of statement Q

[2] i.e. if it is possible to write a theorem of statement Q

- Rule 4. If there is in the current context an axiom or a theorem of statement $\forall x : T \quad P$ and if t is a term of type T in the current context then we can deduce $P[x \leftarrow t]$. ($\forall$-elim)
- Rule 5. If it is possible to prove P in the current context grown with a declaration of a variable x of type T then we can deduce $\forall x : T \quad P$ in the current context. ($\forall$-intro)

3 Fundamental vernacular

3.1 Declarations and axioms

In vernacular the syntax to declare a variable, is the following:

```
Parameter <name of the axiom>.
Inhabits <statement of the axiom>.
```

For instance we can declare two variables e and f of type A. The only thing we have to do before is to declare A as type variable (i.e. a variable of type Type).

```
(* A is a Type *)
Parameter A.
Inhabits Type.

(* e is of type A *)
Parameter e.
Inhabits A.

(* f is of type A *)
Parameter f.
Inhabits A.
```

We can also declare R and Eq as variables.

```
Parameter R.
Inhabits A->A->Prop.

Parameter Eq.
Inhabits A->A->Prop.
```

In arithmetics we can define the type Nat, 0, S, $+$, and Eq for instance.

```
Parameter Nat.
Inhabits Type.

Parameter Zero.
Inhabits Nat.

Parameter S.
Inhabits Nat -> Nat.

Parameter Plus.
Inhabits Nat -> Nat -> Nat.

Parameter NatEq.
Inhabits Nat -> Nat -> Prop.
```

The syntax for adding a new axiom in the context is quite similar.

`Axiom <name of the axiom>.`

`Assumes <statement of the axiom>.`

For instance we can assume the axiom $P \to Q$. The only thing we have to do before is to tell that P and Q are propositional variables (i.e. variables of type Prop).

```
Parameter P.
Inhabits Prop.

Parameter Q.
Inhabits Prop.

Axiom u.
Assumes P->Q.
```

We can also assume P as an axiom.

```
Axiom v.
Assumes P.
```

We can also assume an axiom with a quantifier $\forall$, for instance: we can assume that our relation R is assymmetrical and transitive :

```
Axiom Assym.
Assumes (x:A)(y:A)((R x y) -> (R y x) -> (Eq x y)).

Axiom Trans.
Assumes (x:A)(y:A)(z:A)((R x y) -> (R y z) -> (R x z)).
```

We can also assume that $(R\ e\ f)$ and $(R\ f\ e)$ for instance.

```
Axiom E1.
Assumes (R e f).

Axiom E2.
Assumes (R f e).
```

3.2 Theorem proving

3.2.1 Using the rule $\rightarrow$-elim

If there is a axiom or a theorem of name u which statement is $P \rightarrow Q$ and an other axiom of theorem of name v which statement is P.

Then the rule $\rightarrow$-elim says that we can deduce Q, and in vernacular the way to write that the proof of Q is the application of rule $\rightarrow$-elim to u and v is simply to write that the proof of Q is: $(u\ v)$.

The syntax for proving a new theorem is:

```
Theorem <name of the theorem>.
Statement <statement of the theorem>.
Proof <proof of the theorem>.
```

for instance:

```
Theorem w.
Statement Q.
Proof (u v).
```

3.2.2 Using the rule $\forall$-elim

The same thing stands for the rule $\forall$-elim. If u is the name of a theorem or an axiom whose statement is $\forall x : T \quad P$ and t a term of type T then $(u\ t)$ is the way we write a proof of $P[x \leftarrow t]$.

For instance we can prove the theorem $(R\ e\ f) \rightarrow (R\ f\ e) \rightarrow (Eq\ e\ f)$.

```
Theorem As.
Statement ((R e f) -> (R f e) -> (Eq e f)).
Proof (Assym e f).

(* (Assym e) is a proof of (y:A)((R e y) -> (R y e) -> (Eq e y))
   (Assym e f) is a proof of (R e f) -> (R f e) -> (Eq e f) *)
```

We can then prove $(Eq\ e\ f)$ with the rule $\rightarrow$-elim

```
Theorem E.
Statement (Eq e f).
Proof (As E1 E2).
```

3.2.3 Using the rule: Axiom link

If there is in the context an axiom of name u and statement P then we can write a theorem of statement P a proof for this theorem is simply u.

For instance:

```
Theorem Assym_as_theorem.
Statement (x:A)(y:A)((R x y) -> (R y x) -> (Eq x y)).
Proof Assym.
```

3.2.4 Using the rule $\rightarrow$-intro

What we have to do is to use this rule is to:

1. grow the context with the proposition P (i.e. declare a new axiom $x : P$),

2. prove Q in that context (adding a new theorem of name w (for instance), and statement Q),

3. eliminate the proposition P of the context while modifying w in order that its statement becomes $P \rightarrow Q$.

In mathematical vernacular the two first operations are quite explicit: "to prove $P \rightarrow Q$, let us assume P then prove Q". The third is always implicit: a mathematician does not distinguish proving Q under the hypothesis P and proving $P \rightarrow Q$. But what is clear in a mathematical book is when an hypothesis is valid or not. When the proof of Q under the hypothesis P is over it is implicit that P is not assumed any more. The scope of the hypothesis P is indicated by the modularity of the mathematical text. (On that point mathematical vernacular looks like programming languages.)

So we have in vernacular the possibility to declare local axioms (hypothesis).

An hypothesis is said to be local to a proof of a theorem if its scope is limited to this proof. When the proof is over, all the local hypotheses are erased and appear at the left of an arrow in the theorem.

For instance let us prove the theorem $B \to B$ in any context Γ, where B is a proposition.

```
Parameter B.
Inhabits Prop.

Theorem I.

Statement B->B.

Hypothesis x.
Assumes B.

Proof x.
```

When the proof is finished x in not a variable declared in the context any more.

Let us note that the proof given for the theorem is x i.e. a proof of B, and that the statement of the theorem is written $B \to B$. So assumption of the hypothesis B is local to the proof but not to the statement. It is also possible to declare a hypothesis local to the whole theorem:

```
Theorem I'.

Hypothesis x.
Assumes B.

Statement B.

Proof x.
```

Here the hypothesis is assumed before the statement of the theorem. So the statement is B (and not $B \to B$) and it will be modified when the hypothesis x will be erased in order to become $B \to B$.

3.2.5 Using the rule ∀-intro

The use of this rule is exactly the same as the rule number →-intro, the only difference is that the local declaration is a variable declaration and not an hypothesis. The keyword used is *Variable*.

For instance if we want to quantify the previous proposition over the propositional variable, and prove $(C : Prop)(C \to C)$ (in order to prove further $D \to D$ for a certain D).

```
Theorem I.

Statement (C:Prop)(C->C).

Variable C.
Inhabits Prop.

Hypothesis x.
Assumes C.

Proof x.
```

3.3 Defining constants

The syntax to define constants is:

```
Definition <name of the constant>.
Body <term>.
```

For instance:

```
Definition One.
Body (S Zero).
```

We can also define constants of type Prop.

```
Definition EqZero.
Body (NatEq Zero Zero).
```

This constant definition must not be confused with the theorem $(NatEq\ Zero\ Zero)$, it is also possible to define a constant whose value is $(NatEq\ Zero\ One)$. A constant like $EqZero$ can not be used in a proof of a theorem as a proof of $(NatEq\ Zero\ Zero)$. But, it can be used as an abbreviation for the formula $(NatEq\ Zero\ Zero)$ in the statement of a theorem for instance.

3.4 Local lemmas and local constants

So far we have seen the possibility of declaring local hypotheses and local variables, but when a theorem is proved or when a constant is defined, the theorem or definition remains forever. We are going to see in this section the possibility to prove local theorems, and to declare local definitions. First, this possibility will make the proofs more readable by pointing out the major theorems and definitions in the text. The reader will understand that Pythagore's theorem is more important that one of its technical lemmas. Then the machine also will remark the same thing and will not keep the technical lemma in the same place in its memory as Pythagore's theorem and then the proof checking and program extraction will become more efficient.

The syntax is the following: local lemmas and definitions are written with the local hypothesis and variables, i.e. between the first and the last phrase of the sentence they belong.

The keyword for local lemmas is Remark.

```
Remark <name of the remark>.
Statement <statement of the remark>.
Proof <proof of the remark>.
```

For local definitions the keyword is Local.

```
Local <name of the constant>.
Body <term>.
```

3.5 An example: Tarski's theorem

We are now able to write a non obvious theorem in vernacular. We will show that in a complete partial ordered set, every increasing function has a fixpoint.

Let us consider the set:

$\{x|x \prec f(x)\}$

this set has an least upperbound M (The partial order is complete) we show that this least upperbound is a fixpoint.

Since we do not have yet the existential quantification and the completeness is a second order property, we will only show here the proposition:

if $\prec$ is a partial order over A and f an increasing function then:

$\forall M(M = lub\{x|x \prec f(x)\} \to M = f(M))$

$M = lub\{x|x \prec f(x)\}$ can be expressed in a first order way:

$(\forall x(x \prec f(x) \to x \prec M)) \wedge (\forall y((\forall x(x \prec f(x) \to x \prec y)) \to M \prec y))$

Actually since we do not have yet the connector $\wedge$ we shall show the curryfied form of the proposition:

$(\forall x(x \prec f(x) \to x \prec M)) \to (\forall y((\forall x(x \prec f(x) \to x \prec y)) \to M \prec y))$

We assume a function f and the axiom which assumes that f is increasing:

```
Parameter f.
Inhabits A->A.
```

```
Axiom Incr.
Assumes (x:A)(y:A)((R x y) -> (R (f x) (f y))).
```

We now give the statement of the theorem:

```
Theorem Tarski1.

Statement (M:A) (((x:A) ((R x (f x)) -> (R x M))) ->
                ((x:A)(((y:A)(R y (f y))->(R y x))->(R M x))) ->
                (Eq M (f M))).
```

The theorem begins by a quantifier ∀ and two arrows, we want to use the intro rules, we declare a local variable and two hypotheses:

```
Variable M.
Inhabits A.

Hypothesis Up.
Assumes (x:A) ((R x (f x)) -> (R x M)).

Hypothesis Least.
Assumes (x:A)(((y:A)(R y (f y))->(R y x))->(R M x)).
```

Then we show a few lemmas:

```
Remark One.

Statement (y:A)(R y (f y))->(R y (f M)).

   Variable y.
   Inhabits A.

   Hypothesis v.
   Assumes (R y (f y)).

   Remark T.
   Statement (R y M).
```

```
    Proof (Up y v).

(*    Up is a proof of (x:A) ((R x (f x)) -> (R x M))
      (Up y) is a proof of ((R y (f y)) -> (R y M))
      v is a proof of (R y (f y))
      (Up y v) is a proof of (R y M)  *)

    Remark T'.
    Statement (R (f y) (f M)).
    Proof (Incr y M T).

  Proof (Trans y (f y) (f M) v T').

  Remark Two.
  Statement (R M (f M)).
  Proof (Least (f M) One).

  Remark Three.
  Statement (R (f M) (f (f M))).
  Proof (Incr M (f M) Two).

  Remark Four.
  Statement (R (f M) M).
  Proof (Up (f M) Three).
```

Then we conclude the theorem:

```
Proof (Assym M (f M) Two Four).
```

4 Logical symbols

In the previous section we have seen how to write mathematics using the only connector $\rightarrow$ and the only quantifier $\forall$. In this section we are going to study the others connectors and quantifiers: $\wedge$, $\vee$, $\neg$ and $\exists$.

4.1 Writing propositions

The symbol $\wedge$ (and) is written in vernacular /\. For instance P and Q are two propositions, then $P \wedge Q$ is the proposition "P and Q".

The symbol $\vee$ (or) is written in vernacular \/. For instance P and Q are two propositions then $P \vee Q$ is the proposition "P or Q".

The negation is written ~ in vernacular, if P is a proposition then ~P is the proposition "not P".

The symbol of predicate of arity zero which is always false is written {}. {} is the proposition "Absurdity".

The to use the existential quantification we need a formula P where x may be free and that can be typed as a proposition in the context where x is declared of type T. So that it is possible to form the proposition $\exists x : T \;\; P$. In vernacular this proposition is written `<T> Ex ([x:T]P)`.

All the terms written with these symbols are of type *Prop*.

4.2 Proving theorems

In natural deduction we have rules concerning each symbol. With these rules we can prove propositions. If a proposition P can be proved in natural deduction it can also be proved in the Calculus of Constructions: we can define a theorem of statement P and of proof π. The form of the proof π depend on the rule used. We are going to see in this section how to write in vernacular the proof π of a theorem, from it's proof in natural deduction.

4.2.1 Conjunction

- Rule 6. If there is in the current context an axiom or a theorem v of statement P, and an axiom or a theorem w of statement Q, then we can deduce a theorem of statement $(P \wedge Q)$.

The proof of $(P \wedge Q)$ is written $(conj\ P\ Q\ v\ w)$. For instance:

```
Theorem z.

Statement (P/\Q).

Proof (conj P Q v w).
```

- Rule 7. If there is in the current context an axiom or a theorem z of statement $(P \wedge Q)$ then we can deduce a theorem of statement P ($\wedge$-elim1).

The proof of P is written $(proj1\ P\ Q\ z)$.

- Rule 8. If there is in the current context an axiom or a theorem z of statement $(P \wedge Q)$ then we can deduce a theorem of statement Q ($\wedge$-elim2).

The proof of Q is written $(proj2\ P\ Q\ z)$. For instance:

```
Theorem p.

Statement P.

Proof (proj1 P Q z).

Theorem q.

Statement Q.

Proof (proj2 P Q z).
```

4.2.2 Disjunction

- Rule 9. If there is in the current context an axiom or a theorem u of statement P then we can deduce a theorem of statement $P \vee Q$ ($\vee$-introl).

The proof of $P \vee Q$ is written (*or_introl* P Q u).

- Rule 10. If there is in the current context an axiom or a theorem v of statement Q then we can deduce a theorem of statement $P \vee Q$ ($\vee$-intro2).

The proof of $P \vee Q$ is written (*or_intror* P Q v).

- Rule 11. If there is in the current context an axiom or a theorem u of statement $P \vee Q$, an axiom or a theorem v of statement $(P \rightarrow K)$ and an axiom or a theorem w of statement $(Q \rightarrow K)$ then we can deduce a theorem of statement K. ($\vee$-elim).

The proof of K is written $(u\ K\ v\ w)$. For instance :

```
Variable K.
Inhabits Prop.

Axiom u.
Assumes (P\/Q).

Axiom v.
Assumes (P->K).

Axiom w.
Assumes (Q->K).
```

```
Theorem O.
Statement K.
Proof (u K v w).
```

4.2.3 Negation and absurdity

Three rules concern these symbols in constructive mathematics.

- Rule 12. If there is in the current context an axiom or a theorem u of statement $\{\}$ then wecan deduce A ($\{\}$-elim).

  (This means that every proposition can be proved in an inconsistent theory.) The proof of A is written $(u\ A)$.

- Rule 2'. If there is in the current context an axiom or a theorem u of statement A and an axiom or a theorem v of statement $\neg A$ then we can deduce $\{\}$.

- Rule 3'. If it is possible to prove $\{\}$ in the current context grown with the local hypothesis A the we can deduce $\neg A$.

Actually if we consider $\neg A$ as an abbreviation for $A \rightarrow \{\}$ then the rules 2' and 3' becomes particular cases of the rules 2 and 3 ($\rightarrow$-elim and $\rightarrow$-intro). So the way we write proof constructed with these rules are just particular cases of the way used for $\rightarrow$.

4.2.4 Existential quantification

- Rule 13. If t of type T such that $P[x \leftarrow t]$ is an axiom or a theorem u, then we can deduce $\exists x : T\ \ P$ ($\exists$-intro).

The proof of $\exists x : T\ \ P$ is written $(ex_intro\ T\ ([x : T]P)\ t\ u)$. For instance:

```
Variable T.
Inhabits Type.

Variable t.
Inhabits T.

Variable Q.
Inhabits T->Prop.

Axiom u.
Assumes (Q t).
```

```
Theorem E.
Statement (<T> Ex ([x:T](Q x))).
Proof (ex_intro T ([x:T](Q x)) t u).
```

- Rule 14. If there is in the current context an axiom or a theorem u of statement $\exists x : T\ \ P$, and an axiom or a theorem v of statement $\forall x : T\ \ (P \to K)$ and if x is not free in K then we can deduce K.

The proof of K is written $(u\ K\ v)$.

```
Variable K.
Inhabits Prop.

Axiom v.
Assumes (x:T)(Q x)->K.

Theorem E'.
Statement K.
Proof (E K v).
```

5 Writing higher order theories in vernacular

5.1 Typed lambda calculus

In a theory we want to be able to express not only propositions concerning the elements of the model, but also propositions concerning functions over this set and predicates over this set (i.e. subsets). For instance in Tarski's theorem we want to write "all subsets of A have a least upperbound". In first order theory we can not quantify on a subset, we had to consider completeness as an axiom schema, and assume the instantiation of this axiom for the subsets we were interested in.

In first order theories terms denote elements of the model. They are typed trees. In higher order theories terms must denote not only elements of the model, but also functions over this set and predicates. Higher order terms are defined by extending typed trees structure to typed lambda calculus.

We use a syntax of explicit typed lambda-calculus, where variables are typed in their binders, and we write $[x : A]t$ what is written $\lambda x_A.t$ in Church's notation.

For instance in arithmetic, the duplicator is denoted by $[x : Nat](*\ (S\ (S\ 0))\ x)$.

A term can be typed in a context where all its free variable are typed, for instance $(*\ (S\ (S\ 0))\ (S\ x))$ can be typed in a context where $x : Nat$. Its type is Nat, since if x denotes a natural number $(*\ (S\ (S\ 0))\ (S\ x))$ denotes also a natural number. $[x : Nat](*\ (S\ (S\ 0))\ (S\ x))$ can be typed in the empty context, since it

has no free variable. Its type is $Nat \rightarrow Nat$ since this term denotes the function which associate to every natural number an other natural number.

As in first order theories types are terms of type *Type*.

In fact the lambda-calculus used in the Calculus in Constructions is a more powerful calculus, where the arrow is generalized to type-dependent products, but the subset of lambda-terms presented here seems sufficient to write ordinary mathematical terms.

5.2 A higher order example

We are now able to express and prove Tarski's theorem.

We can define a constant predicate *Lub* (Least Uper bound). This predicates has two arguments the first is an elements x of A and the second a predicate S over A, $(Lub\ x\ S)$ is the proposition "x is the least upper bound of the elements of A which verify S".

```
Definition Lub.
Body [m:A][S:A->Prop](and  ((x:A) ((S x) -> (R x m)))
((y:A)  ((((x:A) ((S x)->(R x y))) -> (R m y)))).
```

Then we can express the completeness axiom: for any predicate S over A, there exists an element x of A which is the least upper bound of the element s which verifies S.

```
Axiom Complete.
Assumes (S:A->Prop)(<A> Ex ([x:A] (Lub x S))).
```

We define a set (predicate) of the elements of A which are under their mapping.

```
Definition Under.
Body [x:A](R x (f x)).
```

Then we can write the theorem Tarski1 in a nicer form:

```
Theorem Tarski1'.

Statement ((M:A) ((Lub M Under) ->  (Eq M (f M)))).
```

Then we declare the variable M and assume the hypothesis *LeastUp* which assumes (*Lub M Under*).

```
Variable M:A.

Hypothesis LeastUp.
Assumes (Lub M Under).
```

and we show from this hypothesis the two hypotheses of the Theorem Tarski1 by using the and-elim rules.

```
Remark Up.
Statement (x:A) ((R x (f x)) -> (R x M)).
Proof (proj1 ((x:A) ((R x (f x)) -> (R x M)))
             ((x:A)(((y:A)(R y (f y))->(R y x))->(R M x)))
             LeastUp).

Remark Least.
Statement (x:A)(((y:A)(R y (f y))->(R y x))->(R M x)).
Proof (proj2 ((x:A) ((R x (f x)) -> (R x M)))
             ((x:A)(((y:A)(R y (f y))->(R y x))->(R M x)))
             LeastUp).
```

Then we prove Tarski1' using Tarski1

```
Proof (Tarski1 M Up Least).
```

Then we show the following lemma: for all M, if M is the least upper bound of *Under* then exists an m in A which is a fixpoint of f.

```
Theorem Tarski2.
Statement ((M:A) ((Lub M Under) -> <A> Ex  ([m:A] (Eq m (f m))))).
```

we declare an M and assumes that it is the least upperbound of *Under*.

```
Variable M.
Inhabits A.
```

```
Hypothesis h.
Assumes (Lub M Under).
```

Then we prove the existence by exhibiting a witness ($\exists$-intro): M is such a witness, as proved by lemma Tarski1'.

```
Proof (ex_intro A ([m:A] (Eq m (f m))) M (Tarski1' M h)).
```

Then we show that *Under* has a least upper bound, by particularization of the axiom of completeness.

```
Theorem Exist_lub_under.
Statement <A> Ex ([m:A] (Lub m Under)).
Proof (Complete Under).
```

Then we prove the theorem by particularizing the lemma Tarski2 to the least upper bound of *Under*. (Here we do not have any witness of this existence since this existence has been proved from an axiom (Completeness) which is not a Harrop formula. We use an $\exists$-elim rule over the existential quantifier of *Exist_lub_under*.)

```
Theorem Tarski.
Statement <A> Ex ([x:A](Eq x (f x))).
Proof (Exist_lub_under (<A> Ex ([x:A](Eq x (f x)))  Tarski2).
```

6 Ergonomy

6.1 Abbreviations

6.1.1 One phrase sentences

All the sentences we have seen were written in several phrases: three for the theorems, two for the variable declarations, axioms, and constants definitions. This separation is useful in some cases, for instance when we want to assume a hypothesis local to a theorem, we have to write this hypothesis between these phrases, but when it is not useful a one-phrase syntax is nicer:

- `Axiom <name>:<statement>.`

  is an abbreviation for:

Axiom <name>.

Assumes <statement>.

- Parameter <name>:<type>.

  is an abbreviation for:

  Parameter <name>.

  Inhabits <type>.

- Theorem <name> <statement> Proof <proof>.

  is an abbreviation for:

  Theorem <name>.

  Statement <statement>.

  Proof <proof>.

- Definition <name> = <term>.

  is an abbreviation for:

  Definition <name>.

  Body <term>.

- Hypothesis <name>:<statement>.

  is an abbreviation for:

  Hypothesis <name>.

  Assumes <statement>.

- Variable <name>:<type>.

  is an abbreviation for:

  Variable <name>.

  Inhabits <type>.

- Remark <name> <statement> Proof <proof>.

  is an abbreviation for:

  Remark <name>.

  Statement <statement>.

  Proof <proof>.

- Local <name> = <term>.

  is an abbreviation for:

  Local <name>.

  Body <term>.

6.1.2 Defining typed constants

In a constant definition we associate a name to a term. The type of this term is infered by the system. It is also possible to give the term and its type, and the system will check that the type is corect. The syntax is :

```
Definition <name>.
Body <term>:<type>.
```

The abbreviated forms are :

```
Definition <name> = <term>:<type>.
```

and:

```
Definition <name>:<type> = <term>.
```

6.1.3 Declaring several variables

To declare several variables with the same type,

- `Variables <name 1>,<name 2>,...,<name n>:<type>.`

  is an abbreviation for:

  ```
  Variable <name 1>:<type>.
  Variable <name 2>:<type>.
  ...
  Variable <name n>:<type>.
  ```

6.1.4 Propositions and Types

There are also a few abbreviation which clarify proofs:

- `Proposition <name>.`

  is an abbreviation for

  `Variable <name>:Prop.`

- `Propositions <name 1>,<name 2>,...,<name n>.`

  is an abbreviation for

  `Variables <name 1>,<name 2>,...,<name n>:Prop.`

- `Type <name>.`

  is an abbreviation for

  `Variable <name>:Type.`

- `Types <name 1>,<name 2>,...,<name n>.`

  is an abbreviation for

  `Variables <name 1>,<name 2>,...,<name n>:Type.`

6.2 Local hypotheses and variables in definitions, axioms and declarations

We have seen in the previous sections the possibility to declare hypotheses, variables, lemmas and definitions local to proofs. This possibility was crucial since it was the incarnation in vernacular of two natural deduction rules. The possibility to enlarge this local declaration to the whole theorem (statement included) was a facility offered to write shorter statements. This facility is also offered in axioms, variable declaration and definitions.

When a local hypothesis P is assumed in an axiom of statement Q, the assumed proposition is $P \rightarrow Q$.

When a local variable $x : T$ is assumed in an axiom of statement Q, the assumed proposition is $(x : T)Q$.

When a local variable $x : T$ is assumed in a definition of body t. Then the term abbreviated is then $[x : T]t$.

Local variable in variables could allow to higher order variables in a nicer fashion.

In definitions and declarations, local hypotheses allow definition and declaration of proof-dependent objects as the square root. (In order to form $(sqrt\ x)$ we have to give the number x, and also a proof that x is greater than zero).

6.3 Factorized local sentences

When we write a book we are used to proves different theorems which share hypothesis. For instance in a group's theory book we may want to write all the theorems concerning commutative groups in one section.

All these theorems are in the form:

Statement $((Commutative\ \ G) \rightarrow P)$.

(G is a parameter, i.e. a global variable).

To prove such a theorem we locally assume $(Commutative\ \ G)$ and prove P.

The use is not to repeat the hypothesis in each statement, and it's assumption in each proof, but to write at the beginning of the section: "In this section we assume that G is commutative". Then we write the theorem

Statement P

we do not assume locally that G is commutative, and if we need this proposition we refer to the section assumption.

Inside the section, the theorem is considered as P, and after the chapter is considered as $((Commutative\ \ G) \rightarrow P)$.

So the hypothesis $(Commutative\ \ G)$ is not an axiom nor an hypothesis local to a theorem: it is an hypothesis local to a section.

In order to allow such section-local-hypothesis we have to allow a section delimiting system:

The beginning of a section is written:

`Section <name of the section>.`

The end is written:

```
End <name of the section>.
```

For instance :

```
Section Commutative_groups.
...
End Commutative_groups.
```

In such a section sentences beginning by *Hypothesis*, *Variable*, *Remark* and *Local* define hypothesis, variable, lemma and constant local to the section. Out of the section, they are discarded and all the items defined in the section are modified as if variables and hypothesis were local to this item. For instance if the item is a theorem of statement P, if H is an hypothesis then the statement of the theorem becomes $H \rightarrow P$, and if x is a variable of type T. The proposition P becomes $(x:T)P$.

7 Inductive definitions

The possibility of defining recursive types in the Constructions Vernacular is described in Inductive Definitions in the Calculus of Constructions by Christine Paulin-Mohring.

8 Lambda-terms as proofs

In the previous section we have seen how to write proofs of theorems. The tricks given for each natural deduction rules could look a bit mysterious. We are going now to understand why the proofs were written that way. Understanding the proofs structure will allow us to write more proofs. We won't be able to prove new theorems, but we will be able to write shorter (but also darker) proofs.

8.1 Heyting's semantics

Let us consider first the propositions only build with propositional variables and the arrow: $\rightarrow$.

In order to prove theorems we have two natural deduction rules: $\rightarrow$-elim, $\rightarrow$-intro.

For instance:

- In a context Γ where A is a proposition it is possible to prove $A \rightarrow A$.

  Γ, A [3] proves A (Axiom link) then we can deduce that Γ proves $A \rightarrow A$ ($\rightarrow$-intro).

[3] We write Γ, A the context Γ grown with an axiom of statement A.

- In a context Γ where A and B are propositions and where there is an axiom of statement B, we can prove $A \rightarrow B$.

  Γ, A proves B (Axiom link), then we can deduce that the context Γ proves $A \rightarrow B$ ($\rightarrow$-intro).

- In a context Γ where A and B are propositions, and there is an axiom of statement $(A \rightarrow A) \rightarrow B$ we can prove B.

  Γ, A proves A (Axiom link), then we can deduce that Γ proves $A \rightarrow A$ ($\rightarrow$-intro).

  In the context *Gamma* $(A \rightarrow A) \rightarrow B$ is an axiom and $A \rightarrow A$ a theorem then Γ proves B ($\rightarrow$-elim).

Heyting's semantics proposes to associate to each propositional variable a set of justifications (the set of its proofs) and to define recursively the proof of $P \rightarrow Q$ as a function which maps every proof of P to a proof of Q.

These functions are of course denoted by lambda terms.

For each axiom we define a variable which is by its definition a proof of this axiom.

A proposition is said to be provable in a context Γ if we can exhibit a proof, i.e. a lambda-term, in which all the free variable are proofs of the axioms of Γ.

For instance $A \rightarrow A$ has a proof in a context with no axioms (a closed term) which is $[x]x$ since $[x]x$ is a function which maps every proof of A with a proof of A.

Let b be a variable which is by definition a proof of B. $[x]b$ is a proof of $A \rightarrow B$ since $[x]b$ maps every proof of A with a proof of B.

If f is a proof of $(A \rightarrow A) \rightarrow B$ then $(f([x]x))$ is a proof of B since $[x]x$ is a proof of $A \rightarrow A$ and f associates to each proof of $A \rightarrow A$ a proof of B.

A proof of a proposition P written in natural deduction can easily translated into a lambda-term. If this proposition is proved in a context Γ, then its proof (i.e. lambda-term) will have free variables which are proofs of the axioms of Γ.

We will show this theorem by induction over the length of the proof :

If the last rule used is the *Axiom link* then P is an axiom and the variable p introduced for P is a proof of P.

If the last rule used is $\rightarrow$-elim then we have a proof f of $Q \rightarrow P$ and a proof u of Q, $(f\ u)$ is a proof of P since f maps every proof of Q with a proof of P.

If the last rule used is $\rightarrow$-intro then P is in the form $Q \rightarrow R$. Let x be a variable proof of Q. We have a natural deduction proof of R in the context Γ, Q, then we can build a lambda-term r which is a proof of R and in which the free variables are x or proofs of the axioms of Γ, $[x]r$ is a proof of $Q \rightarrow R = P$ and all the free variable of $[x]r$ are proofs of axioms of Γ.

8.2 Curry-Howard isomorphism

Let us consider now the types of the proofs of a proposition P.

We will introduce for each propositional variable A a type that we will also write A. So we can associate to each proposition a type, by unifying the arrow of implication and the arrow of types.

When we declare an axiom P we introduce a new variable p. We can decide that this variable is a variable of type P.

Then it is easy to show that the type the proof of every proposition is the proposition itself.

For instance the proposition $A \to A$ has a proof which is $[x : A]x$ and the type of $[x : A]x$ is $A \to A$.

In a context Γ where $(A \to A) \to B$ is an axiom f the proposition B has a proof $(f([x : A]x))$ and in the context $f : (A \to A) \to B$ the term $(f([x : A]x))$ has the type B.

It is easy to show that from any typed term t of type T, it is possible to deduce a proof in natural deduction of T.

This isomorphism between proposition and types and proofs and term is called the Curry-Howard isomorphism.

8.3 Universal Quantification

Heyting's semantics suggests to represent a proof of $\forall x : T \;\; P$ by a function which maps every term of type T with a proof of $P[x \leftarrow t]$.

These proofs look like proofs of $Q \to P$. The mains difference stands in the fact that when f is a proof of $Q \to P$ and t a proof of Q, t occurs in the proof $(f\ t)$ of P, but not in the proposition P itself, and when f is a proof of $\forall x : T \;\; P$ the term t occurs in the proof $(f\ t)$ and also may occur in the proposition $(P[x \leftarrow t])$. In order to type such a term we need to extend lambda-calculus, adding type dependent products. We add the following construction :

if $\Gamma, x : T$ types $t : T'$ (where x may occur in t and T') then Γ types $[x : T]t$ with type $(x : T)T'$ (read product of the T' for x in T).

The arrow is now a particular case of product : $T \to T'$ is $(x : T)T'$ when x is a variable which does not occur in T'.

Now we can extend our algorithm of translation of proofs written in natural deduction into lambda-terms :

If the last rule used is $\forall$-elim then we have a proof u of $(x : T)P$ and t a term of type T. $(u\ t)$ is a proof of $P[x \leftarrow t]$ since u maps every term of type t with a proof of $P[x \leftarrow t]$.

If the last rule used is $\forall$-intro, then let x be a variable of type T, we have a proof u (where x may occur) of P (where x may occur), then $[x : T]u$ is a proof of $(x : T)P$ and x is not free any more in $[x : T]u$.

Since we have written the universal quantification as a product, the Curry-Howard isomorphism can be extended to the propositions with a quantifier $\forall$. So every proof written in natural deduction can be translated into a lambda-term and every typed lambda-term can be interpreted as a proof of its type.

8.4 Encoding of others logical symbols in a non predicative system

First let us note that the existence of a type *Prop* and the possibility of quantifying over that type allow to prove theorems which are usually theorems schemas in ordinary mathematical texts. For instance for every proposition A it is possible to prove $A \to A$, but it is also possible to prove $(A : Prop)(A \to A)$. A proof of this proposition is $[A : Prop][x : A]x$.

We will use that possibility to define usual logical symbols.

8.4.1 Conjunction

In natural deduction we have three rules concerning conjunction.

If Γ proves $A \wedge B$ then Γ proves A.

If Γ proves $A \wedge B$ then Γ proves B.

If Γ proves A and Γ proves B then Γ proves $A \wedge B$.

These rules are satisfied if we take the definition

$A \wedge B = (C : Prop)((A \to B \to C) \to C)$.

Indeed if we have a proof x of $(C : Prop)((A \to B \to C) \to C)$ let us apply this proof to A.

$(x\ A) : (A \to B \to A) \to A$

and we know :

$[y : A][z : B]y : (A \to B \to A)$

so $(x\ A\ [y : A][z : B]y) : A$

so we get a proof of A.

The same stands for B:

$(x\ B\ [y : A][z : B]z) : B$

If we have a proof v of A and w of B then

$[C : Prop][f : A \to B \to C](f\ v\ w) : A \wedge B$

we can build a proof $(C : Prop)((A \to B \to C) \to C)$.

So we can represent $A \wedge B$ by $(C : Prop)((A \to B \to C) \to C)$.

$\wedge = [A : Prop][B : Prop](C : Prop)((A \to B \to C) \to C)$

$proj1 = [A : Prop][B : Prop][x : A \wedge B](x\ A\ [y : A][z : B]y)$

$proj2 = [A : Prop][B : Prop][x : A \wedge B](x\ A\ [y : A][z : B]z)$

$conj = [v : A][w : B][C : Prop][f : A \to B \to C](f\ v\ w)$

8.4.2 Disjunction, Existential quantification and Contradiction

The same thing stands for the disjunction and the existential quantification : $\vee$, *or_introl*, *or_intror*, *Ex*, *ex_intro*, $\{\}$ are lambda terms.

No lambda term correspond to the rules or-elim, ex-elim, $\{\}$-elim since the lambda-terms $\vee$, *Ex* and $\{\}$ are such that if P, Q, K are propositions, u a proof of $P \vee Q$, v a proof of $P \to K$ and w a proof of $P \to K$ then $(u\ K\ v\ w)$ is a proof of K, if u is a proof of $\exists x : T\ \ P$, and v proof of $(\forall x : T\ \ P) \to K$ then $(u\ K\ v)$ if a proof of K, if u if a proof of $\{\}$ and A a proposition the $(u\ A)$ is a proof of A.

Let us note that there are a few syntactic features in order to write proposition in an easyer way.

9 Miscellaneous commands

9.1 Running and quitting the system

9.1.1 Running

In order to run the system, type: `constr`, you get the prompt symbol `#`. You are in CAML interactive loop.

If you want to use the system in an interactive mode type: `go();;` and you get the prompt symbol `->`. You are in the Constructions interactive loop.

If you want to check a proof written in an file, type: `V ''<name of file>'';;`

9.1.2 Quitting

When the prompt symbol is `->` type: `Drop.`, you get the CAML prompt symbol `#`. Then type: `quit();;` to quit CAML.

9.2 Printing the state of the context

The current context can be printed with the following commands :

`Print` prints the last items, i.e. all the axioms and declarations and all the theorems and definitions written after the last axiom or declaration.

`Print All` print all the context.

`Print <ident>` print all the items written after the item <ident>.

`Inspect <num>` prints the <num> last items.

9.3 Resetting the context

These commands are used to reset the system in a previous state, in order to correct mistakes for instance.

`Reset <ident>` resets the system to the state it was in before writing the item <ident>.

`Reset After <ident>` resets the system to the state it was in after writing <ident>.

`Reset Section <ident>` resets the system to the state it was in before openning the section <ident>.

`Reset Initial` resets the system to its initial state, (i.e. the initial context corresponds to the standard prelude.)

References

[1] N.G. De Bruijn "The Mathematical Vernacular, A Language For Mathematics With Typed Sets" Procedings, Workshop on Programming Logic, Marstrand, Sweden, 1987

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

[3] Th. Coquand. "Metamathematical investigations of a Calculus of Constructions" 1989.

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

[5] T. Coquand, G. Huet. "The Calculus of Constructions." Information and Computation, Volume 76, 1988.

[6] G. Huet "A Uniform Approach to Type Theory" Rapport de recherche INRIA no 795 , Fevrier 88. Courses Notes, Institute on Logical Foundation of Functional Programming, University of Texas at Austin, June 1987. To appear, Addison-Wesley, 1988.

[7] G. Huet "The Constructive Engine" Conférence invitée, 2nd European Symposium on Programming, Nancy, March 1988.

[8] Ch. Paulin-Mohring. "Extraction de programmes dans le Calcul des Constructions." Thèse, Université Paris 7, 1989.

[9] Ch. Paulin-Mohring. "Extracting $F\omega$ programs from proofs in the Calculus of Construction." Proceedings of POPL 89.

[10] D. Simon "Checking Natural Language Proofs" 9th International Conference on Automated Deduction.

Inductive definitions in the Calculus of Constructions

Christine Paulin-Mohring

DRAFT

Introduction

During the development of part of mathematics or programs, we need to define new notions. For example, we will use natural numbers or ordinals, order relation on these types etc. It is possible to start with a fixed set of predefined notions and to use them in order to code new concepts. But it is more convenient to be able to directly express these notions inside the system.

There exists a large class of notions that can be defined using the same pattern. It is possible from a description of their *constructors* (how to build new elements of this type) to derive a complete specification of the notion. We call them *inductive definitions.*

This feature can be used to introduce new constants and reduction rules in a system like Martin-Löf's type system [4, 3] or in the Calculus of Constructions [2]. But an inductive definition can also be internally represented by closed terms in an impredicative system. This has been studied by Böhm and Berarducci [1] for the second-order polymorphic λ-calculus. This can be generalized to the Calculus of Constructions and has been described by F. Pfenning [6] and in our thesis [5].

We have implemented a "macro-command" that generates the type , introduction and elimination rules from the specification of an inductive definition. We describe in this note the general rules that correspond to the command `Inductive` of the vernacular.

Impredicativity. The ability of coding internally product or natural numbers uses the impredicativity of the system. We have the following inference rule :

$$\frac{x : A \vdash B \in \mathit{Prop}}{\vdash (x : A)B \in \mathit{Prop}}$$

It may be applied with $A = \mathit{Prop}$. Then we get that $(x : \mathit{Prop})B$ is of type *Prop.*

Propositions and types. Because we use the impredicativity, the type of natural numbers will be *Prop.* From our point of view, we identify propositions and types and then proofs and programs. In the following, the constant *Prop* will be replaced by $*$.

1 Non-recursive inductive types

1.1 Product

Let A and B be of type $*$, we want to build an object $A \times B$ of type $*$ that represents the binary product of A and B.

The *specification* of $A \times B$ is a type with only one constructor pair of type $A \to B \to A \times B$ and two destructors fst of type $A \times B \to A$ and snd of type $A \times B \to B$. We want also the two convertibility rules :

$$(\mathsf{fst}\ (\mathsf{pair}\ a\ b)) = a \qquad (\mathsf{snd}\ (\mathsf{pair}\ a\ b)) = b$$

It is equivalent to use the two projections or only one elimination rule ProdElim of type $A \times B \to (C : *)(A \to B \to C) \to C$ with the convertibility rule :

$$(\mathsf{ProdElim}\ (\mathsf{pair}\ a\ b)\ C\ f) = (f\ a\ b)$$

It appears that in the second order typed λ-calculus (the system F of J.-Y. Girard) and then in the Calculus of Constructions, it is possible to find terms with only A and B as free variables that meet the specification of the product. We take :

$$\begin{array}{ll} A \times B & \equiv (C : *)(A \to B \to C) \to C \\ & \in * \\ \mathsf{ProdElim} & \equiv [H : A \times B]H \\ & \in A \times B \to (C : *)(A \to B \to C) \to C \\ \mathsf{pair} & \equiv [a : A][b : B][C : *][f : A \to B \to C](f\ a\ b) \\ \in A \to B \to A \times B & \end{array}$$

It is easy to show that the term $(\mathsf{ProdElim}\ (\mathsf{pair}\ a\ b)\ C\ f)$ reduces by the β-rule to the term $(f\ a\ b)$.

We remark that the pure λ-term extracted from $(\mathsf{pair}\ a\ b)$ is the usual coding of pairs : $\lambda f.(f\ a\ b)$.

1.2 Disjoint sums

The other basic case of non-recursive inductive type is the disjoint sum $A + B$ of two types A and B. This type has two constructors inl of type $A \to A + B$ and inr of type $B \to A + B$. The rule of elimination (definition by cases) is SumElim of type

$$A + B \to (C : *)(A \to C) \to (B \to C) \to C$$

with the rules of convertibility :

$$(\mathsf{SumElim}\ (\mathsf{inl}\ a)\ C\ f\ g) = (f\ a) \qquad (\mathsf{SumElim}\ (\mathsf{inr}\ b)\ C\ f\ g) = (g\ b)$$

In this case also, it appears that just taking

$$A + B \equiv (C : *)(A \to C) \to (B \to C) \to C,$$

we get a representation of the disjoint sum.

$$\begin{array}{lll} A+B & \equiv (C:*)(A \to C) \to (B \to C) \to C & \in * \\ \mathsf{SumElim} & \equiv [H : A+B]H & \\ & \in A+B \to (C:*)(A \to C) \to (B \to C) \to C & \\ \mathsf{inl} & \equiv [a:A][C:*][f:A \to C][g:B \to C](f\ a) & \in A \to A+B \\ \mathsf{inr} & \equiv [b:B][C:*][f:A \to C][g:B \to C](g\ b) & \in B \to A+B \end{array}$$

It is easy to check that the convertibility rules are satisfied using β-reduction.

1.3 General case of non-recursive inductive type

We describe now the general scheme of definition of non-recursive inductive types. Let Γ be an environment. We write $\Gamma \vdash A$ *Type* to means that A is of type $*$ or $Type(i)$.

Definition 1 (specification of constructors) *Let Γ be an environment and X be a variable not occurring in Γ. A specification of a constructor of X (in Γ) is either X or $(x : A)B$ if $\Gamma \vdash A$ Type and B is a specification of a constructor of X in $\Gamma, x : A$.*

In the following, an environment Γ is given and X denotes a variable not occuring in Γ.

Definition 2 *An* inductive definition *of X in Γ is given by a list of specifications of constructors of X in Γ.*

We show now how to build terms in the environment Γ that will represent an implementation of an inductive definition.

Theorem 1 *Let $[F_1, \dots, F_n]$ be an inductive definition of X in Γ, then there exist terms M, c_1, ..., c_n and* MElim *such that :*

$$\begin{array}{l} \Gamma \vdash M \in * \\ \forall\, 1 \le i \le n. \quad \Gamma \vdash c_i \in F_i[X/M] \\ \Gamma \vdash \mathsf{MElim} \in M \to (X:*)F_1 \to \cdots F_n \to X \end{array}$$

We have for i between 1 and n, if $F_i = (x_1 : A_1) \dots (x_p : A_p)X$ and if the following terms are well-typed :

$$(\mathsf{MElim}\ (c_i\ a_1 \dots a_p)\ C\ f_1 \dots f_n) =_\beta (f_i\ a_1 \dots a_p)$$

It is sufficent to take :

$$\begin{array}{lll} M & \equiv (X:*)F_1 \to \cdots F_n \to X & \in * \\ \mathsf{MElim} & \equiv [H:M]H & \in M \to (X:*)F_1 \to \cdots F_n \to X \end{array}$$

and for i between 1 and n, if $F_i = (x_1 : A_1) \dots (x_p : A_p)X$:

$$\begin{array}{ll} c_i & \equiv [x_1 : A_1] \dots [x_p : A_p][X:*][f_1 : F_1] \dots [f_n : F_n](f_i\ x_1 \dots x_p) \\ & \in (x_1 : A_1) \dots (x_p : A_p)M \end{array}$$

1.4 Examples

Absurdity. It is possible to define a type with an empty list of specifications of constructors. We get the usual coding of absurdity :

$$\bot \equiv (C : *)C$$

without introduction rule and with an elimination rule of type $\bot \rightarrow (C : *)C$.

Dependent product. Let A be a type and P be a dependent proposition (P of type $A \rightarrow *$). The existential proposition $\exists x : A.(P\ x)$ is identified with the dependent product $\Sigma x : A.(P\ x)$, that is the type of pairs (a, p) with a of type A and p of type $(P\ a)$. The dependent product has one constructor of type :

$$(x : A)(P\ x) \rightarrow (\Sigma x : A.(P\ x))$$

The elimination term corresponds to the usual rule of elimination for existential propositions :

$$(\exists x : A.(P\ x)) \rightarrow (C : *)((x : A)(P\ x) \rightarrow C) \rightarrow C$$

ML concrete types. An ML concrete type can be represented by a type in the Calculus of Constructions. The elimination rule will correspond to the scheme of definition by pattern.

Remark. It is possible to represent these inductive definitions just using dependent product, disjoint sum and the "unit" type (type with one 0-ary constructor). But this direct coding is more convenient to use.

2 Recursive inductive types

We now look at the definition of recursive types. We assume that types in specifications are in normal form.

2.1 Positivity

In the Calculus of Constructions every term is strongly normalizable. Then it is not possible to represent every kind of recursive type. In ML, for example, it is possible to define the following type :

```
type L = lambda of L->L;;
```

We get a function `lambda` of type `(L->L)->L` and using matching we get a term of type `L->(L->L)`. It is then very easy to build a non-normalizing term without recursion, just simulating the well-known λ-term $\Delta = (\lambda x.(x\ x)\ \lambda x.(x\ x))$.

We will be able to represent in the Calculus of Constructions recursive types with some restriction of positivity.

Definition 3 *Let A be a type, X be a variable of type $*$, we define two mutually recursive predicates $Pos_X(A)$ and $Neg_X(A)$.*

- *If X does not occur in A then $Pos_X(A)$ and $Neg_X(A)$ are satisfied.*
- *$Pos_X(X)$ is satisfied.*
- *$Pos_X((x:A)B)$ is satisfied if $Pos_X(B)$ and $Neg_X(A)$ are.*
- *$Neg_X((x:A)B)$ is satisfied if $Neg_X(B)$ and $Pos_X(A)$ are.*

We will say that X is positive (resp. negative) in A if the predicate $Pos_X(A)$ (resp. $Neg_X(A)$) is satisfied.

Remark. The variable X is positive in A means that either it occurs at a positive position or else it does not occur.

Lemma 1 *Let Γ be an environment, X be a variable and A be a term such that $\Gamma, X : * \vdash A$ Type. Let M, N and f be terms such that $\Gamma \vdash M \in *$, $\Gamma \vdash N \in *$ and $\Gamma \vdash f \in M \to N$. If $Pos_X(A)$ (resp. $Neg_X(A)$) is satisfied then there exists a term $A(f)$ such that :*

$$\Gamma \vdash A(f) : A[X/M] \to A[X/N] \quad (\textit{resp.}\ \ \Gamma \vdash A(f) \in A[X/N] \to A[X/M]).$$

Restriction. The construction of the term $A(f)$ is simplified in the restricted case where the products $(x:P)Q$ occurring in A are either non-dependent product $(P \to Q)$ or such that X does not occur in P. It is only implemented in this case, that seems to be the only one used in the examples.

In this case, the construction of $A(f)$ is very easy.

- If X does not occur in A then $A(f) = [x:A]x$.
- $A(f) = f$ if $A = X$.
- If X is positive in $A = (x:P)Q$ then

$$A(f) = [u : A[X/M]][x : P[X/N]](Q(f)\ (u\ (P(f)\ x))).$$

- If X is negative in $A = (x:P)Q$ then

$$A(f) = [u : A[X/N]][x : P[X/M]](Q(f)\ (u\ (P(f)\ x))).$$

It is easy to check that $A(f)$ has the correct type. The restriction allows to prove that the hypothesis of induction $\Gamma, X : * \vdash A$ *Type* is satisfied.

2.2 One unary recursive constructor

We start with the basic example of a recursive type with one constructor of specification $A \to X$. We must assume that X is positive in A. Let M be the term $(X:*)(A \to X) \to X$ of type $*$, then we are going to build its constructor c, that is a term of type $A[X/M] \to M$. This term will have the following structure :

$$[c : A[X/M]][X : *][f : A \to X](f\ \alpha)$$

with α be a term of type A. If we find a term g of type $M \to X$ then the term $\alpha = (A(g)\ c)$ will be of type A. We take $g = [x : M](x\ X\ f)$.

Primitive recursion As an elimination rule, we get a term :

$$[H : M]H \text{ of type } M \to (X : *)(A \to X) \to X$$

This may seem an unusual elimination rule. It corresponds to a particular scheme of recursive definition called iteration. In the case of integers we will get the program that takes a natural number n, an initial element x_0 of type X and a function f of type $X \to X$ and that gives the result of iterating n times f from x_0. In the case of lists we will get the analogue of the ML function `list_it`.

We want a bit more, for example the pattern matching of type :

$$M \to (X : *)(A[X/M] \to X) \to X$$

This scheme is a particular case of a more general one that corresponds to the primitive recursion for natural numbers and that is of type :

$$M \to (X : *)(A[X/X \times M] \to X) \to X$$

Because X is positive in A and the existence of terms of type $X \times M \to X$ and $X \times M \to M$, there exist terms of type $A[X/X \times M] \to A$ and $A[X/X \times M] \to A[X/M]$. The scheme of primitive recursion gives both iteration and pattern-matching. We are going to show that using pairing and iteration it is possible to build a term for primitive recursion.

Definition of primitive recursion. The terms for product, pairing and projections have been defined before. Let m be of type M, X be of type $*$ and f be of type $A[X/X \times M] \to X$. We first build with an iterative scheme a term Mrec of type $X \times M$. Let us take :

$$\mathsf{Mrec} = (m\ X \times M\ [y : A[X/X \times M]](\mathsf{pair}\ (f\ y)\ (\mathsf{c}\ (A(\mathsf{snd})\ y))))$$

It is easy to check that this term is well-formed : y is of type $A[X/X \times M]$, snd is of type $X \times M \to M$, then $(A(\mathsf{snd})\ y)$ is of type $A[X/M]$ and the constructor c is of type $A[X/M] \to M$.

In order to finally get a term of type X, we just apply the first projection.

2.3 Natural numbers

In general we want to combine disjoint sums, products and recursion in the definition of an inductive type. In the case of the natural numbers, the specification is with two constructors :

$$0 \in \mathsf{nat} \quad \text{and} \quad \mathsf{S} \in \mathsf{nat} \to \mathsf{nat}$$

Following the same scheme we take :

$$\begin{array}{lcl} \mathsf{nat} & \equiv & (C:*)C \to (C \to C) \to C \\ 0 & \equiv & [C:*][x:C][f:C \to C]x \\ \mathsf{S} & \equiv & [n:\mathsf{nat}][C:*][x:C][f:C \to C](f\ (n\ C\ x\ f)) \end{array}$$

The primitive recursion is the following term :

$$\begin{array}{lcl} \mathsf{NatRec} & \equiv & [n:\mathsf{nat}][C:*][x:C][f:C \times \mathsf{nat} \to C] \\ & & (\mathsf{fst}\ (n\ C \times \mathsf{nat}\ (\mathsf{pair}\ x\ 0)\ [y:C \times \mathsf{nat}](\mathsf{pair}\ (f\ y)\ (\mathsf{S}\ (\mathsf{snd}\ y))))) \\ & \in & \mathsf{nat} \to (C:*)C \to (C \times \mathsf{nat} \to C) \to C \end{array}$$

3 General case

We give the general results concerning (recursive or not) inductive types. This generalizes the results of the preceding section. The definition of specification of constructors is extended.

Definition 4 (specification of constructors) *Let Γ be an environment and X be a variable not occurring in Γ. A specification of constructor of X (in Γ) is either X or $(x:A)B$ if $\Gamma, X:* \vdash A$ Type, $Pos_X(A)$ is satisfied and B is a specification of constructor of X in $\Gamma, x:A$.*

In the following, an environment Γ is given and X denotes a variable not occurring in Γ.

Definition 5 *An* inductive definition *of X in Γ is given by a list of specification of constructors of X in Γ.*

Let A be a specification of constructor of X, and N be a term. We define a "left-substitution" $A\{X/N\}$ inductively on the structure of the specification. If $A = X$ then $A\{X/N\} = X$ and if $A = (y:B)C$ then $A\{X/N\} = (y:B[X/N])C\{X/N\}$.

Theorem 2 *Let $[F_1, \ldots, F_n]$ be an inductive definition of X in Γ, then there exist terms M, $c_1, \ldots, c_n$ and* MRec *such that :*

$$\begin{array}{l} \Gamma \vdash M \in * \\ \forall\, 1 \le i \le n. \quad \Gamma \vdash c_i \in F_i[X/M] \\ \Gamma \vdash \mathsf{MRec} \in M \to (X:*)F_1\{X/X \times M\} \to \cdots F_n\{X/X \times M\} \to X \end{array}$$

It is sufficient to take :

$$M \equiv (X:*)F_1 \to \cdots F_n \to X : *$$

and for i between 1 and n, if $F_i = (x_1 : A_1)\dots(x_p : A_p)X$:

$$\begin{aligned} c_i \ &\equiv [x_1 : A_1[X/M]]\dots[x_p : A_p[X/M]][X:*][f_1 : F_1]\dots[f_n : F_n] \\ &\quad (f_i\ (A_1(g)\ x_1)\dots(A_p(g)\ x_p)) \\ &\in F_i[X/M] \end{aligned}$$

with $g = [m : M](m\ X\ f_1 \dots f_n)$ of type $M \to X$. To build the primitive recursive combinator is a bit more complicated. Let m be of type M, X be of type $*$ and $f_1, \dots f_n$ be of type respectively $F_1\{X/X \times M\}, \dots F_n\{X/X \times M\}$. We build a term of type X in the following way :

$$(\mathsf{fst}\ (m\ X \times M\ \phi_1 \dots \phi_n))$$

Where ϕ_i of type $F_i[X/X \times M]$ is the following term :

$$\begin{aligned} \phi_i = \ & [x_1 : A_1[X/X \times M]]\dots[x_p : A_p[X/X \times M]] \\ & (\mathsf{pair}\ (f_i\ x_1 \dots x_p)\ (c_i\ (A_1(\mathsf{snd})\ x_1)\dots(A_p(\mathsf{snd})\ x_p))) \end{aligned}$$

3.1 Problems

There is a difficulty with the reduction rules for the primitive recursion operator. For the example of natural numbers, we have :

$$(\mathsf{NatRec}\ 0\ C\ x\ f) =_\beta x$$

For the reduction in $(\mathsf{S}\ n)$ we are expecting :

$$(\mathsf{NatRec}\ (\mathsf{S}\ n)\ C\ x\ f) =_\beta (f\ (\mathsf{pair}\ (\mathsf{NatRec}\ n\ C\ x\ f)\ n))$$

but we only get :

$$(\mathsf{NatRec}\ (\mathsf{S}\ n)\ C\ x\ f) =_\beta (f\ (\mathsf{pair}\ (\mathsf{NatRec}\ n\ C\ x\ f)\ \phi_n))$$

with ϕ_n a program that depends on n and that satisfies :

$$\phi_0 =_\beta 0 \quad \text{and} \quad \phi_{(\mathsf{S}\ n)} =_\beta (\mathsf{S}\ \phi_n)$$

So we have $\phi_n =_\beta n$ if $n = (\mathsf{S}^p\ 0)$ but in general we do not have $\phi_n =_\beta n$. It is possible to prove using for example Leibniz's equality that $\phi_n = n$ for each n that satisfies the induction principle. For natural numbers, it is possible to justify the consistency of the induction principle, but this principle is not provable.

We lost the fact that the convertibility rule for primitive recursion is an internal conversion (the best we can do is to find a proof of this equality). A most crucial problem is that this program ϕ_n does a "copy" of n and that such copies may be iterated leading to very inefficient reductions.

This suggests to implement "primitive" recursive types. We will then get for the natural numbers the right internal rule of conversion and also the ability of doing proofs (or building terms) using induction.

4 Predicates

The definition of inductive types can be generalized to the definition of predicates or n-ary relations. It corresponds to the concept of defining a relation to be the smallest relation such that some closure properties hold. The basic case of such a predicate is Leibniz's equality. Actually using Leibniz's equality, dependent product, disjoint sum, unit type and recursive predicate with one unary constructor, it is possible to code every kind of inductive definition.

4.1 Equality

We want to define the intensional equality on a type A. It is possible to say that it is the smallest reflexive relation on A. The specification is $(x : A)(\mathbf{eq}\ x\ x)$. We get the following definition :

$$\mathbf{eq} \equiv [x,y : A](R : A \to A \to *)((x : A)(R\ x\ x)) \to (R\ x\ y)$$

With this definition we get a constructor **refl** of type $(x : A)(\mathbf{eq}\ x\ x)$. It is also possible to define, given an x of type A, the predicate $\mathbf{eq}_x$ on A "to be equal to x". This is the smallest predicate that is true for x. The specification is $(\mathbf{eq}_x\ x)$ (x is given). We then get the usual definition of Leibniz's equality :

$$\mathbf{eq}_x \equiv [y : A](P : A \to *)(P\ x) \to (P\ y)$$

We get a constructor $\mathbf{refl}_x$ of type $(\mathbf{eq}_x\ x)$. We then "abstract" these definitions with respect to x. It is very easy to show that both notions of equality are isomorphic.

4.2 General case

We now need to define the positive occurrences of a predicate in a type. We take the case of an unary predicate on a type B.

Definition 6 *Let P be a variable of type $B \to *$ and let M be a type. We say that P is positive in M if the only occurrences of P in M are in expressions like $(P\ b)$ with b a term in which P does not occur and if for a variable X of type $*$, $Pos_X(M[P/[x : A]X])$ is satisfied (it means that if we replace every expression $(P\ b)$ of M by X then we get a type in which X is positive).*

We define for two predicates M and N the type $M \subset N$ to be $(x : B)(M\ x) \to (N\ x)$. We get the following result :

Lemma 2 *Let Γ be an environment, X be a variable and A be a term such that $\Gamma, P : B \to * \vdash A$ Type. Let M, N and f be terms such that $\Gamma \vdash M \in B \to *$, $\Gamma \vdash N \in B \to *$ and $\Gamma \vdash f \in M \subset N$. If P occurs positively in A then there exists a term $A(f)$ such that :*

$$\Gamma \vdash A(f) \in A[P/M] \to A[P/N]$$

We only have to change the basic case of the definition of $A(f)$ for types. If $A = (P\ a)$ then $A(f) = (f\ a)$ is of type $(M\ a) \rightarrow (N\ a)$.

We give the definition of a specification of an inductive predicate.

Definition 7 (Specification of constructor) *Let Γ be an environment and X be a variable of type $B \rightarrow *$ not occurring in Γ. A specification of constructor of X (in Γ) is either $(X\ a)$ with some term a such that X does not occur in a or $(x : A)B$ if $\Gamma, X : B \rightarrow * \vdash A$ Type, if X is positive in A and if B is a constructor of X in $\Gamma, x : A$.*

In the following, an environment Γ is given and X denotes a variable not occurring in Γ.

Definition 8 *An* inductive definition *of X in Γ is given by a list of specifications of constructors of X in Γ.*

Let M and N be two predicates, we call $M \cap N$ the predicate $[x : B](M\ x) \times (N\ x)$.

Theorem 3 *Let $[F_1, \ldots, F_n]$ be an inductive definition of a predicate X in Γ, then there exist terms M, $c_1, \ldots, c_n$ and* MRec *such that :*

$$\begin{array}{l} \Gamma \vdash M \in B \rightarrow * \\ \forall\ 1 \leq i \leq n. \quad \Gamma \vdash c_i \in F_i[X/M] \\ \Gamma \vdash \mathsf{MRec} \in (x : B)(M\ x) \\ \qquad\qquad \rightarrow (X : B \rightarrow *)F_1\{X/X \cap M\} \rightarrow \cdots F_n\{X/X \cap M\} \rightarrow (X\ x) \end{array}$$

It is sufficient to take :

$$M \equiv [x : B](X : *)F_1 \rightarrow \cdots F_n \rightarrow (X\ b) \in B \rightarrow *$$

and for i between 1 and n, if $F_i = (x_1 : A_1) \ldots (x_p : A_p)(X\ a)$:

$$\begin{array}{rcl} c_i & \equiv & [x_1 : A_1[X/M]] \ldots [x_p : A_p[X/M]] \\ & & [X : B \rightarrow *][f_1 : F_1] \ldots [f_n : F_n] \\ & & (f_i\ (A_1(g)\ x_1) \ldots (A_p(g)\ x_p)) \\ & \in & F_i[X/M] \end{array}$$

with $g = [x : B][m : (M\ x)](m\ X\ f_1 \ldots f_n)$ of type $M \subset X$. We build also the "primitive recursive combinator". Let x be of type X, m be of type $(M\ x)$, X be of type $B \rightarrow *$ and $f_1, \ldots f_n$ be of type respectively $F_1\{X/X \cap M\}, \ldots F_n\{X/X \cap M\}$. We still call snd the term of type $X \cap M \subset M$. We build a term of type $(X\ x)$ in the following way :

$$(\mathsf{fst}\ (m\ X \cap M\ \phi_1 \ldots \phi_n))$$

Where ϕ_i of type $F_i[X/X \cap M]$ is the following term :

$$\begin{array}{rl} \phi_i & = [x_1 : A_1[X/X \cap M] \ldots [x_p : A_p[X/X \cap M]] \\ & \quad (\mathsf{pair}\ (f_i\ x_1 \ldots x_p)\ (c_i\ (A_1(\mathsf{snd})\ x_1) \ldots (A_p(\mathsf{snd})\ x_p))) \end{array}$$

5 The *Inductive* command

The *Inductive* command is a macro that generates the relation, its constructors, and the rule of "primitive recursion" that becomes an elimination in the non-recursive case.

5.1 Syntax

To specify an *inductive definition* we need the "arity" of the definition (it means the type of the notion we want to define) and the specification of the constructors. This may be specified with a notation like :

$$\mu X : \textit{arity.list of constructors}$$

that emphasizes the fact that X is bound in this specification. But we want also to specify the name to give to the type and its constructors. Then the *Inductive* command will have the following syntax :

Inductive *definition name* : *arity* = *list of named constructors.*

A *definition* can be one of the keywords **Definition**, **Define**, **Let**, **Global** or **Local** with the same rules of scope as in the mechanism of definition.

A *named constructor* has the following syntax :

name : *constr*

and a *list of named constructors* is either a named constructor or has the form :

named constructor | *list of named constructors*

There are some special syntactical constructions for particular arities.

Inductive Proposition *name* = *list of named constructors.*

declares the arity to be *Prop.*

Inductive Data *name* = *list of named constructors.*

declares the arity to be *Data.*

5.2 Behavior

Assume that we declare an inductive definition of name X of arity K with a list of named constructors $[(n_1 : C_1), \ldots, (n_n : C_n)]$. The system checks that considering X as a variable of type K, the types C_i are specifications of constructors of X. It declares a constant of name X and of type K, then the constructors of name c_i and of type C_i (with now X considered as a constant). It then defines the primitive recursive combinator with the name *X_elim* in the non-recursive case and *X_rec* in the recursive case.

Remark. If the recursive type M we are defining is an informative one (of type *Data* or *Spec*) then the intermediate value of type $M \rightarrow (C : *)F'_1 \rightarrow \cdots \rightarrow F'_p \rightarrow (M \times C)$ needed for the computation of the primitive recursive term is named M_REC and keeped in the environment. This feature is done in order to simplify the proofs of reduction rules for the term of primitive recursion.

5.3 Parameters

It appears that very often we are defining inductive notions in an environment with parameters. For example the product is defined with respect to two types A and B. It will be possible to define the product to be an inductive relation of type $* \rightarrow * \rightarrow *$. This definition is equivalent to the one we gave before and more cumbersome. The right definition of product is the following one.

```
Variables A, B : Prop.
Inductive Definition and : Prop = pair : A->B->and.
```

and then to abstract the various definitions with respect to A and B. We add some syntactic sugar and allow to directly write the following declaration :

```
Inductive Definition and [A,B:Prop] : Prop = pair : A->B->(and A B).
```

This generates the sequence of commands described before. The general syntax of inductive definitions with parameters is :

Inductive *definition name* [*list of declarations*] : *arity*
= *list of named constructors.*

A *list of declarations* is either a *declaration* or :

declaration ; *list of declarations*

And a *declaration* is

list of names : *constr*

A *list of names* is names separated by commas.

If the arity is *Prop*, it is possible to use the special syntactic form :

Inductive Connective *name* [*list of declarations*] = *list of named constructors.*

Several examples of inductive definitions may be found in the prelude file.

References

[1] C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39, 1985.

[2] Th. Coquand and C. Paulin-Mohring. Inductively defined types. 1989. presented at the workshop on Programming Logic.

[3] P. Dybjer. An inversion principle for Martin-Löf's type theory. In *Workshop on Programming Logic*, University of Göteborg and Chalmers University of Technology, 1989. to appear as a report of the Programming Methodology Group.

[4] N. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, 1988.

[5] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* Thèse d'université, Université Paris 7, 1989.

[6] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the calculus of constructions. 1989. presented at the conference : Mathematical foundations of programming language semantics.

The Tactics Theorem Prover, User's guide, Version 4.10

Thierry Coquand

INRIA

1 General presentation

The calculus of constructions is a typed λ-calculus which is used as a general framework where one can develop in a uniform way functional programs and proofs. The "theoretical" basis of such a system is the close analogy between propositions and types (known as Curry-Howard "isomorphism"). Here this idea is taken at a pragmatic level, both in the implementation of the logic, and in the notation. It gives a good presentation of natural deduction[1]. Since we want to do manipulation on proofs (cut-elimination, pruning, realisability,...) it is natural to follow the Automath "proof-as-objects" paradigm.

We will first recall what is the formal system we use before a presentation of our search system, in order to get a self-contained presentation.

The inference rules define inductively a ternary relation $\Gamma \vdash M : P$ where Γ is a context of typed variables and constants.

1.1 Inference rules

From the user's point of view, the following system is implemented.

$$\frac{\Gamma \text{ is valid}}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}} \qquad (TI1)$$

$$\frac{\Gamma \text{ is valid}}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}} \qquad (TI2)$$

$$\frac{\Gamma \text{ is valid} \quad x \text{ is bound in } \Gamma}{\Gamma \vdash x : \Gamma_x} \qquad (TI3)$$

$$\frac{\Gamma, x : M \vdash N : P}{\Gamma \vdash [x : M].N : (x : M)P} \qquad (TI4)$$

$$\frac{\Gamma, x : M \vdash N : \mathsf{Prop}}{\Gamma \vdash (x : M)N : \mathsf{Prop}} \qquad (TI5)$$

$$\frac{\Gamma \vdash M : \mathsf{Type} \quad \Gamma, x : M \vdash N : \mathsf{Type}}{\Gamma \vdash (x : M)N : \mathsf{Type}} \qquad (TI6)$$

$$\frac{\Gamma \vdash M : \mathsf{Prop} \quad \Gamma, x : M \vdash N : \mathsf{Type}}{\Gamma \vdash (x : M)N : \mathsf{Type}} \qquad (TI7)$$

[1]For a general presentation of natural deduction, and its use, see the book of Kleene "Mathematical Logic", 1967.

$$\frac{\Gamma \vdash M : (x : Q)P \quad \Gamma \vdash N : R \quad Q \equiv R}{\Gamma \vdash (M\ N) : [N/x]P} \tag{TI8}$$

1.1.1 Type Equality Rule

$$\frac{\Gamma \vdash M : N \quad \Gamma \vdash P : \mathsf{Prop} \quad N \equiv P}{\Gamma \vdash M : P} \tag{TE1}$$

$$\frac{\Gamma \vdash M : N \quad \Gamma \vdash P : \mathsf{Type} \quad N \equiv P}{\Gamma \vdash M : P} \tag{TE2}$$

1.1.2 Context formation

The empty context is valid, and

$$\frac{\Gamma \vdash P : \mathsf{Prop}}{\Gamma, x : P \text{ is valid}} \tag{CT1}$$

$$\frac{\Gamma \vdash P : \mathsf{Type}}{\Gamma, x : P \text{ is valid}} \tag{CT2}$$

$$\frac{\Gamma \vdash M : P}{\Gamma, x = M : P \text{ is valid}} \tag{CT3}$$

In these rules, and everywhere in this paper, the square brackets $[x : T]M$ stands for the typed λ-abstraction $(\lambda x : T)M$.

The basic system is simply a type checker for these rules. We call a *type* (resp. a *proposition*) a term of type Type (resp. Prop). The conversion between terms is β,δ-conversion: the rule $(CT3)$ allows the user to introduce constants, and δ-conversion consists in unfolding the definitions.

The analogy between functional constructions (at the level of types) and proof constructions (at the level of propositions) is clearly reflected in these rules. The principle used is "Truth is Inhabitation", so that to prove that a proposition is true is to build an element of type this proposition.

Technical Remark: as it stands however, this system is logically inconsistent (that is all propositions are inhabited) and furthermore it doesn't satisfy the normalisation property!

This important drawback is solved by introducing a stratification at the type level. This stratification is hidden, so that the user does not have to bother about it. Notice that, with such a stratification, there is a set-theoretic semantics (in Zermelo-Fraenkel-Skolem set theory where Prop is interpreted as a two element set), and so we can "safely" assume the consistency of the system. In practice, the problem of stratification never occurs. In the rare case where the user tries to build an inconsistent term with respect to this stractification, the message "You have reached a paradox" appears.

There is however an "unsafe mode" for working in the Type : Type system. In the files `EXAMPLES/Log_Rel` and `EXAMPLES/Reynolds` it is shown how to build a non normalisable term in such a system.

1.2 How to use this system as a proof-checker

We can embed a logical calculus (called "minimal" calculus) in this system. First, a proposition is simply a term of type Prop. We take $P \to Q$ to be $(x : P)Q$, where P : Prop and Q : Prop. We thus get a compact representation of proofs in minimal logic. For instance $P \to Q \to P$ has for proof $[x : P][y : Q]x$, and the deduction theorem becomes λ-abstraction. We have also a representation of universal quantification: if A : Type and $P : A \to$ Prop is a predicate over the type A, the universal quantification of P over A is the term $(x : A)(P\ x)$, which is a proposition. Furthermore, we can instantiate by application: if p is a proof (i.e. a term) of $(x : A)(P\ x)$, and if t is a term of type A then $(p\ t)$ is a proof of $(P\ t)$. We can also generalise: if in the context $\Gamma, x : A$, we have a proof $p(x)$ of $(P\ x)$, then, in the context Γ, $[x : A]p(x)$ is a proof of $(x : A)(P\ x)$.

We have thus shown that we can represent universal quantification (over any types) and implication. The logic based only on implication and universal quantification is called "minimal logic" (however, notice that we can quantify over propositions, hence this calculus is higher-order). One may argue that this is the most "primitive logic", and it is possible to define other connectives from these primitives alone (this was done as soon as 1903 in B. Russell "Principles of Mathematics"!).

Without going into the details, we can say that this proof-checker is well adapted for doing proofs in intuitionistic Higher-Order Logic (see the book of J. Lambek and P.J. Scott "Higher-Order Categorical Logic" for a presentation of a similar logic). The fact that we can formulate, and prove theorems with *type* variables gives it the uniformity needed in practice (by complete analogy with the polymorphism used in CAML).

1.3 How to search a proof and build a term

For building a term, the user can declare variables (or axioms), and later discharge them. The idea of having a system with the possibility of moving in local "context", directly suggested by the language ALGOL, comes from the Automath language. It is convenient for doing proofs in natural deduction. Furthermore, in developping a mathematical theory, it is possible (and essential) to have the possibility of introducing constants.

In practice, though possible[2], it is too cumbersome to write explicitely a full proof. A proof search facility has been written in CAML. But nothing prevents the user to use this system also for building terms (using so completely the "proof as objects" paradigm), and we can hope building in this way very complicated terms (read "programs") that would have been difficult to write directly (and also, type checking is done incrementally).

The tactics theorem prover is based on ideas from LCF. It is based on the notion of tactics. Roughly speaking, a tactic is the inverse of a desired inference rule. I.e. it develops a proof top-down in a goal directed manner. Here, the program keeps simply a partial proof, whose leaves are the subgoals, which can be refined against a clause (which is the type of a correct term in this framework). The important point is that the theorem prover is *extensible* by programing more

[2]For instance, B. Jutting in the Automath project has been able to check a complete book on the foundation of analysis in a type checker used as a proof-checker, without any search facility.

tactics in CAML.

2 A simple example of the use of the tactics theorem prover

We call synthesis machine the proof searching facility which has been written in CAML. It is based on a tactic system, inspired from LCF, which permits to construct a proof step by step. It has its own toplevel, which is called from CAML by the function go.

The problem is to find a proof of a term. In a mathematical use, terms of which we search a proof are propositions. As we have seen in introduction, using completely the "proof as objects" paradigm, it may be any term. The synthesis machine permits to construct progressively a proof with help from tactics. From an initial term of which we search a proof, we go to more elementary terms to prove through a transformation we know the nature of. Knowing the nature of this transformation and assuming to know proofs of the more elementary terms, we are able to know the whole proof of the initial term. By using these transformations in a recursive way until there is no term left to prove, we are able to construct the whole proof.

The environment of the synthesis consists of a tree which is a representation of the proof (as a λ-term) that we are looking for. There are gaps in this tree-term which are the parts yet unproved of the demonstration. These gaps consist of a pair made of a proposition and a local context of variables and hypotheses. At the beginning no branch of the tree is known and at the end of the demonstration there is no gap left in the tree. The tactics are the functions which complete progressively the prooftree. They take as an argument a goal (i.e. a pair made of a term and a local context of variables and hypotheses) and give a prooftree with gaps in it, the latter being new terms to prove.

We will first give an example to illustrate what the synthesis machine offers. Then we will explain what tactics do.

Tactics are CAML functions and it is always possible to create new tactics in CAML. In the machine described here, only some tactics are given, those which are essential to construct a proof or very helpful. There are some constructors of tactics too to create more complicated tactics, that can be defined in CAML, and then used in the toplevel of the synthesis. The last section will present such a tactic.

The example we present now will show how to declare a goal, and how to build a proof of a theorem via synthesis machine.

We suppose first that we have declared in the vernacular mode the following section.

```
Section inclus.

Type U.

Definition Set.
```

```
Body U->Prop.

Definition in : U->Set->Prop = [x:U][A:Set](A x).

Definition inclus : Set->Set->Prop = [A,B:Set](x:U)(in x A)->(in x B).
```

We can now check the environment by the command Inspect n, where n is the number of the constants we want to see. We recall that `->` is the vernacular prompt.

```
->Inspect 5.

inclus : Set->Set->Prop
in : U->Set->Prop
Set : Type
*** [U :Type]
 >>>>>>> Section inclus
```

We thus see that we are in the section `inclus`, and under one variable U, with two definitions for `in` and `Set`.

The proposition to prove is given to the machine as the goal:

```
->Goal (A,B,C:Set)(inclus A B)->(inclus B C)->(inclus A C).
  (A:Set)(B:Set)(C:Set)(inclus A B)->(inclus B C)->(inclus A C)
```

Since we have to prove a universal quantification followed by implications, we use the introduction tactic repeatedly. The tactical REPEAT repeats a tactic until it fails.

```
->By (REPEAT intro).
1 subgoal
  (inclus A C)
  ============================
    H0 : (inclus B C)
    H : (inclus A B)
    C : Set
    B : Set
    A : Set
```

Note that the system prints the local context.

We now unfold the definition of inclus in the goal:

```
->Unfold inclus.
1 subgoal
  (x:U)(in x A)->(in x C)
  ============================
    H0 : (inclus B C)
    H : (inclus A B)
    C : Set
    B : Set
    A : Set
```

We can then use the introduction tactic, and the tactical REPEAT

```
->By (REPEAT intro).
1 subgoal
  (in x C)
  ============================
    H1 : (in x A)
    x : U
    H0 : (inclus B C)
    H : (inclus A B)
    C : Set
    B : Set
    A : Set
```

The goal (in x C) becomes (in x B) after one resolution with the hypothesis H0. We simply do a resolution with the Horn clause defined by the type of the hypothesis H0.

```
->Resolve H0.
1 subgoal
  (in x B)
  ============================
    H1 : (in x A)
    x : U
    H0 : (inclus B C)
```

```
    H : (inclus A B)
    C : Set
    B : Set
    A : Set
```

We can then resolve against the hypothesis H:

```
->Resolve H.
1 subgoal
  (in x A)
  ============================
    H1 : (in x A)
    x : U
    H0 : (inclus B C)
    H : (inclus A B)
    C : Set
    B : Set
    A : Set
```

Finally, the goal is proved by hypothesis H1:

```
->Exact H1.
goal proved
```

It is possible to add this theorem to the environment of the constructive machine:

```
->Save trans_inclus.
```

We can then check the environment.

```
->Inspect 2.

trans_inclus :
(A:Set)(B:Set)(C:Set)(inclus A B)->(inclus B C)->(inclus A C)
inclus : Set->Set->Prop
```

It is also possible to display the proof that has just been built:

```
->Show_proof.
[A:Set][B:Set][C:Set][H:(inclus A B)][H0:(inclus B C)]
[x:U][H1:(in x A)](H0 x (H x H1))
```

We can in the same way show the reflexivity of **inclus**.

```
Goal (A:Set)(inclus A A).
By (REPEAT intro).
Unfold inclus.
By (REPEAT intro).
By assumption.
Save ref_inclus.
```

We can then close the section.

```
->End inclus.
```

3 The tactics

Tactics enable us to construct a proof in a backward mode. We begin with a *goal* (a statement or desired theorem, or a type of a program) and the tactics reduce it to simpler and simpler subgoals. Here are the CAML types that are used. We assume a basic knowledge of CAML.

- A local context is of type `signature` which is the type of a list of variables (or hypotheses),
- a goal is of type `goal = signature * constr`
- a prooftree is of type `prooftree`, as follows:

```
type prooftree =
 INTRO of string * constr * prooftree
|APP of constr * (prooftree list)
|PF of constr
|INCOMPLET of goal ;;
```

* `tactic = goal -> prooftree.`

A good intuition for λ-calculus specialists is that the user builds Böhm tree approximations of the partial proof.

Here follow the rules which permit to construct the proofs. They are derived rules from the rules presented in the introduction.

The application rule is in a generalized form and we have written $\rightarrow$ for a variable on which the rest of the term is non dependent.

The abstraction rule is divided into two rules according to whether the term is dependent or not on the abstracted variable, but it is always syntactically the same rule.

We always deal here with a valid context Γ of variables, hypotheses (i.e. variables of type a proposition), constants and proofs of theorem (i.e. constants of type a proposition).

$$\Gamma \vdash p : M \quad (if\ (p : M)\ is\ in\ \Gamma) \qquad \text{variable}$$

$$\frac{\Gamma, x : N \vdash p : M}{\Gamma \vdash [x : N]p : (x : N)M} \quad \text{abstraction}$$

$$\frac{\Gamma, q : N \vdash p : M}{\Gamma \vdash [q : N]p : N \rightarrow M} \quad \text{abstraction, } q \text{ does not occur in } M$$

$$\frac{\Gamma \vdash p : (x_1 : N_1) \ldots (x_q : N_q)M \quad \Gamma \vdash p_i : N_i}{\Gamma \vdash (p\ N_1 \ldots N_q) : [t_i/x_i]M} \quad \text{application}$$

$$\frac{\Gamma \vdash p : M \quad M \equiv N}{\Gamma \vdash p : N} \quad \text{equality}$$

To complete the system of tactics, i.e. to be able to construct any proof, it is necessary to have a minimal set of tactics which enables us to go backwards through the rules above.

We have one basic tactic for each of the four rules above.

Let us notice that (almost) all tactics here described deal with terms in β-normal form (the only exception is the **change** tactic).

We first give some notations to describe these tactics:

We will write a goal (i.e. a term M to be proved under a local context Γ of hypotheses) by:

$$\begin{array}{c} \Gamma \\ \vdots \\ M \end{array}$$

Let's notice that the synthesis machine works implicitly with the global context which is the same for a whole proof search. And when we say that we have to prove M under the local context Γ, it means that the used context is the union of Γ and the global context.

We will write a tactic which associates to the term M to prove under the context Γ the list of terms M_i to prove under the contexts Γ_i respectively by:

$$\begin{array}{c}\Gamma\\ \vdots\\ M\end{array} \rightsquigarrow \begin{array}{c}\Gamma_1\\ \vdots\\ M_1\end{array} \ldots \begin{array}{c}\Gamma_n\\ \vdots\\ M_n\end{array}$$

We will write $\square$ for the empty list of goals (that means that the demonstration is finished).

3.1 The *assumption* tactic

It is the rule which is associated to the variable rule of λ-calculus: if a proof of the term is among the hypotheses then there is nothing left to prove.

Taking this rule the other way round, the **assumption** tactic has this effect :

$$\begin{array}{c}\Gamma\\ \vdots\\ M\end{array} \rightsquigarrow \square \; (\textit{if a proof of } M \textit{ is in } \Gamma)$$

3.2 The *intro* tactic

It is the rule which is associated to the λ-abstraction rule of the λ-calculus. There are two meanings for this rule depending on whether the abstraction is an abstraction of a dependent variable or of a non dependent variable.

Taking this rule the other way round, the **intro** tactic has this effect:

$$\text{for a variable} \quad \begin{array}{c}\Gamma\\ \vdots\\ (x:U)M\end{array} \rightsquigarrow \begin{array}{c}\Gamma,(x:U)\\ \vdots\\ M\end{array}$$

$$\text{for an hypothesis} \quad \begin{array}{c}\Gamma\\ \vdots\\ N \rightarrow M\end{array} \rightsquigarrow \begin{array}{c}\Gamma,(HypN:N)\\ \vdots\\ M\end{array}$$

They are two versions of the introduction tactic. One version, called `intro_with`, is of type `string->tactic` and the user has to give a name to the variable or hypothesis he introduces. The other version `intro : tactic` has an implicit mechanism for generating internal names.

3.3 The *resolve* tactic

This is the inversion of the rule of application to a variable or constant. Here is a standard example: the tactic `resolve`. This tactic can be used directly from the vernacular by using the command `Resolve` (see below how to call a tactic defined in CAML from the vernacular).

```
->Goal (A,B,C:Prop)(A->B->C)->(A->B)->A->C.
  (A:Prop)(B:Prop)(C:Prop)(A->B->C)->(A->B)->A->C

->By (REPEAT intro).
1 subgoal
  C
  ============================
    H1 : A
    H0 : A->B
    H : A->B->C
    C : Prop
    B : Prop
    A : Prop


->Resolve H.
2 subgoals
  A
  ============================
    H1 : A
    H0 : A->B
    H : A->B->C
    C : Prop
    B : Prop
    A : Prop
subgoal 2 is:
  B
```

The system prints the local context only for the first (in the lexicographic ordering) subgoal.

As seen in the first general example of the transitivity of inclusion, the tactic refines on the head-normal form of the type of the given construction.

3.4 The *change* tactic

It is one of the rules associated to the type equality rule.

The **change** tactic enables us to substitute a term to prove with an equivalent one. It checks that the new term is β, δ-convertible to the initial one.

Its syntax is **change** *com*, where *com* is a term in concrete syntax and its effect is:

$$\begin{array}{ccc}\Gamma & & \Gamma \\ \vdots & \leadsto & \vdots \\ M & & N\end{array} \quad if\ \Gamma \vdash M \equiv N$$

Example

```
->Show.
1 subgoal
  (inclus A P P)
  ============================
    P : (Set A)
    A : Type



->Change (x:A)(P x)->(P x).
1 subgoal
  (x:A)(P x)->(P x)
  ============================
    P : (Set A)
    A : Type
```

Another basic tactic associated to this equality rule is the `unfold` tactic, which unfolds a given name in the goal.

```
->Undo.
1 subgoal
  (inclus A P P)
  ============================
    P : (Set A)
    A : Type

->Unfold inclus.
1 subgoal
  (x:A)(in A x P)->(in A x P)
  ============================
    P : (Set A)
```

```
A : Type
```

Notice the Undo command, that is "undoing" what has done the previous tactic.

3.5 The *exact* function

When we want to give explicitly the proof of one of the subgoals, we use the function `exact`

```
#exact;;
<fun> : (num -> command -> prooftree -> prooftree)
```

As we can see from its type, the function `exact` ***is not a tactic.*** It has indeed a global action on the prooftree, in opposition with the application of tactics that act only on one chosen subgoal.

3.6 Some tacticals

Here follow constructors of tactics (tacticals), that are CAML function for building more elaborate tactics.

tac1 `THEN` *tac2*:
This applies the *tac2* tactic to all the new goals generated by the *tac1* tactic.

tac1 `ORELSE` *tac2*:
This applies the *tac2* tactic , if , and only if, the *tac1* tactic fails.

tac `NEXT` *fun*:
It applies the *tac* tactic, and then the function *fun*, of type `prooftree -> prooftree` on the resulting prooftree (it is useful for instance in combination with the function `exact`).

`TRY` *tac*:
This applies the *tac* tactic and if this one fails, does nothing instead of generating an error.

`FIRST [`tac_1;...;tac_n`]`:
This applies the first tactic of the list which does not fail.

`REPEAT` *tac*:
This repeats the *tac* tactic as long as it does not fail. For instance we can define the following, that keep doing introduction.

```
let intros = REPEAT intro ;;
Value intros = <fun> : tactic
```

`AT_LEAST_ONE` *tac*:

This repeats the *tac* tactic at least one time and keep repeating it as long as it does not fail.

`IDTAC`:

This is the identity tactic: it sends back a prooftree with only a gap which represent the initial goal.

`FAILTAC`:

This is the tactic which always fails.

`COMPLETE` *tac*:

It applies *tac* only if *tac* solves completely the goal, and fails otherwise.

3.7 Some more elaborate tactics

Some useful tactics, not definable with the primitive ones, are provided.

3.7.1 The *pattern* tactic

For dealing with proofs by induction with a first-order matching, it is necessary to change a goal of the shape $\phi[a]$ to $(\lambda x.\phi(x)\ a)$. It may be cumbersome to use the tactic `Change`. A special tactic `Pattern` has thus been written for that. This tactic is useful for doing proofs by induction. Here is an example in the beginning of Peano arithmetic (which is here axiomatised inside second-order logic; notice that we have a finite axiomatisation since we can quantify over predicates).

```
Section Peano.
Type N.
Variable O:N.
Variable S:N->N.
Variable Eq:N->N->Prop.

Hypothesis trans:(x,y,z:N)(Eq x y)->(Eq y z)->(Eq x z).
Hypothesis sym:(x,y:N)(Eq x y)->(Eq y x).
Hypothesis Eq_S:(x,y:N)(Eq x y)->(Eq (S x) (S y)).

Hypothesis induction:(P:N->Prop)(P O)->((x:N)(P x)->(P (S x)))->(x:N)(P x).

Variable add:N->N->N.

Hypothesis add1:(x:N)(Eq (add x O) x).
Hypothesis add2:(x,y:N)(Eq (add x (S y)) (S (add x y))).
```

Here is a short session for a proof that uses induction.

```
->Goal (x:N)(Eq (add 0 x) x).
  (x:N)(Eq (add 0 x) x)

->Intro.
1 subgoal
  (Eq (add 0 x) x)
  ============================
    x : N

->Pattern x.
1 subgoal
  ([L:N](Eq (add 0 L) L) x)
  ============================
    x : N
```

We can then conclude the proof.

```
Resolve induction.
Resolve add1.
Intros.
Do res_exact trans (S (add 0 x0)).
Resolve add2.
Resolve Eq_S.
Assumption.
Save add3.
```

The command Do will be explained later (it allows the user to call CAML functions in the vernacular mode), and **res_exact** is a tactic programmed in CAML, that does a resolution and use the further arguments to solve immediatly the subgoals.

3.7.2 The *reduct* tactic

reduct: this is a variation of the change tactic. It reduces in complete head-normal form the goal.

4 The synthesis toplevel

4.1 The manipulation of the environment

Here are the commands of the synthesis machine. Some of them display the state of the proof-tree or more exactly display the list of the propositions which are yet to be proved with their local context. In case there is no proposition left to prove, it is displayed `goal proved`.

`Goal` *com* :

This command initializes the tree with a gap to which is associated the proposition and its local context. Then it displays the new state of the tree.

`Save` *thname* :

This command adds the theorem that has just been proved to the environment of the constructive machine, after checking that the proof is well-typed and of the wanted typed.

In opposition with system like LCF, the tactics may produce a wrong proof, that is a complete prooftree which is ill-typed or not of the right type. However, since the `Save` command goes through the type checking of the constructive engine, it is not possible to build an invalid environment.

`Show_proof`:

Displays the proof of the theorem that has just been proved.

It is possible (by cut and paste) to evaluate this proof via the vernacular `Eval` command. This may be useful for debugging a tactic.

`Undo` :

There is a stack of the last states of the proof-tree. This command allows to go back to the last state of the proof-tree. There is no display of this last state with this command. It is very useful when we have taken a bad direction and want to go back.

`Show` :

Displays the incomplete leaves of the current proof-tree.

Are available directly from the toplevel the basic tactics, and the `exact` function.

`Assumption`

Tries to solve the current goal with one local hypothesis.

`Exact` *com*

Tries to solve the current goal with the given proof *com*.

`Resolve` *com*

Tries to do a resolution against the head-normal form of the type of *com*.

`Intro`

Apply the `intro`tactic.

For calling the tactics programmed in CAML from the synthesis toplevel, we have provided two operators

`Do` *tac* $com_1 \ldots com_p$

If we have a function `tac` of type `command list-> tactic` then `Do` will execute the tactic `tac` $[com_1; \ldots; com_p]$.

`By` *tac* $name_1 \ldots name_p$

If we have a function `tac` of type `string list-> tactic` then `By` will execute the tactic `tac` $[name_1; \ldots; name_p]$.

For instance, here is a function that reduces the goal in head-normal form and then introduces the hypothesis named by the user:

```
let rec intros_with = function
 [] -> IDTAC
|x::l -> intro_with x THEN intros_with l;;

let hyps l = red THEN intros_with l;;
```

we have then.

```
Definition Nat.
Body (X:Prop)X->(X->X)->X.

->Goal Nat.
->By hyps X a f.

->1 subgoal
  X
  ============================
    f : X->X
    a : X
    X : Prop
```

`Abort`

Finally, if we do not succeed in proving a theorem, you can remove the goal from the stack of goals by the command `Abort`.

5 An example of added tactic : the *trivial* tactic

There are tactics that we would like to be applied automatically (for instance `intro` and `assumption`). We would like also the system to solve by itself some goals that seem trivial.

As an example of the use of CAML for extending our collection of tactics, we show how to program in CAML a tactic **trivial**. This tactic uses a list of tactics which have to be automatically applied as long as possible. After each application of a tactic of the trivial tactics list, a comment is displayed to describe what happens. Notice that this tactics is in some sense dynamics: it becomes more and more complete when the trivial tactics list increases.

There are commands to manipulate the list of trivial tactics:

`add_triv` *tac comment* :

This adds a new tactic to the list of trivial tactics. The comment has to explain what this tactic does. Usually it is the name of the tactic.

`print_triv` () :

This command displays the list of what the trivial tactics do.

`rm_triv` tac :

This command removes the *tac* tactic from the list of trivial tactics.

Here is the CAML code which defines the **trivial** tactic :

```
(* An ORELSE which take comments *)
let (ORELSE_COMMENT : (tactic * string) * tactic -> tactic)
                                            ((T1,comm),T2) g =
try let pf = T1 g in (message comm;print_newline();pf)
with UserError _ -> T2 g ;;

infix "ORELSE_COMMENT" ;;

let rec trivial_list = function
    []     -> FAILTAC
  | (t,c)::l -> (t,c) ORELSE_COMMENT (trivial_list l);;

(* The pointer to the list of trivial tactics *)
let triv = ref([]:(tactic * string) list);;

let trivial g = REPEAT (trivial_list (!triv)) g ;;

(* The commands which manipulate the !triv list *)
let add_triv (t,c) = triv := (t,c)::(!triv);() ;;
```

```
let rm_triv name = triv := delrec !triv;() where rec delrec = function
      []  -> message(name^" is not currently a trivial tactic");[]
  |  (t,s)::rest -> if s=name then rest else (t,s)::(delrec rest) ;;
```

Here is an example of use of `trivial` (we use in this example the command `Inductive`, that has been explained in the paper on the vernacular).

```
Inductive Definition And [A,B:Prop]:Prop =
And_intro : A->B->(And A B).

Inductive Definition implies [A,B:Prop]:Prop =
implies_intro : (A->B)->(implies A B).

Inductive Definition equiv [A,B:Prop]:Prop =
equiv_intro : (implies A B)->(implies B A)->(equiv A B).

Section Base.

Variables A,B,C:Prop.

->Goal (implies A (implies B C))->(implies (And A B) C).
->Intro.

1 subgoal
  (implies (And A B) C)
  ============================
    H : (implies A (implies B C))

->Resolve implies_intro.

1 subgoal
  (And A B)->C
  ============================
    H : (implies A (implies B C))

->Intro.

1 subgoal
```

```
  C
  ============================
    H0 : (And A B)
    H : (implies A (implies B C))

->Resolve H0.

1 subgoal
  A->B->C
  ============================
    H0 : (And A B)
    H : (implies A (implies B C))

->Resolve H.

1 subgoal
  (A->(implies B C))->A->B->C
  ============================
    H0 : (And A B)
    H : (implies A (implies B C))

->Intros.

1 subgoal
  C
  ============================
    H3 : B
    H2 : A
    H1 : A->(implies B C)
    H0 : (And A B)
    H : (implies A (implies B C))

->Resolve (H1 H2).

1 subgoal
  (B->C)->C
  ============================
    H3 : B
    H2 : A
    H1 : A->(implies B C)
```

```
    H0 : (And A B)
    H : (implies A (implies B C))

->Intros.

1 subgoal
  C
  ============================
    H4 : B->C
    H3 : B
    H2 : A
    H1 : A->(implies B C)
    H0 : (And A B)
    H : (implies A (implies B C))

->Exact (H4 H3).

goal proved

->Save And_elim.

And_elim is defined

->End Base.

And_elim :
(A:Prop)(B:Prop)(C:Prop)(implies A (implies B C))->(implies (And A B) C)

->Drop.

add_triv (resolve<<And_intro>>,"And_intro");;
add_triv (resolve<<implies_intro>>,"implies_intro");;
add_triv (resolve<<equiv_intro>>,"equiv_intro");;
add_triv (resolve<<And_elim>>,"And_elim");;
add_triv (intro,"introduction");;
add_triv (assumption,"assumption");;

go();;

Goal (A:Prop)(equiv A A).
```

```
By trivial.
Save idem_equiv.

Goal (A,B:Prop)(equiv (And A B) (And B A)).
By trivial.
Save abel_And.

Goal (A,B,C:Prop)(equiv (And A (And B C)) (And (And A B) C)).
By trivial.
Save assoc_And.
```

A short user's guide

for the Calculus of Constructions

Version 4.10

Gérard Huet

INRIA

1 Getting the system to run

In order to run the system, you need CAML version 2.6. Send requests for CAML to chenetier@ilog.fr.

Once you have CAML installed, you may run the system by calling the command `constr` from the constructions installation directory. You should get, after a few seconds, the banner:
`Calcul des Constructions Version 4.10`
and then the CAML prompt `#`.

If this does not work, check the contents of the command file `constr`, which should read:
`caml 35 -r constr.core`
The trouble may come from `caml` not being known. Make the proper PATH adjustment, abbrev declaration, or link. The trouble may be an inconsistency of core images. For instance, the image `constr.core` is not loadable from the old CAML version 2.5. In this case, if you do have CAML version 2.6, you may try to reconstruct the core image by running the command `makeconstr`. Even if you don't, and if you are trying to execute on a SUN 3, you ought to be able to run the core image `constr.core` as an executable directly. However, this will give you only a degraded system, since it will have very little memory, and thus will spend its time garbage-collecting, and furthermore it may not be able to auto-load crucial CAML libraries.

2 Navigating in the system

Initially, you are in CAML's top-level, indicated by its prompt sign `#`. You may in this state execute any CAML top-level phrase. For instance, if you want to run the system from an Emacs shell window, you want to execute initially `echo false;;`.

From CAML's top level, you may exit permanently in a graceful manner, by typing in `quit();;`, followed by RETURN, or forcibly by sending the QUIT signal (usually bound to key CTRL \). You may interrupt with the INT signal (usually bound to key CTRL C). You may escape to the surrounding shell by sending the

STOP signal (usually bound to key CTRL Z). In this last case, you may reenter your CAML session with the shell command `fg`.

In CAML's top level, you may load tactic files, using the standard commands `load` and `compile`. You may also operate directly the Constructive Engine in a step-by-step fashion, but this is not the usual mode of operation, which is to give Vernacular commands. Vernacular commands may be loaded from a file, say `th1.v`, by the command `V "th1"`. This is the standard way to initialize a mathematical theory, by loading in the vernacular file which defines its vocabulary, axioms, definitions, theorems.

Such vernacular commands may also be executed interactively, by entering the main loop of the system, by the CAML command `go();;`. You are now in the vernacular's top-level loop, indicated by the prompt `->`. All vernacular commands are terminated with a period. A basic manual of how the Constructions Vernacular is given in the technical note "The Vernacular Syllabus", by Gilles Dowek. The sub-language which concerns inductively defined notions is explained in the technical note "Inductive Definitions in the Calculus of Constructions", by Christine Paulin-Mohring. You come back to CAML from the vernacular loop with the command `Drop`.

This basic vernacular is a higher-level notation that compiles mathematical definitions and proofs into operations of the Constructive Engine. This engine, a theorem-checker for the calculus, is described in the article "The Constructive Engine", by Gérard Huet.

After a little practice with the use of the basic vernacular, the user may attempt to use the Tactics Theorem Prover. This is a goal-directed inference engine, in the spirit of Prolog, but with a proof-search mechanism driven by tactics in the spirit of the LCF proof assistant. Basic tactics are predefined, and the user may extend this initial set of tactics by writing his own tactics and tacticals in CAML. In order to understand this facility, read the note "The Tactics Theorem Prover", by Thierry Coquand.

The theorem prover is entered from the vernacular loop by the command `Goal`, which sets the initial goal you are trying to prove. Most tactics are accessible from the vernacular loop, with their own syntax. More complex tactics may be defined and executed from CAML. It is always possible to escape to CAML from the vernacular by typing in: `#(... CAML top-level phrase ...)`.

When the proof is completed, with the tactics assistance, the theorem may be entered in the current context under the name `name` by the vernacular command `Save name`. You may then go to the next proof, etc. You may backtrack from dead-ends using the command `Undo`, and you may completely abort the current search for a proof with the command `Abort`.

It is possible to save the state of a proof using a special description language called Ppl (portable proof language), but this facility is very crude and undocumented at present.

A small library of examples

Calculus of Constructions

Version 4.10

```
(***************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3    *)
(***************************************************************************)
(*                                                                         *)
(*           Prelude :  Logical connectives, quantifiers, equality        *)
(*                                                                         *)
(***************************************************************************)

Chapter Prelude.

Section Logic.

Inductive Proposition T = I : T.

Syntax I "<_>Id".

Section Negation.

(* Absurdity *)

Definition void (C:Prop)C.
 Syntax void "{}".

(* Negation *)

Definition not [A:Prop]A->{}.
 Syntax not "~_".

End Negation.

Section Conjunction.

(* Pairing / Introduction *)

 Syntax and "_/\_".

Inductive Connective and [A,B:Prop] = conj : A->B->(A/\B).

Syntax conj "<_,_>(_,_)".

Subsection Projections.

Propositions A,B.
Hypothesis x:A/\B.

Theorem proj1.
Statement A.
Proof (x A [y:A][z:B]y).

Theorem proj2.
Statement B.
Proof (x B [y:A][z:B]z).

End Projections.

Syntax proj1 "<_,_>Fst(_)".
Syntax proj2 "<_,_>Snd(_)".

End Conjunction.

Section Disjunction.

Syntax or "_\/_".
Inductive Connective or [A,B:Prop]
    = or_introl : A -> (A\/B) | or_intror : B -> (A\/B).

End Disjunction.

Section universal_quantification.

Definition all.
   Type A.
   Variable P : A->Prop.
Body (x:A)(P x).

Syntax all "<_>All(_)".

Type A.
Variable P : A->Prop.

Theorem inst.
Statement   (x:A)<A>All(P)->(P x).
Proof      [x:A][h:<A>All(P)](h x).

Theorem gen.
Statement (B:Prop)(f:(y:A)B->(P y))B-><A>All(P).
Proof [B:Prop][f:(y:A)B->(P y)][y:B][z:A](f z y).

End universal_quantification.

Section existential_quantification.

Syntax ex "<_>Ex(_)".
```

```
Inductive Connective
          ex [A:Type;P:A->Prop] = ex_intro : (x:A)(P x)->(<A>Ex(P)).

Syntax ex2 "<_>Ex2(_,_)".

Inductive Connective ex2 [A:Type;P,Q:A->Prop]
    = ex_intro2 : (x:A)(P x)->(Q x)->(<A>Ex2(P,Q)).

End existential_quantification.

Section equality.

(* Leibniz equality : [A:Type][x,y:A] (P:A->Prop)(P x)->(P y) *)

Syntax eqt "<_>_==_".
Inductive Definition eqt [A:Type;x:A] : A -> Prop
                      = refl_eqt : <A>x==x.

End equality.

End Logic.

(*****************************************************************************)
(*                                                                           *)
(*              Basic Logic with Spec and Data                               *)
(*                                                                           *)
(*****************************************************************************)

(* Product *)

Syntax prod  "_&_".
Inductive Definition prod [A,B:Spec] : Spec = pair : A->B->(A&B).

Section pair_definition.
Syntax pair "<_,_>(_,_)".

  Variables A,B:Spec.
  Variable x:A&B.

Theorem fst.
Statement A.
Proof (x A [y:A][z:B]y).

Theorem snd.
Statement B.
Proof (x B [y:A][z:B]z).

End pair_definition.

Syntax fst "<_,_>Fst(_)".
Syntax snd "<_,_>Snd(_)".

(* Disjunctions *)

Syntax sumbool "{_}+{_}".
Inductive Definition sumbool [A,B:Prop] : Spec
    = left : A ->({A}+{B}) | right : B->({A}+{B}).

Syntax sumor "_+{_}".
Inductive Definition sumor [A:Spec;B:Prop] : Spec
    = inleft : A -> (A+{B}) | inright : B -> (A+{B}).

Hypothesis SumOr : (A,B:Prop)({A}+{B})->(A\/B).

Syntax sum "_+_".
Inductive Definition sum [A,B:Spec] : Spec
    = inl : A -> (A+B) | inr : B -> (A+B).

(* Programming Language *)

(* Product of Datas *)

Syntax PROD  "_*_".
Inductive Definition PROD [A,B:Data] : Data = PAIR : A->B->(A*B).
Syntax PAIR "<_,_><_,_>".

Section programming.
   Variables A,B:Data.
   Variable x:A*B.
Theorem FST.
Statement A.
Proof (x A [y:A][z:B]y).

Theorem SND.
Statement B.
Proof (x B [y:A][z:B]z).
End programming.

Syntax FST "<_,_>Fst<_>".
Syntax SND "<_,_>Snd<_>".

(* First-order connectors *)

Definition pi.
```

```
Body [A:Data][P:A->Prop](x:A)(P x).
Syntax pi "<_>Pi(_)".

Syntax sig "<_>Sig(_)".
Inductive Definition sig [A:Data;P:A->Prop] : Spec
    = exist : (x:A)(P x)->(<A>Sig(P)).

Syntax exi "<_>Exi(_)".
Inductive Definition exi [A:Data;P:A->Prop] : Prop
    = exi_intro : (x:A)(P x)->(<A>Exi(P)).

Syntax sig2 "<_>Sig2(_,_)".
Inductive Definition sig2 [A:Data;P,Q:A->Prop] : Spec
    = exist2 : (x:A)(P x)->(Q x)->(<A>Sig2(P,Q)).

Syntax exi2 "<_>Exi2(_,_)".
Inductive Definition exi2 [A:Data;P,Q:A->Prop] : Prop
    = exi_intro2 : (x:A)(P x)->(Q x)->(<A>Exi2(P,Q)).

Hypothesis SigExi2 : (A:Data)(P,Q:A->Prop)<A>Sig2(P,Q)-><A>Exi2(P,Q).

(* Equality *)

Syntax eqd "<_>_=_".
Inductive Definition eqd [A:Data;x:A] : A->Prop
    = refl_equal : <A>x=x.

End Prelude.
```

```
(*******************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3      *)
(*******************************************************************************)
(*                                                                             *)
(*            Prelude :  Logical connectives, quantifiers, equality            *)
(*                                                                             *)
(*******************************************************************************)

Chapter Prelude_lemmas.

Goal (A:Prop)(C:Prop)A->(~A)->C.
By (intros THEN elim_unfold_last THEN assumption).
Save absurd.

Section Equality_is_a_congruence.
Types A,B.
Variable  f : A->B.
Variables x,y,z : A.

 Variable x,y,z : A.

 Goal (<A>x==y) -> <A>y==x.
 By (intros THEN elim_last).
 Resolve refl_eqt.
 Save sym_eqt.

 Goal (<A>x==y) -> (<A>y==z) -> <A>x==z.
 By (intros THEN elim_last THEN assumption).
 Save trans_eqt.

 Goal (<A>x==y)-><B>(f x)==(f y).
 By (intros THEN elim_last).
 Resolve refl_eqt.
 Save  congr_eqt.

End Equality_is_a_congruence.

Axiom except : (C:Spec){}->C.

Inductive Definition unit : Data = tt : unit.

Inductive Definition bool : Data = true : bool | false : bool.

Inductive Definition nat : Data = O : nat | S : nat->nat.

Section equality.
 Variable A,B : Data.
 Variable f   : A->B.
 Variable x,y,z : A.

 Goal (<A>x=y) -> <A>y=x.
 By (intros THEN elim_last).
 Resolve refl_equal.
 Save sym_equal.

 Goal (<A>x=y) -> (<A>y=z) -> <A>x=z.
 By (intros THEN elim_last THEN assumption).
 Save trans_equal.

 Goal (<A>x=y)-><B>(f x)=(f y).
 By (intros THEN elim_last).
 Resolve refl_equal.
 Save f_equal.
End equality.

Axiom eq_spec : (A:Data)(a,b:A)(<A>a=b)->(P:A->Spec)(P a)->(P b).

Section Properties_of_Relations.
 Variable A                : Data.
 Variable R                : A->A->Prop.

     Definition refl.
     Body (x:A)(R x x).
     Definition trans.
     Body (x,y,z:A)(R x y) -> (R y z) ->(R x z).
     Definition sym.
     Body (x,y:A)(R x y) -> (R y x).
     Definition equiv.
     Body refl /\ trans /\ sym.

End Properties_of_Relations.

End Prelude_lemmas.
```

```
(*************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3  *)
(*************************************************************************)
(*                                                                       *)
(*                          Tarski's Theorem                             *)
(*                                                                       *)
(*************************************************************************)

Parameter A:Type.

Parameter R:A->A->Prop.

Parameter Eq:A->A->Prop.

Axiom Assym.
Assumes (x:A)(y:A)((R x y) -> (R y x) -> (Eq x y)).

Axiom Trans.
Assumes (x:A)(y:A)(z:A)((R x y) -> (R y z) -> (R x z)).

Parameter f:A->A.

Axiom Incr.
Assumes (x:A)(y:A)((R x y) -> (R (f x) (f y))).

Definition Lub.
Body [m:A][S:A->Prop](and ((x:A) ((S x) -> (R x m)))
                          ((y:A)  (((x:A) ((S x)->(R x y))) -> (R m y)))).

Axiom Complete.
Assumes (S:A->Prop)(<A> Ex ([x:A] (Lub x S))).

Theorem Tarski.

Statement <A> Ex ([x:A](Eq x (f x))).

   Local Under.
   Body [x:A](R x (f x)).

   Remark Exist_lub_under.
   Statement <A> Ex ([m:A] (Lub m Under)).
   Proof (Complete Under).

   Remark Tarski1.

   Statement ((M:A) ((Lub M Under) -> <A> Ex  ([m:A] (Eq m (f m))))).

      Variable M:A.

      Hypothesis LeastUp.
      Assumes (Lub M Under).

      Remark Up.
      Statement (x:A) ((R x (f x)) -> (R x M)).
      Proof (proj1 ((x:A) ((R x (f x)) -> (R x M)))
                   ((x:A)(((y:A)(R y (f y))->(R y x))->(R M x)))
                   LeastUp).

      Remark Least.
      Statement (x:A)(((y:A)(R y (f y))->(R y x))->(R M x)).
      Proof (proj2 ((x:A) ((R x (f x)) -> (R x M)))
                   ((x:A)(((y:A)(R y (f y))->(R y x))->(R M x)))
                   LeastUp).


      Remark One.

      Statement (y:A)(Under y)->(R y (f M)).

         Variable y:A.

         Hypothesis v.
         Assumes (Under y).

         Remark T.
         Statement (R y M).
         Proof (Up y v).

         Remark T'.
         Statement (R (f y) (f M)).
         Proof (Incr y M T).

      Proof (Trans y (f y) (f M) v T').

      Remark Two.
      Statement (R M (f M)).
      Proof (Least (f M) One).

      Remark Three.
      Statement (R (f M) (f (f M))).
      Proof (Incr M (f M) Two).

      Remark Four.
      Statement (R (f M) M).
```

```
        Proof (Up (f M) Three).

        Remark Five.
        Statement (Eq M (f M)).
        Proof (Assym M (f M) Two Four).

    Proof (ex_intro A ([m:A] (Eq m (f m))) M Five).

Proof (Exist_lub_under (<A> Ex ([x:A](Eq x (f x))))  Tarski1).
```

```
(**************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3   *)
(**************************************************************************)
(*                                                                        *)
(*           Relation R on type A - Reflexive-transitive closure R*       *)
(*                                                                        *)
(**************************************************************************)

(* Properties of a binary relation R on type A *)

  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][h:(P x x)][h':(u:A)(R x u)->(Rstar u x)->(P x x)]h.

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)))).
```

```
(*****************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3      *)
(*****************************************************************************)
(*                                                                           *)
(*                      Newman's Lemma                                       *)
(*                                                                           *)
(*                      (uses Relations)                                     *)
(*                                                                           *)
(*****************************************************************************)

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

Theorem coherence_intro.
Statement  (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.
```

```
(******************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3       *)
(******************************************************************************)
(*                                                                            *)
(*                      Sch_Set.v                                             *)
(*                                                                            *)
(*       Embedding of elementary notion of the set theory in the              *)
(*       Calculus of Constructions in order to prove the                      *)
(*       Theorem of Schroeder-Bernstein.                                      *)
(*                                                                            *)
(*       Codage de base de la theorie des ensembles dans le but de            *)
(*       demontrer le theoreme de Schoeder-Bernstein                          *)
(*                                                                            *)
(*                     Hugo Herbelin                                          *)
(*                                                                            *)
(******************************************************************************)
(* #use "CPtac";;                                                             *)


(******************************************************************************)
(*  The objects of the theory of sets (individuals and sets) are              *)
(* considered as elements of universes. This is said in the Cacul with        *)
(* taking universes as terms of type Type and individiduals and sets as       *)
(* terms of type a univers.                                                   *)
(*                                                                            *)
(* Les objets ensemblistes (individuels et ensembles sont consideres          *)
(* comme appartenant a des univers. On traduit cela dans le Calcul en         *)
(* considerant les univers comme des termes de type Type et les objets        *)
(* ensemblistes comme des termes de type un univers.                          *)

Section Base_des_ens.

Variables U:Type.


(******************************************************************************)
(* Sets over individuels of the univers U are seen as their charac-           *)
(* teristic function.                                                         *)
(* Notion de base : l'ensemble vu comme sa fonction caracteristique           *)

Definition Set U->Prop.



(******************************************************************************)
(* Some fundamental notions and their properties                              *)
(* Quelques notions de base et leurs proprietes                               *)


Section Inclusion.

Definition inclus [A,B:Set](x:U)(A x)->(B x).

Variables A,B,C:Set.

Theorem refl_inclus (inclus A A)
 Proof [x:U]<(A x)>Id.

Theorem trans_inclus (inclus A B)->(inclus B C)->(inclus A C)
 Proof [h1:(inclus A B)][h2:(inclus B C)][x:U][p:(A x)]
      (h2 x (h1 x p)).

End Inclusion.


(* Section Difference.*)


Global diff:Set->Set->Set = [A,B:Set][x:U](A x)/\(~(B x)).


(*
Theorem diff_culbute (A,B,B':Set)(inclus B' B)->(inclus (diff A B) (diff A B'))
*)
Goal (A,B,B':Set)
       (inclus B' B)->(inclus (diff A B) (diff A B')).
By (intros THEN red THEN intros THEN red).
Resolve conj.
Resolve H0.
Intros.
Assumption.
By (red THEN intro).
Resolve H0.
Intros.
Resolve H3.
Resolve H.
Assumption.
Save diff_culbute.

End Base_des_ens.
```

```
(* La somme        *)

Section Somme.

Variable U:Type.

Global somme:(Set (Set U))->(Set U) =
     [D: (Set (Set U))][x:U](ex2 (Set U) D [B:(Set U)](B x)).


(*
Theorem somme_inclus1
          ((U:Type)(D:(Set (Set U)))(A:(Set U))
          ((C:(Set U))(D C)->(inclus U C A))
                ->(inclus U (somme U D) A))
*)
Goal (D:(Set (Set U)))(A:(Set U))
          ((C:(Set U))(D C)->(inclus U C A))
          ->(inclus U (somme D) A).
By (intros THEN red THEN intros).
Resolve H0.
Intros.
Exact (H x0 H1 x H2).
Save somme_inclus1.

(*
Theorem somme_inclus2
          ((U:Type)(D:(Set (Set U)))(A:(Set U))
          (D A)->(inclus U A (somme U D))
*)
Goal (D:(Set (Set U)))(A:(Set U))(D A)->(inclus U A (somme D)).
By (intros THEN red THEN intros THEN red).
Do resolve_with (ex_intro2 A).
Assumption.
Assumption.
Save somme_inclus2.

End Somme.

(***************************************************************************)
(*                              Relations                                  *)
(***************************************************************************)
(* The relation : seen as a characteristic function over a product of     *)
(* universes                                                               *)
(* L'objet relation : vu comme une fonction caracteristique                *)
(* sur un produit d'univers                                                *)

Definition Relation [U,U':Type]U->U'->Prop.


(***************************************************************************)
(* Characterization of a relation over two sets                           *)
(* Caracterisation d'une relation d'un ensemble donne ds un autre         *)

Inductive Definition
   Rel [U,U':Type;A:(Set U);B:(Set U');R:(Relation U U')] : Prop =
   Rel_intro : ((x:U)(y:U')(R x y)->(A x))
            ->((x:U)(y:U')(R x y)->(B y))->(Rel U U' A B R).


(***************************************************************************)
(*              Image of a set through a relation                          *)

Section Image.

Variables U,U':Type.

Inductive Definition Im [R:(Relation U U');A:(Set U);y:U'] : Prop  =
     Im_intro : (x:U)(R x y)->(A x)->(Im R A y).

(*
Theorem Im_stable_par_incl
        (R:(Relation U U'))(A,B:(Set U))
        (inclus U A B)->(inclus U' (Im U U' R A) (Im U U' R B)).
*)
Goal (R:(Relation U U'))(A1,A2:(Set U))
        (inclus U A1 A2)->(inclus U' (Im R A1) (Im R A2)).
By (intros THEN red THEN intros).
Resolve H0.
Intros.
Do resolve_with (Im_intro x0).
Assumption.
Resolve H.
Assumption.
Save Im_stable_par_incl.

End Image.


(***************************************************************************)
```

```
(*                              Fonctions                              *)

Section Fonctions.

Variables U,U':Type.
Variable A:(Set U).
Variable B:(Set U').
Variable f:(Relation U U').

Definition au_plus_une_im (x:U)(y,z:U')(f x y)->(f x z)->(eqt U' y z).

Definition au_moins_une_im ((x:U)(A x)->(ex U' (f x))).

Definition au_plus_un_ant (x,y:U)(z:U')(f x z)->(f y z)->(eqt U x y).

Definition au_moins_un_ant ((y:U')(B y)->(ex U [x:U](f x y))).


Inductive Definition fonction : Prop =          (* fun_in *)
   fonction_intro : (Rel U U' A B f)
                  ->au_plus_une_im
                  ->au_moins_une_im   ->fonction.

Inductive Definition surjection : Prop =        (* fun_on *)
   surjection_intro :   (Rel U U' A B f)
                  ->au_plus_une_im
                  ->au_moins_une_im
                  ->au_moins_un_ant   ->surjection.

Inductive Definition injection : Prop =         (* map_in *)
   injection_intro :   (Rel U U' A B f)
                  ->au_plus_une_im
                  ->au_moins_une_im
                  ->au_plus_un_ant  ->injection.

Inductive Definition bijection : Prop =         (* map_on *)
   bijection_intro : (Rel U U' A B f)
                  ->au_plus_une_im
                  ->au_moins_une_im
                  ->au_plus_un_ant
                  ->au_moins_un_ant  ->bijection.

End Fonctions.


(*************************************************************************)
(*        Equipollence  and relation "is of cardinal less than"         *)

Section Equipollence.

Variables U,U':Type.
Variable A:(Set U).
Variable B:(Set U').
Local Rela (Relation U U').

Inductive Definition equipollence : Prop =
    equipollence_intro :   (f:Rela)(bijection U U' A B f)->equipollence.

Inductive Definition inf_card : Prop =
    inf_card_intro : (f:Rela)(injection U U' A B f)->inf_card.

End Equipollence.
```

```
(****************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3     *)
(****************************************************************************)
(*      Theorem of Schroeder-Bernstein in the Calculus of Constructions     *)
(*                                                                          *)
(*      If A is of cardinal less than B and reciprocally then A and B       *)
(*        are equipollent                                                   *)
(*    Said in a different manner, if there is a map from A in B and         *)
(*      a map from B in A then there exists a map from A onto B.            *)
(*                                                                          *)
(*                                                                          *)
(*      Theoreme de Schroeder-Bernstein dans le Calcul des Constructions    *)
(*                          Hugo Herbelin                                   *)
(*                (d'apres une demonstration de Fraenkel)                   *)
(*                                                                          *)
(*   S'il existe une injection de A dans B et une injection de B dans A,    *)
(*       alors il existe une bijection de A vers B.                         *)
(*     En termes ensemblistes, si A est de cardinal inferieur a B et        *)
(*       reciproquement, alors A et B ont meme cardinal.                    *)
(*                                                                          *)
(****************************************************************************)
(* #use "CPtac" ; V "Sch_Set";;                                             *)


Section Schroeder_Bernstein.

(****************************************************************************)
(*  The axiom of excluded-middle is assumed                                 *)
(*  On suppose l'axiome du tiers_exclu                                      *)


Hypothesis tiers_exclu : (U:Type)(x:U)(A:U->Prop)(not (A x))\/(A x).


(*             Le theoreme d'introduction du tiers-exclu                    *)

Remark exclusion (U:Type)(x:U)(A:U->Prop)(P:Prop)
        ((~(A x))->P)->((A x)->P)->P
  Proof [U:Type][x:U][A:U->Prop][P:Prop][h1:(~(A x))->P][h2:(A x)->P]
        (tiers_exclu U x A P h1 h2).


(****************************************************************************)
(*  The context : A is a set of elements in the univers U and B a set       *)
(*  over the univers U'                                                     *)
(*  On introduit le contexte :                                              *)
(*       A est un ensemble d'elements pris dans l'univers U                 *)
(*       B est un ensemble d'elements pris dans l'univers U'                *)


Variables U,U':Type.

Local SU  (Set U).
Local SU' (Set U').

Variable A:SU.  (* A est un ensemble d'elements de l'univers U  *)
Variable B:SU'. (* B est un ensemble d'elements de l'univers U' *)


  Section Bijection.

  (**************************************************************************)
  (* On montre dans ce paragraphe que si f et g sont des injections resp    *)
  (* de A dans B et de B dans A alors on peut trouver un sous-ensemble J de *)
  (* A tq h qui est f sur J et g sur A\J est une bijection de A dans B      *)

  Variable f:(Relation U U').  (* f et g sont des relations *)
  Variable g:(Relation U' U).

  Hypothesis f_inj:(injection U U' A B f). (* f et g sont des injections *)
  Hypothesis g_inj:(injection U' U B A g).

  Local Imf (Im U U' f).
  Local Img (Im U' U g).


  (* Construction de J tq g(B\f(J))=A\J *)

    (* diff U A C designe la difference A\C          *)
    (* inclus U A C signifie que A est inclus dans C  *)

    Local F [C:SU](diff U A (Img (diff U' B (Imf C)))).


    Local D [C:SU](inclus U C (F C)).

    Local J (somme U D).
```

```
(*  On va montrer que J correspond a ce que l'on cherche *)

  (* J correspond au point fixe de Tarski pour F qui conserve les
         relations d'inclusion *)

  (* Lemma F est croissante *)

   Goal (C,C':SU).(inclus U C C')->(inclus U (F C) (F C')).
     By (intros THEN unfold"F").
     Resolve diff_culbute.
     By unfolds Img.
     Resolve Im_stable_par_incl.
     Resolve diff_culbute.
     By unfolds Imf.
     Resolve Im_stable_par_incl.
     Assumption.
   Save_remark F_croissante.

  (* On va montrer que F(J)=A\Img(B\Imf(J))=J *)

     (* D'abord l'inclusion dans un sens *)

       (*  Lemma J_dans_FJ (inclus U J (F J))   *)

       Goal (inclus U J (F J)).
         By unfolds J.
         Resolve somme_inclus1.
         Intros.
         Do (incomplet [3]) (trans_inclus (F C)).
         (* Que C est inclus dans (F C) *)
           Assumption.
         (* Que (F C) inclus dans (F (somme U D)) *)
           Resolve F_croissante.
           Resolve somme_inclus2.
           Assumption.
       Save_remark J_dans_FJ.

     (* Puis dans l'autre sens *)

       (*  Lemma FJ_dans_J (inclus U (F J) J)   *)

       Goal (inclus U (F J) J).
         By unfolds J.
         Resolve somme_inclus2.
         Red.
         Resolve F_croissante.
         Exact J_dans_FJ.
       Save_remark FJ_dans_J.


(* On montre que h qui est f sur J et g ailleurs est une bijection *)

  Inductive Definition h [x:U;y:U'] : Prop =
     hl_intro : (J x)->(f x y)->(h x y)
   | hr_intro : (diff U' B (Imf J) y)->(g y x)->(h x y).

(*  Theorem h_bij (bijection U U' A B h)      *)

Theorem h_bij.

Statement (bijection U U' A B h).


  (* h est de A dans B *)
  Goal (Rel U U' A B h).

      Resolve Rel_intro.
      (* h est sur A *)
      Intros.
      Resolve H.
      (* sur J : f est de A dans B *)
      Resolve f_inj.
      Intros.
      Resolve H0. (* H0 :f est Rel sur A et B *)
      Intros.
      Do resolve_with (H6 y).
      Assumption.

      (* sur A\J: g est de B dans A *)
      Resolve g_inj.
      Intros.
      Resolve H0. (* H0 :g est Rel sur B et A *)
      Intros.
      Resolve (H7 y).
      Assumption.

    (* h est sur B *)
      Intros.
      Resolve H.
      (* sur J : f est de A dans B *)
      Resolve f_inj.
```

```
    Intros.
    Resolve H0. (* H0 :f est Rel sur A et B *)
    Intros.
    Resolve (H7 x).
    Assumption.

    (* sur A\J: g est de B dans A *)
    Resolve g_inj.
    Intros.
    Resolve H0. (* H0 :g est Rel sur B et A *)
    Intros.
    Do resolve_with (H6 x).
    Assumption.

Save_remark h1.


(* h verifie au_plus_une_im *)
Goal (au_plus_une_im U U' h).

  By (red THEN intros).
  Resolve H.

   (* sur J *)
    Resolve H0.
     (* cas ou (h x y) et (h x z) se comporte comme f : correct *)
      Resolve f_inj.
      Intros.
      Do cut (x:U)(y,z:U')(f x y)->(f x z)-><U'>y==z.
      Do resolve_with (Hyp x).
      Assumption.
      Assumption.
      Assumption.

     (* Cas ou (h x y) se comporte comme f et
               (h x z) comme g : contradiction *)
      Intros.
      Do cut (diff U A (Img (diff U' B (Imf J))) x).
      Resolve Hyp.
      Intros.
      Do elim_unfold H6.
      Red.
      Do resolve_with (Im_intro z).
      Assumption.
      Assumption.
      Do cut (x:U)(J x)->(diff U A (Img (diff U' B (Imf J))) x).
      Resolve Hyp.
      Assumption.
      Exact J_dans_FJ.

   (* sur A\J *)
    Resolve H0.
     (* Cas ou (h x y) se comporte comme g et
               (h x z) comme f : contradiction *)
      Intros.
      Do cut (diff U A (Img (diff U' B (Imf J))) x).
      Resolve Hyp.
      Intros.
      Do elim_unfold H6.
      Red.
      Do resolve_with (Im_intro y).
      Assumption.
      Assumption.
      Exact (J_dans_FJ x H1).

     (* cas ou (h x y) et (h x z) se comporte comme g : correct *)
      Resolve g_inj.
      Intros.
      Do cut (y,z:U')(x:U)(g y x)->(g z x)-><U'>y==z.
      Do resolve_with (Hyp x).
      Assumption.
      Assumption.
      Assumption.
Save_remark h2.

(* h verifie au_moins_une_im *)
Goal (au_moins_une_im U U' A h).
  Red.
  Intros.
  Resolve (exclusion U x (Img (diff U' B (Imf J)))).

   (* sur J *)
  Intros.
     (* De f fonction, on deduit f verifie au_moins_une_im *)
    Resolve f_inj.
    Intros.
    Do elim_unfold (H3 x).
    Assumption.
    Intros.
    Do resolve_with (ex_intro x0).
    Resolve h1_intro.
    Do elim_unfold FJ_dans_J.
    By (red THEN red).
```

```
    Resolve conj.
    Assumption.
    Assumption.
    Assumption.

   (* sur A\J *)
   Intros.
     (* De f injective on deduit f verifie au_moins_une_im *)
    Resolve g_inj.
    Intros.
    Resolve H0.
    Intros.
    Do resolve_with (ex_intro x0).
    Resolve hr_intro.
    Assumption.
    Assumption.

Save_remark h3.


(* h verifie au_plus_un_ant *)
Goal (au_plus_un_ant U U' h).

  By (red THEN intros).
  Resolve H.

   (* sur J *)
    Resolve H0.
     (* cas ou (h x y) et (h x z) se comporte comme f : correct *)
      Resolve f_inj.
      Intros.
      Do cut (x,y:U)(z:U')(f x z)->(f y z)-><U>x==y.
      Do (incomplet [3]) (Hyp z).
      Assumption.
      Assumption.
      Assumption.

     (* Montrer qu'on ne peut avoir (f x z) et (g z y) avec x dans J et
         z hors de (Imf J) sans contradiction *)
      Intros.
      Do elim_unfold H1.
      Intros.
      Do elim_unfold H6.
      Red.
      Do resolve_with (Im_intro x).
      Assumption.
      Assumption.

   (* sur A\J *)
    Resolve H0.
     (* Montrer qu'on ne peut avoir (f y z) et (g z x) avec x dans J et
         z hors de (Imf J) sans contradiction *)
      Intros.
      Do elim_unfold H3.
      Intros.
      Do elim_unfold H6.
      Red.
      Do resolve_with (Im_intro y).
      Assumption.
      Assumption.
     (* De g fonction on deduit g verifie au_plus_une_im c'est-a-dire
                                              au_plus_un_ant pour h *)
      Resolve g_inj.
      Intros.
      Do cut (z:U')(x,y:U)(g z x)->(g z y)-><U>x==y.
      Resolve (Hyp z).
      Assumption.
      Assumption.
      Assumption.

Save_remark h4.


(* h verifie au_moins_un_ant *)
Goal (au_moins_un_ant U U' B h).

  Red.
  Intros.
  Resolve (exclusion U' y (Imf J)).

   (* sur A\J *)
    Intros.
     (* De g injective on deduit g verifie au_moins_une_im c'est-a-dire
                                              au_moins_un_ant pour h *)
      Resolve g_inj.
      Intros.
      Do elim_unfold (H3 y).
      Assumption.
      Intros.
      Do resolve_with (ex_intro x).
      Resolve hr_intro.
      Red.
      Resolve conj.
```

```
        Assumption.
        Assumption.
        Assumption.

    (* sur J *)
      Intros.
      (* De f injective on deduit f verifie au_moins_un_ant *)
        Resolve f_inj.
        Intros.
        Do elim_unfold H0.
        Intros.
        Do resolve_with (ex_intro x).
        Resolve h1_intro.
        Assumption.
        Assumption.

  Save_remark h5.


  Proof (bijection_intro U U' A B h h1 h2 h3 h4 h5).

 End Bijection.


(*    Le theoreme de Schroeder-Bernstein-Cantor      *)

Goal (inf_card U U' A B)->(inf_card U' U B A)->(equipollence U U' A B).

 By intro_with A_inf_B.
 By intro_with B_inf_A.
 Resolve A_inf_B.
 Intros.
 Resolve B_inf_A.
 Intros.
 Do resolve_with (equipollence_intro (h f f0)).
 Resolve h_bij.
 Assumption.
 Assumption.

Save Schroeder.


End Schroeder_Bernstein.

                           (* The end *)
```

```
(***************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3    *)
(***************************************************************************)
(*                                                                         *)
(*        Prelude :  Logical connectives, quantifiers, equality            *)
(*                                                                         *)
(***************************************************************************)

Chapter Prelude_lemmas.

Theorem absurd.
Statement (A:Prop)(C:Prop)A->(~A)->C.
Proof [A,C:Prop][h1:A][h2:~A](h2 h1 C).

Section Equality_is_a_congruence.
Types A,B.
Variable  f : A->B.
Variables x,y,z : A.

  Theorem sym_eqt        (<A>x==y) -> <A>y==x
  Proof                  [h:<A>x==y]
                         (h [u:A]<A>u==x (refl_eqt A x)).
  Theorem trans_eqt      (<A>x==y) -> (<A>y==z) -> <A>x==z
  Proof                  [h1:<A>x==y][h2:<A>y==z](h2 [u:A]<A>x==u h1).

  Theorem congr_eqt      (<A>x==y)-><B>(f x)==(f y)
  Proof                  [h:<A>x==y](h [u:A]<B>(f x)==(f u) (refl_eqt B (f x))).

End Equality_is_a_congruence.

Axiom except : (C:Spec){}->C.

Inductive Data unit = tt : unit.

Inductive Data bool = true : bool | false : bool.

Inductive Data nat  = O : nat | S : nat->nat.

Section equality.
 Variable A,B : Data.
 Variable f   : A->B.
 Variable x,y,z : A.

 Theorem sym_equal.
 Statement (<A>x=y) -> <A>y=x.
 Proof [h:<A>x=y]
       (h [u:A]<A>u=x (refl_equal A x)).

 Theorem trans_equal.
 Statement (<A>x=y) -> (<A>y=z) -> <A>x=z.
 Proof [h1:<A>x=y][h2:<A>y=z](h2 [u:A]<A>x=u h1).

 Theorem f_equal.
 Statement (<A>x=y)-><B>(f x)=(f y).
 Proof [h:<A>x=y](h [u:A]<B>(f x)=(f u) (refl_equal B (f x))).

End equality.

Axiom eq_spec : (A:Data)(a,b:A)(<A>a=b)->(P:A->Spec)(P a)->(P b).

Section Properties_of_Relations.
 Variable A          : Data.
 Variable R          : A->A->Prop.

     Definition refl.
     Body (x:A)(R x x).
     Definition trans.
     Body (x,y,z:A)(R x y) -> (R y z) ->(R x z).
     Definition sym.
     Body (x,y:A)(R x y) -> (R y x).
     Definition equiv.
     Body refl /\ trans /\ sym.

End Properties_of_Relations.

End Prelude_lemmas.
```

```
(*****************************************************************************)
(*   Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3      *)
(*****************************************************************************)
(*                                                                           *)
(*                       Formatting program                                  *)
(*                                                                           *)
(*                       Christine Paulin-Mohring                            *)
(*                                                                           *)
(*****************************************************************************)


    (*********************************************************************)
    (* A list of words separated by white characters (space or line feed) *)
    (* is given. The program produces a formated list, that is the same   *)
    (* list of words separated by just one line-feed or space in order to *)
    (* put the maximum number of words on the same line.                  *)
    (*********************************************************************)

(* V "Prel_Lem"; *)

Chapter Format.

(* Environment *)

(*****************************************************************************)
(* The non-white characters (letters) are indexed by a type Ind.            *)
(* The white spaces (separators) are                                        *)
(* indexed by the type bool (space = true and linefeed = false)             *)
(* The data structure is a type of lists with two constructors for cons of  *)
(* letters and cons of separators                                           *)
(*****************************************************************************)

 Parameter      Ind :   Data.
 Inductive Data l_ch = nil : l_ch
                     | consbl : bool->l_ch->l_ch
                     | consltr : Ind->l_ch->l_ch.

 Local conssp (consbl true).
 Local conslf (consbl false).

(* Induction on the type l_ch *)

 Axiom LCH_Ind     :
     (x:l_ch)(P:l_ch->Prop)
              (P nil)->((b:bool)(l:l_ch)(P l)->(P (consbl b l)))
              ->((i:Ind)(l:l_ch)(P l)->(P (consltr i l)))
              ->(P x).

 Axiom LCH_rec     :
     (x:l_ch)(P:l_ch->Spec)
              (P nil)->((b:bool)(l:l_ch)(P l)->(P (consbl b l)))
              ->((i:Ind)(l:l_ch)(P l)->(P (consltr i l)))
              ->(P x).

(* case definition *)

Section Casebl.

 Variable  A : Data.
 Variable  c  : l_ch.
 Variable  xnil : A.
 Variables xbl,xltr:A->A.
 Global    Casebl.
 Body      (c A xnil [b:bool]xbl [i:Ind]xltr) : A.

End Casebl.

(* Length function *)

 Define lgth.
 Body [l:l_ch](Casebl nat l O S S) : l_ch->nat.

(* Append function *)

 Define app.
 Body [l,m:l_ch](l l_ch m consbl consltr)  : l_ch->l_ch->l_ch.

 Local  appsp [l,m:l_ch](app l (conssp m)).

 Local  applf [l:l_ch][m:l_ch](app l (conslf m)).

(* Maximal allowed length of a line *)

 Parameter  max : nat.

(* Predicate that recognize a word : sequenz (eventually empty) of letters *)

Inductive Definition word : l_ch -> Prop
    = wordnil : (word nil)
    | wordltr : (j:Ind)(l:l_ch)(word l)->(word (consltr j l)).

(* Predicate that recognize a non-empty sequenz of separators *)

Inductive Definition space  :  l_ch->Prop
```

```
      =  space_bl     : (b:bool)(space (consbl b nil))
        |space_co_bl : (b:bool)(l:l_ch)(space l)->(space (consbl b l)).

 (* We give an axiomatisation of predicates inf and sup on integers.      *)
 (* We will cut  a line according these predicates.                      *)
 (* inf n means  n < max and sup n means n >= max                         *)

 Parameter inf, sup : nat->Prop.
 Axiom ax1      :    (n:nat)({(inf n)}+{(sup n)}).

 Local  inflg   [m:l_ch](inf (lgth m)).

 Local  suplg   [m:l_ch](sup (lgth m)).

 Local plus.
 Body [m:l_ch][n:nat](lgth m nat n S) : l_ch->nat->nat.

(* The words must be of length less than max. We define a predicate valide. *)
(* A proof of (valide l) gives a decomposition of the list of characters    *)
(* in words of length less than max, separated by spaces                    *)

Inductive Definition valide  :l_ch->Spec
    = vword : (m:l_ch)(word m)->(inflg m)->(valide m)
     |vapp  : (m1,m2,p:l_ch)(word m1)->(inflg m1)->(space m2)->(valide p)
            ->(valide (app m1 (app m2 p))).

(* The empty list is valide if O<max *)

 Hypothesis infmax : (inf O).

(* A list will be non-valid if it contains a too long word *)

Inductive Definition NV : l_ch->Prop
    = NVword :   (l:l_ch)(word l)->(suplg l)->(NV l)
    |NVapp1 :   (l,m:l_ch)(NV l)->(NV (app l m))
    |NVapp2 :   (l,m:l_ch)(NV l)->(NV (app m l)).

(* We show that every list satisfies valide or NV, this is an informative *)
(* proof (a preprocessing of the text.                                    *)

Inductive Definition val_or_no [l:l_ch] : Spec
    = inval : (valide l)->(val_or_no l)
     |inNV : (NV l) -> (val_or_no l).

(* To add a space to a valid list is easy *)

Remark valbl.
Statement  (b:bool)(l:l_ch)(valide l)->(valide (consbl b l)).
Proof [b:bool][l:l_ch][h:(valide l)]
        (vapp nil (consbl b nil) l wordnil infmax (space_bl b) h).

(* For adding a letter to a valid list, we need to test whether max *)
(* is overheaded.                                                   *)

Remark valltr.
Statement (i:Ind)(l:l_ch)(valide l)->(val_or_no (consltr i l)).
 Proof
     [i:Ind][l:l_ch][h:(valide l)]
     (valide_rec l h [m:l_ch](val_or_no (consltr i m))
        [m:l_ch][t1:(word m)][t2:(inflg m)]
                (ax1 (S (lgth m)) (val_or_no (consltr i m))
                          [h1:(inflg (consltr i m))]
                          (inval (consltr i m)
                                 (vword (consltr i m)(wordltr i m t1) h1))
                          [h2:(suplg (consltr i m))]
                          (inNV (consltr i m)
                                (NVword (consltr i m) (wordltr i m t1)
                                        h2)))
        [m1,m2,p:l_ch][t1:(word m1)][t2:(inflg m1)][t3:(space m2)]
               [f:(valide p)&(val_or_no (consltr i p))]
               (ax1 (S (lgth m1)) (val_or_no (consltr i (app m1 (app m2 p))))
                     [h11:(inflg (consltr i m1))]
                       (inval (consltr i (app m1 (app m2 p)))
                              (vapp (consltr i m1) m2 p (wordltr i m1 t1)
                                    h11 t3
                                (fst (valide p) (val_or_no (consltr i p)) f)))
                     [h12:(suplg (consltr i m1))]
                         (inNV (consltr i (app m1 (app m2 p)))
                               (NVapp1 (consltr i m1) (app m2 p)
                                    (NVword (consltr i m1)
                                            (wordltr i m1 t1) h12))))).

(* The complete proof is by induction on the structure of the list *)

Lemma preproc.
Statement (l:l_ch)((valide l)+{(NV l)}).
Proof  [l:l_ch]
     (LCH_rec l val_or_no
          (inval nil (vword nil wordnil infmax))
          ([b:bool][m:l_ch][h:(val_or_no m)]
           (h (val_or_no (consbl b m))
              ([h1:(valide m)](inval (consbl b m) (valbl b m h1)))
              ([h2:(NV m)]
               (inNV (consbl b m) (NVapp2 m (consbl b nil) h2)))))
```

```
          ([i:Ind][m:l_ch][h:(val_or_no m)]
           (h (val_or_no (consltr i m))
              (valltr i m)
              ([t:(NV m)]
               (inNV (consltr i m) (NVapp2 m (consltr i nil) t)))))).

(* We define the relation "two lists are equivalent", the intended meaning *)
(* is that they represent the same list of words                           *)

Section Equivalence.

 Inductive Definition Eq  : l_ch->l_ch->Prop
    = Eq_nil : (Eq nil nil)
    | Eq_bl_nil : (b:bool)(l:l_ch)(Eq l nil)->(Eq (consbl b l) nil)
    | Eq_co_bl  : (b,c:bool)(l,m:l_ch)(Eq l m)->(Eq (consbl b l) (consbl c m))
    | Eq_co_ltr : (i:Ind)(l,m:l_ch)(Eq l m)->(Eq (consltr i l) (consltr i m))
    | Eq_bl_bl  : (b,c:bool)(l,m:l_ch)
           (Eq l (consbl b m))->(Eq (consbl c l)(consbl b m))
    | Eq_tran : (l,m,n:l_ch)(Eq l m)->(Eq m n)->(Eq l n).

(* Properties of equivalence *)

(* Reflexivity *)

 Property Eq_re.
 Statement   (l:l_ch)(Eq l l).
 Proof     [l:l_ch](LCH_Ind l [u:l_ch](Eq u u) Eq_nil
                     [b:bool][m:l_ch](Eq_co_bl b b m m)
                     [i:Ind][m:l_ch](Eq_co_ltr i m m)).

(* Eq stability with respect to append *)

 Property Eq_app.
 Statement  (l,m,n:l_ch)(Eq m n)->(Eq (app l m) (app l n)).
 Proof [l,m,n:l_ch][h:(Eq m n)]
        (LCH_Ind l [u:l_ch](Eq (app u m) (app u n)) h
             [b:bool][u:l_ch](Eq_co_bl b b (app u m) (app u n))
             [i:Ind][u:l_ch](Eq_co_ltr i (app u m) (app u n))).

(* particular case n=nil, directly proved *)

 Property Eq_app_nil.
 Statement (l,m:l_ch)(Eq m nil)->(Eq (app l m) l).
 Proof  [l,m:l_ch][h:(Eq m nil)]
        (LCH_Ind l [u:l_ch](Eq (app u m) u) h
             [b:bool][u:l_ch](Eq_co_bl b b (app u m) u)
             [i:Ind][u:l_ch](Eq_co_ltr i (app u m) u)).

(* Adding more than one separator does not change anything *)
 Property Eq_space_bl.
 Variable b : bool.
 Variables l,m,n:l_ch.
 Statement (space l)->(Eq m (consbl b n))->(Eq (app l m) (consbl b n)).
 Proof  [h1:(space l)][h2:(Eq m (consbl b n))]
         (h1 [u:l_ch](Eq (app u m) (consbl b n))
             [c:bool](Eq_bl_bl b c m n h2)
             [c:bool][u:l_ch][t:(Eq (app u m) (consbl b n))]
             (Eq_bl_bl b c (app u m) n t)).

(* Adding separators at the beginning of the list is equivalent to nil *)
 Property Eq_space_nil.
 Statement (m:l_ch)(space m)->(Eq (app m nil) nil).
 Proof  [m:l_ch][h:(space m)]
        (h [u:l_ch](Eq (app u nil) nil)
           ([b:bool](Eq_bl_nil b nil Eq_nil))
           ([b:bool][u:l_ch][t:(Eq (app u nil) nil)]
            (Eq_bl_nil b (app u nil) t))).

(* Adding separators is equivalent to add a space *)
 Property Eq_space.
 Statement (l,m,n:l_ch)(space l)->(Eq m n)->(Eq (app l m) (conssp n)).
 Proof [l,m,n:l_ch][h1:(space l)][h2:(Eq m n)]
        (h1 [u:l_ch](Eq (app u m) (conssp n))
            ([c:bool](Eq_co_bl c true m n h2))
            ([c:bool][u:l_ch][t:(Eq (app u m) (conssp n))]
             (Eq_bl_bl true c (app u m) n t))).

End Equivalence.

(* Definition of the predicate formn : nat->l_ch->Prop. (formn n l) means that
 l is a formated list with n characters on its last line *)

Inductive Definition formn :nat->l_ch->Prop
    = fword : (m:l_ch)(word m)->(inflg m)->(formn (lgth m) m)
    | fwordinf : (p:nat)(m1:l_ch)(b:bool)(m2:l_ch)
             (inf (plus m1 p))->(word m1)->(formn p (consbl b m2))
              ->(formn (plus m1 p) (appsp m1 m2))
    | fwordsup : (p:nat)(m1:l_ch)(b:bool)(m2:l_ch)
         (sup (plus m1 p))->(word m1)->(inflg m1)->
         (formn p (consbl b m2)) ->(formn (lgth m1) (applf m1 m2))
    | fsp : (p:nat)(i:Ind)(m:l_ch)
            (formn p (consltr i m))->(formn (S p) (conssp (consltr i m))).

(* Program's specification for a list l : there exists n : nat and m : list *)
```

```
(* such that (formn n m) and (Eq l m)                                    *)

 Inductive Definition SigFormat [l:l_ch] : Spec
    = ExFormat : (n:nat)(m:l_ch)(formn n m)->(Eq l m)->(SigFormat l).

(* Two lemmas *)

Lemma Lem1.
 Variable b    :   bool.
 Variables p,m1,m2      :   l_ch.
 Variable n    :   nat.

 Hypothesis h1 :   (word m1).
 Hypothesis h2 :   (inflg m1).
 Hypothesis t1 :   (formn n (consbl b m2)).
 Hypothesis t2 :   (Eq p (consbl b m2)).

 Statement   (SigFormat (app m1 p)).
 Proof   (ax1 (plus m1 n) (SigFormat (app m1 p))
        ([f1:(inf (plus m1 n))]
        (ExFormat (app m1 p) (plus m1 n) (appsp m1 m2)
            (fwordinf n m1 b m2 f1 h1 t1)
            (Eq_app m1 p (conssp m2)
                (Eq_tran p (consbl b m2) (conssp m2) t2
                    (Eq_co_bl b true m2 m2 (Eq_re m2))))))
        ([f2:(sup (plus m1 n))]
        (ExFormat (app m1 p) (lgth m1) (applf m1 m2)
            (fwordsup n m1 b m2 f2 h1 h2 t1)
            (Eq_app m1 p (conslf m2)
                (Eq_tran p (consbl b m2) (conslf m2) t2
                    (Eq_co_bl b false m2 m2 (Eq_re m2))))))).


Lemma Lem2.

Local prop2  [m1,m2,p:l_ch][n:nat][u:l_ch]
    (formn n u)->(Eq p u)->(SigFormat (app m1 (app m2 p))).

 Variables     m1,m2,p :   l_ch.
 Hypothesis    h1  :   (word m1).
 Hypothesis    h2  :   (inflg m1).
 Hypothesis    h3  :   (space m2).

 Variable  n   :   nat.
 Variable  p0  :   l_ch.

 Statement (prop2 m1 m2 p n p0).
 Proof (LCH_rec p0 (prop2 m1 m2 p n)
            ([t1:(formn n nil)][t2:(Eq p nil)]
             (ExFormat (app m1 (app m2 p)) (lgth m1) m1
                 (fword m1 h1 h2)
                 (Eq_app_nil m1 (app m2 p)
                     (Eq_tran (app m2 p) (app m2 nil) nil
                         (Eq_app m2 p nil t2)
                         (Eq_space_nil m2 h3)))))
           ([b:bool][l:l_ch][h:(prop2 m1 m2 p n l)]
             [t1:(formn n (consbl b l))][t2:(Eq p (consbl b l))]
             (Lem1 b (app m2 p) m1 l n h1 h2 t1
                   (Eq_space_bl b m2 p l h3 t2)))
           ([i:Ind][l:l_ch][h:(prop2 m1 m2 p n l)]
             [t1:(formn n (consltr i l))][t2:(Eq p (consltr i l))]
             (Lem1 true (app m2 p) m1 (consltr i l) (S n) h1 h2
                   (fsp n i l t1)
                   (Eq_space  m2 p (consltr i l)  h3 t2)))).

(* Formatting of a valid list *)

 Theorem format.
 Statement  (l:l_ch)(valide l)->(SigFormat l).
 Proof  [l:l_ch][h:(valide l)]
        (h SigFormat
        ([m:l_ch][h1:(word m)][h2:(inflg m)]
         (ExFormat m (lgth m) m (fword m h1 h2) (Eq_re m)))
        ([m1,m2,p:l_ch][h1:(word m1)][h2:(inflg m1)]
         [h3:(space m2)][h4:(SigFormat p)]
         (h4 (SigFormat (app m1 (app m2 p))) (Lem2 m1 m2 p h1 h2 h3)))).

(* Formatting of a list : this function will detect if the list is not valid *)

 Theorem format_all.
 Statement (l:l_ch)(SigFormat l)+{(NV l)}.
 Proof [l:l_ch](preproc l (SigFormat l)+{(NV l)}
            [h:(valide l)](inleft (SigFormat l) (NV l) (format l h))
            (inright (SigFormat l) (NV l))).

End Format.
```

```
(****************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3      *)
(****************************************************************************)
(*                                                                          *)
(*                       Natural numbers                                    *)
(*                                                                          *)
(*                    (uses Prelude + Prel_Lem)                             *)
(*                                                                          *)
(****************************************************************************)

Chapter Natural_numbers.

Section Peano.
Variable n : nat.

Axiom peano : (P:nat->Prop)(P O)->((u:nat)(P u)->(P (S u)))->(P n).

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

End Peano.

Section less_or_equal.
Variable n,m,p:nat.

Theorem le_n_S  (le n m)->(le (S n) (S m))
    Proof [h:(le n m)]
         (h [q:nat](le (S n) (S q))
            (le_n (S n))
            [q:nat](le_S (S n) (S q))).

Theorem le_trans (le n m)->(le m p)->(le n p)
    Proof [h1:(le n m)][h2:(le m p)]
         (h2 [q:nat](le n q) h1 (le_S n)).

Theorem le_n_Sn (le n (S n))
    Proof (le_S n n (le_n n)).

Theorem le_O_n (le O n)
    Proof (peano n (le O) (le_n O) (le_S O)).

End less_or_equal.

Section primitive_recursion.

(* Operator for primitive recursion *)

Theorem rec_S.
Variable A : Data.
Variable init : A.
Variable iter : (nat*A) -> A.

Local REC.
Body [l:nat](nat_REC l A init iter).

Local rec.
Body [l:nat](nat_rec l A init iter).

Statement (n:nat)<A>(iter <nat,A><n,(rec n)>)=(rec (S n)).
Proof [n:nat]
      (f_equal nat*A A iter <nat,A><n,(rec n)> (REC n)
          (peano n [x:nat]<nat*A><nat,A><<nat,A>Fst<(REC x)>,(rec x)>=(REC x)
            (refl_equal nat*A <nat,A><<nat,A>Fst<(REC O)>,(rec O)>)
            [u:nat][H:<nat*A><nat,A><<nat,A>Fst<(REC u)>,(rec u)>=(REC u)]
                (refl_equal nat*A <nat,A><<nat,A>Fst<(REC (S u))>,(rec (S u))>)
          [x:nat*A]<nat*A><nat,A><n,(rec n)>=x
              (peano n [x:nat]<nat>x=<nat,A>Fst<(REC x)> (refl_equal nat O)
                   [u:nat][H:<nat>u=<nat,A>Fst<(REC u)>]
                   (f_equal nat nat S u <nat,A>Fst<(REC u)> H)
          [x:nat]<nat*A><nat,A><n,(rec n)>=<nat,A><x,(rec n)>
          (refl_equal nat*A <nat,A><n,(rec n)>)))).

Subsection Recnat.
Variable A : Data.

Variable init : A.
Variable iter : nat -> A -> A.

Definition recnat.
Body [n:nat](nat_rec n A init [h:nat*A](h A iter)) : nat->A.

Theorem recnat_O.
Statement <A>init=(recnat O).
Proof (refl_equal A init).

Theorem recnat_S.
Statement (n:nat)<A>(iter n (recnat n))=(recnat (S n)).
Proof (rec_S A init [h:nat*A](h A iter)).

End Recnat.
End primitive_recursion.

Global pred.
```

```
Body (recnat nat O [u:nat][f:nat]u) : nat->nat.

Theorem predS (m:nat)<nat>m=(pred (S m))
Proof (recnat_S nat O [u:nat][f:nat]u).

Lemma le_pred_n (n:nat)(le (pred n) n)
Proof [n:nat](peano n [v:nat](le (pred v) v)
          (le_n O)
          [v:nat][h:(le (pred v) v)]
          (predS v [w:nat](le w (S v)) (le_n_Sn v))).

Theorem le_S_n (n,m:nat)(le (S n) (S m))->(le n m)
Proof
     [n,m:nat][h:(le (S n) (S m))]
     (sym_equal nat n (pred (S n)) (predS n)
     [u:nat](le u m)
     (sym_equal nat m (pred (S m)) (predS m)
     [u:nat](le (pred (S n)) u)
     (h [v:nat](le (pred (S n)) (pred v))
        (le_n (pred (S n)))
        [v:nat][f:(le (pred (S n)) (pred v))]
         (le_trans (pred (S n)) (pred v) (pred (S v)) f
             (predS v [u:nat](le (pred v) u ) (le_pred_n v)))))).

Definition gt [n,m:nat](le n m)->{}.

Axiom le_Sn_O : (n:nat)~(le (S n) O).

Theorem O_S (n:nat)~(<nat>O=(S n))
Proof [n:nat][h:<nat>O=(S n)]
     (le_Sn_O n (h [m:nat](le m O) (le_n O))).

End Natural_numbers.
```

```
(****************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3     *)
(****************************************************************************)
(*                                                                          *)
(*                       Lambo function                                     *)
(*                                                                          *)
(*                                                                          *)
(*                                                                          *)
(****************************************************************************)
    (**********************************************************************)
    (* f : nat->nat is an increasing and unbounded function               *)
    (* lambo(f)(n) = inf {m:nat| f(m)>n}                                  *)
    (* is computed using the following algorithm :                        *)
    (* lambo f n = if f(0) > n the 0 else limbo(1)                        *)
    (*        where limbo(i) = if f(i) > n then 0                          *)
    (*                         else let j = limbo(2i) in                   *)
    (*                  if f(j+i) > n then j else j+i                      *)
    (**********************************************************************)

(* use Prelude, Prel_Lem, Nat *)

Let one     : nat     = (S O).

(* Addition and multiplication by 2 *)
Let add : nat->nat->nat     = [n:nat][m:nat](n nat m S).

Let twice : nat->nat = [n:nat](add n n).

(* Operatof for pattern-matching *)
 Let      nat_match
     :     (n:nat)(P:nat->Prop)(P O)->((m:nat)(P (S m)))->(P n)
     =     [n:nat][P:nat->Prop][h1:(P O)][h2:(m:nat)(P (S m))]
         (peano n P h1 [m:nat][ind:(P m)](h2 m)).

(* Properties of addition                               *)
Theorem add_nO      (n:nat)<nat>(add n O)=n
Proof     [n:nat](peano n [u:nat](<nat>(add u O)=u)
            (refl_equal nat O)
            ([u:nat](f_equal nat nat S (add u O) u))).

Theorem add_ass     (u,v,w:nat)<nat>(add (add u v) w)=(add u (add v w))
Proof     [u,v,w:nat]
     (peano u [m:nat]<nat>(add (add m v) w)=(add m (add v w))
         (refl_equal nat (add v w))
         [m:nat](f_equal nat nat S (add (add m v) w) (add m (add v w)))).

(* Induction de contenu positif                              *)
Axiom nat_ind :(n:nat)(P:nat->Spec)
               (P O)->((m:nat)(P m)->(P (S m)))->(P n).

(* Absurdity *)

 Theorem abs      (A:Prop)({} -> A)
 Proof [A:Prop][h:{}](h A).

(* An axiomatized order relation on natural numbers *)

 Parameter      inf : nat->nat->Prop.

 Axiom tran_inf     :      (trans nat inf).
 Axiom infOO     :     (inf O O).
 Axiom infS_inf     :     (n,m:nat)(inf (S n) (S m))->(inf n m).
 Axiom inf_InfS     :     (n,m:nat)(inf n m)->(inf (S n) (S m)).
 Axiom infS     :     (n:nat)(inf n (S n)).
 Axiom absO     :     (inf one O)->{}.

 Let sup     : nat->nat->Prop     = [n,m:nat](inf (S m) n).

(* Properties of this order *)

 Lemma re_inf      (n:nat)(inf n n)
 Proof     [n:nat](peano n [u:nat](inf u u) infOO
           [u:nat][h:(inf u u)](inf_infS u u h)).

 Lemma infO       (n:nat)(inf O n)
 Proof     [n:nat](peano n (inf O) (infOO)
           [u:nat][h:(inf O u)](tran_inf O u (S u) h (infS u))).


 Lemma infS_O     (n:nat)(inf (S n) O)->{}
 Proof     [n:nat](peano n [u:nat](inf (S u) O)->{} absO
           ([u:nat][h:(inf (S u) O)->{}][t:(inf (S (S u)) O)]
            (h (tran_inf (S u) (S (S u)) O (infS (S u)) t)))).


 Lemma infSn_n      (n:nat)(inf (S n) n)->{}
 Proof      [n:nat](peano n [u:nat](inf (S u) u)->{} absO
           ([u:nat][h:(inf (S u) u)->{}]
            [t:(inf (S (S u)) (S u))](h (infS_inf (S u) u t)))).

Lemma inf_sup_abs     (n,m:nat)(inf n m)->(sup n m)->{}
Proof      [n,m:nat][h1:(inf n m)][h2:(sup n m)]
        (infSn_n n (tran_inf (S n) (S m) n (inf_infS n m h1) h2)).
```

```
(* A lemma : (sup n m)->(sup (twice n) (S m)) *)

 Lemma inf_add  (n,m:nat)(inf m (add n m))
 Proof [n,m:nat]
       (peano n [u:nat](inf m (add u m))
       (re_inf m)
       ([u:nat][h:(inf m (add u m))]
        (tran_inf m (add u m) (add (S u) m)  h (infS (add u m))))).

 Lemma sup_twice  (n,m:nat)(sup n m)->(sup (twice n) (S m))
 Proof [n,m:nat]
       (nat_match n [u:nat](sup u m)->(sup (twice u) (S m))
       [h:(sup O m)](abs (sup (twice O) (S m)) (infS_O m h))
       [u:nat][h:(sup (S u) m)]
       (inf_infS (S m) (add u (S u))
           (tran_inf (S m) (S u) (add u (S u))
           h (inf_add u (S u))))).


(* Program Lambo *)

(* Hypotheses *)
Variable f    : nat->nat.

(* f is unbounded *)
Hypothesis Unbound
    :     (n:nat)<nat>Sig([y:nat](sup (f y) n)).

(* f is increasing *)
Hypothesis Increas    : (n:nat)(m:nat)(inf n m)->(inf (f n) (f m)).

(* There is a procedure to decide if inf or sup hold *)

Axiom inf_sup     : (x:nat)(y:nat) {(inf x y)}+{(sup x y)}.
Axiom inf_sup0     : (x:nat)(y:nat)(inf x y)\/(sup x y).

(* We give n : nat and try to compute lambo(n) *)
Variable n : nat.

 Let Inf : nat->Prop   = [m:nat](inf (f m) n).
 Let Sup : nat->Prop   = [m:nat](sup (f m) n).

(* We only use the following properties of Inf and Sup *)

 Let bound : <nat>Sig([y:nat](sup (f y) n))
           = (Unbound n).

 Let Inf_Sup : (u:nat){(Inf u)}+{(Sup u)}
             = [u:nat](inf_sup (f u) n).

 Let Inf_Sup_abs : (u:nat)(Inf u)->(Sup u)->{}
             = [u:nat](inf_sup_abs (f u) n).

(* F is increasing is used this way *)

 Lemma infInf     (u,v:nat)(inf u v)->(Inf v)->(Inf u)
 Proof    [u,v:nat][h1:(inf u v)][h2:(Inf v)]
    (tran_inf (f u) (f v) n (Increas u v h1) h2).

(* Specifications *)
 Let Small : nat->Prop = [m:nat](i:nat)(inf (S i) m)->(Inf i).

 Lemma SmallO  (Small O)
 Proof [i:nat][h:(inf (S i) O)](abs (Inf i) (infS_O i h)).

(* The initial specification *)
 Let Lambo : Spec = <nat>Sig2(Sup,Small).

 Fact Lem1  (m:nat)(Inf m)->(Small (S m))
 Proof          [m:nat][h:(Inf m)][i:nat][q:(inf (S i) (S m))]
          (infInf i m (infS_inf i m q) h).

(* Transformation of the specification *)
 Let Lambo1 : Spec
            = <nat>Sig2(Inf,[m:nat](Sup (S m)))+{(Sup O)}.

 Lemma Reduct1  Lambo1->Lambo
 Proof [h:Lambo1]
       (h Lambo [h1:<nat>Sig2(Inf,[m:nat](Sup (S m)))]
        (h1 Lambo
           ([m:nat][f1:(Inf m)][f2:(Sup (S m))]
            (exist2 nat Sup Small (S m) f2 (Lem1 m f1))))
        [h2:(Sup O)](exist2 nat Sup Small O h2 SmallO)).

(* Intermediate function *)
 Let Limbo : nat -> Spec
     = [i:nat]
     (sup i O)-> <nat>Sig2(Inf,[m:nat](Sup (add i m)))+{(Sup O)}.

(* (add one m)=(S m) by Beta reduction *)

 Lemma Reduct2  (Limbo one)->Lambo1
 Proof [h:(Limbo one)](h (re_inf one)).
```

```
(* Termination *)

(* A parametrzed order *)
 Let bd : nat->nat->nat->Prop
        = [y:nat][u:nat][v:nat]((sup u v)/\(inf v y)).

(* Property to be well-formed *)
 Let wf_bd
     : nat->nat->Spec
     = [y:nat][i:nat]
     (P:nat->Spec)((v:nat)((u:nat)(bd y u v)->(P u))->(P v))->(P i).

(* Some property of bd *)

 Fact Lem2  (y1,y2,u,v:nat)(bd y1 u v)->(inf v y2)->(bd y2 u v)
 Proof [y1,y2,u,v:nat][h1:(bd y1 u v)][h2:(inf v y2)]
        <(sup u v),(inf v y2)>{<(sup u v),(inf v y1)>Fst{h1},h2}.

 Fact Lem3  (y,u,v,w:nat)(bd y u v)->(bd y v w)->(sup y w)
 Proof [y,u,v,w:nat][h1:(bd y u v)][h2:(bd y v w)]
       (tran_inf (S w) v y <(sup v w),(inf w y)>Fst{h2}
       <(sup u v),(inf v y)>Snd{h1}).

 Lemma Term  (i,y:nat)(sup i y)->(wf_bd y i)
 Proof [i,y:nat][h:(sup i y)]
       [P:nat->Spec][q:(v:nat)((u:nat)(bd y u v)->(P u))->(P v)]
       (q i [u:nat][g:(bd y u i)]
      (except (P u) (inf_sup_abs i y <(sup u i),(inf i y)>Snd{g} h))).

(* Proof of (wf_bd y) by induction on y *)

 Lemma cas_base (i:nat)(sup i O)->(wf_bd O i)
 Proof [i:nat][h:(sup i O)][P:nat->Spec]
       [q:(v:nat)((u:nat)(bd O u v)->(P u))->(P v)]
    (q i [u:nat][g:(bd O u i)]
       (except (P u) (inf_sup_abs i O <(sup u i),(inf i O)>Snd{g} h))).

 Lemma cas_ind  (i,y:nat)(wf_bd y i)->(wf_bd (S y) i)
 Proof [i,y:nat][ind:(wf_bd y i)][P:nat->Spec]
       [q:(v:nat)((u:nat)(bd (S y) u v)->(P u))->(P v)]
       (ind P ([v:nat][g:(u:nat)(bd y u v)->(P u)]
           (q v ([u:nat][h:(bd (S y) u v)]
             (inf_sup v y (P u)
             [t1:(inf v y)](g u (Lem2 (S y) y u v h t1))
             [t2:(sup v y)]
             (q u ([w:nat][l:(bd (S y) w u)]
                    (except (P w)
                      (infSn_n v
                      (tran_inf (S v) (S y) v
                          (Lem3 (S y) w u v l h)
                          t2)))))))))).

Theorem Wf  (y,i:nat)(sup i O)->(wf_bd y i)
 Proof     [y,i:nat][h:(sup i O)]
     (nat_ind y [u:nat](wf_bd u i) (cas_base i h) (cas_ind i)).

(* Actually we will use a simpler induction scheme *)

 Global Ind : nat->nat->(nat->Spec)->Spec
            = [y:nat][i:nat][P:nat->Spec]
          ((k:nat)((inf k y)->(P (twice k)))->(P k))->(P i).

(* Proof of the induction scheme *)
Theorem wf_Ind (y:nat)(i:nat)(sup i O)->(P:nat->Spec)(Ind y i P)
 Proof [y:nat][i:nat][h:(sup i O)][P:nat->Spec]
       (nat_ind y [u:nat](Ind u i P)
       ([q:(k:nat)((inf k O)->(P (twice k)))->(P k)]
        (q i [r:(inf i O)](except (P (twice i)) (inf_sup_abs i O r h))))
       ([u:nat][ind:(Ind u i P)]
        [q:(k:nat)((inf k (S u))->(P (twice k)))->(P k)]
        (ind ([k:nat][r:(inf k u)->(P (twice k))]
          (q k [t:(inf k (S u))]
             (inf_sup k u (P (twice k)) r
             [v:(sup k u)]
             (q (twice k)
                [s:(inf (twice k) (S u))]
                (except (P (twice (twice k)))
                  (inf_sup_abs (twice k) (S u) s
                      (sup_twice k u v)))))))))).

(* Proof of the Limbo's program  *)

 Fact Lem4  (u,v:nat)(Sup u)->(Inf v)->(inf v u)
  Proof [u,v:nat][h1:(Sup u)][h2:(Inf v)]
        (inf_sup0 v u (inf v u)
       [t1:(Inf v u)]t1
       [t2:(sup v u)]
       (abs (inf v u)
         (Inf_Sup_abs u
           (infInf u v (tran_inf u (S u) v (infS u) t2) h2)
           h1))).

 Let LimboSig : nat->Spec
           = [i:nat]<nat>Sig2(Inf,[m:nat](Sup (add i m))).
```

```
Lemma LimboLem  (i:nat)(sup i O)->(Inf O)->(LimboSig i)
Proof [i:nat][h1:(sup i O)][h2:(Inf O)]
      (bound (LimboSig i)
         [y:nat][hy:(Sup y)]
         (wf_Ind y i h1 LimboSig
            [k:nat][hk:(inf k y)->(LimboSig (twice k))]
            (Inf_Sup (add k O) (LimboSig k)
            [t1:(Inf (add k O))]
            (hk (Lem4 y k hy (add_nO k Inf t1)) (LimboSig k)
                [m:nat][u1:(Inf m)][u2:(Sup (add (twice k) m))]
                (Inf_Sup (add k m) (LimboSig k)
                [s1:(Inf (add k m))]
                (exist2 nat Inf [p:nat](Sup (add k p))
                    (add k m) s1 (add_ass k k m Sup u2))
                [s2:(Sup (add k m))]
                (exist2 nat Inf [p:nat](Sup (add k p))
                    m u1 s2)))
            [t2:(Sup (add k O))]
            (exist2 nat Inf [p:nat](Sup (add k p))
                O h2 t2)))).

Lemma Prog  (i:nat)(Limbo i)
Proof [i:nat][h:(sup i O)]
      (Inf_Sup O (LimboSig i)+{(Sup O)}
      [t1:(Inf O)](inleft (LimboSig i) (Sup O) (LimboLem i h t1))
      [t2:(Sup O)](inright (LimboSig i) (Sup O) t2)).

(* Final proof *)
Theorem LamboProg  Lambo
Proof   (Reduct1 (Reduct2 (Prog one))).
```

```
(*****************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3      *)
(*****************************************************************************)
(*                                                                           *)
(*                          Lists                                            *)
(*                                                                           *)
(*                         (use Prelude)                                     *)
(*                                                                           *)
(*****************************************************************************)

(* V "Prel_Lem"; V "Nat" *)

(* Some programs and results about lists *)
Chapter Lists_with_parameter.

Parameter A:Data.

Inductive Data list = nil : list | cons : A -> list -> list.

Axiom list_initial :
    (l:list)(P:list->Prop)(P nil)->((x:A)(m:list)(P m)->(P (cons x m)))->(P l).

Global app.
Body [l,m:list](l list m cons) : list->list->list.

Theorem app_nil.
Statement (l:list)<list>l=(app nil l).
Proof [l:list](refl_equal list l).

Theorem app_cons.
Statement (a:A)(l,m:list)<list>(cons a (app l m))=(app (cons a l) m).
Proof [a:A][l,m:list](refl_equal list (cons a (app l m))).

Theorem app_ass.
Statement (l,m,n : list)<list>(app (app l m) n)=(app l (app m n)).
Proof [l,m,n : list]
      (list_initial l [x:list]<list>(app (app x m) n)=(app x (app m n))
        (refl_equal list (app (app nil m) n))
          [x:A][m0:list][H:<list>(app (app m0 m) n)=(app m0 (app m n))]
          (app_cons x m0 m
               [x0:list]<list>(app x0 n)=(app (cons x m0) (app m n))
               (app_cons x (app m0 m) n
                 [x0:list]<list>x0=(app (cons x m0) (app m n))
                 (app_cons x m0 (app m n)
                    [x0:list]<list>(cons x (app (app m0 m) n))=x0
                    (f_equal list list (cons x)
                       (app (app m0 m) n) (app m0 (app m n)) H))))).

Global mil.
Body [a:A][l,m:list](app l (cons a m)) : A->list->list->list.

(* Primitive recursion on lists *)

Theorem rec_cons.

Variable B : Data.

Variable init : B.
Variable iter : A -> (list*B) -> B.

Local REC.
Body [l:list](list_REC l B init iter).

Local rec.
Body [l:list](list_rec l B init iter).

Statement (a:A)(l:list)<B>(iter a <list,B><l,(rec l)>)=(rec (cons a l)).

Proof [a:A][l:list]
      (f_equal list*B B (iter a) <list,B><l,(rec l)> (REC l)
        (list_initial l
           [x:list]<list*B><list,B><<list,B>Fst<(REC x)>,(rec x)>=(REC x)
           (refl_equal list*B <list,B><<list,B>Fst<(REC nil)>,(rec nil)>)
           [x:A][m:list]
           [H:<list*B><list,B><<list,B>Fst<(REC m)>,(rec m)>=(REC m)]
           (refl_equal list*B
             <list,B><<list,B>Fst<(REC (cons x m))>,(rec (cons x m))>)
        [x:list*B]<list*B><list,B><l,(rec l)>=x
        (list_initial l [x:list]<list>x=<list,B>Fst<(REC x)>
           (refl_equal list nil)
           [x:A][m:list][H:<list>m=<list,B>Fst<(REC m)>]
           (f_equal list list (cons x) m <list,B>Fst<(REC m)> H)
        [x:list]<list*B><list,B><l,(rec l)>=<list,B><x,(rec l)>
        (refl_equal list*B <list,B><l,(rec l)>)))).

Section Reclist.
Variable B : Data.

Variable init : B.
Variable iter : A -> list -> B -> B.

Definition reclist.
Body [l:list](list_rec l B init [a:A][h:list*B](h B (iter a))) : list->B.
```

```
Theorem reclist_nil.
Statement <B>init=(reclist nil).
Proof (refl_equal B init).

Theorem reclist_cons.
Statement (a:A)(l:list)<B>(iter a l (reclist l))=(reclist (cons a l)).
Proof (rec_cons B init [a:A][h:list*B](h B (iter a))).

End Reclist.

Definition tail.
Body (reclist list nil [a:A][m,t:list]m) : list->list.

Theorem tail_cons.
Statement (a:A)(l:list)<list>l=(tail (cons a l)).
Proof (reclist_cons list nil [a:A][m,t:list]m).

(* L'ordre des longueurs sur les listes *)

Definition length.
Body [l:list](l nat O [a:A]S) : list->nat.

Section length_order.
Definition lel [l,m:list](le (length l) (length m)).

Variables a,b:A.
Variable l,m,n:list.

    Theorem lel_refl (lel l l)
    Proof (le_n (length l)).

    Theorem lel_trans (lel l m)->(lel m n)->(lel l n)
    Proof (le_trans (length l) (length m) (length n)).

    Theorem lel_cons_cons (lel l m)->(lel (cons a l) (cons b m))
    Proof (le_n_S (length l) (length m)).

    Theorem lel_cons (lel l m)->(lel l (cons b m))
    Proof (le_S (length l) (length m)).

    Theorem lel_tail (lel (cons a l) (cons b m)) -> (lel l m)
    Proof (le_S_n (length l) (length m)).

End length_order.

Definition Null [l:list]<list>nil=l.

Theorem lel_nil (l:list)(lel l nil)->(Null l)
Proof [l:list]
    (list_initial l [m:list](lel m nil)->(Null m)
        [h:(lel nil nil)](refl_equal list nil)
        [a:A][m:list][h:(lel m nil)->(Null m)]
        [h':(lel (cons a m) nil)]
        (le_Sn_O (length m) h' (Null (cons a m)))).

Theorem nil_cons (a:A)(m:list)~(Null (cons a m))
Proof [a:A][m:list][h:(Null (cons a m))]
        (O_S (length m) (f_equal list nat length nil (cons a m) h)).

End Lists_with_parameter.
```

```
(****************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3     *)
(****************************************************************************)
(*                                                                          *)
(*                          QUICKSORT for lists                             *)
(*                                                                          *)
(*                        Christine Paulin-Mohring                          *)
(*                                                                          *)
(****************************************************************************)
(* #use "CPtac"; V "Prel_Lem"; V "Nat"; V "List" *)

(* Induction on lists *)
Axiom list_ind :
    (l:list)(P:list->Spec)(P nil)->((x:A)(m:list)(P m)->(P (cons x m)))->(P l).

Chapter Quicksort.

(******************************)
(* A decidable relation on A *)
(******************************)

Variable   inf : A -> A -> Prop.
Definition sup  [x,y:A]~(inf x y).

Hypothesis inf_sup : (x,y:A){(inf x y)}+{(sup x y)}.

(******************************)
(* Equivalence of two lists  *)
(******************************)

Inductive Definition permut : list->list->Prop =
     permut_nil  : (permut nil nil)
    |permut_tran : (l,m,n:list)(permut l m)->(permut m n)->(permut l n)
    |permut_cmil : (a:A)(l,m,n:list)
          (permut l (app m n))->(permut (cons a l) (mil a m n))
    |permut_milc : (a:A)(l,m,n:list)
          (permut (app m n) l)->(permut (mil a m n) (cons a l)).

(*******************************)
(* if l eq m then a.l eq b.m  *)
(*******************************)

Section Equivalence_lemma.
Variable a:A.
Variables l,m:list.

Theorem permut_cons.
Statement (permut l m)->(permut (cons a l) (cons a m)).
Proof (permut_cmil a l nil m).

(************************************)
(* if l eq m then m eq l        *)
(************************************)

Theorem permut_sym.
Statement (permut l m)->(permut m l).
Proof [h:(permut l m)]
        (h [u,v:list](permut v u) permut_nil
            [u,v,w:list][f1:(permut v u)][f2:(permut w v)]
            (permut_tran w v u f2 f1)
            permut_milc permut_cmil).

End Equivalence_lemma.

Section Equivalence_append.

(*****************************************************************)
(*  if m1 eq n1  and m2 eq n2 then (app m1 m2) eq (app n1 n2)  *)
(*****************************************************************)


Variables m,m1,m2,n1,n2 :list.
Hypothesis h0 : (permut m (app m1 m2)).
Hypothesis h1:(permut m1 n1).
Hypothesis h2:(permut m2 n2).


Remark permut_app_mil.
Variables t,z:list.
Variable  a : A.
Variables u,v,w:list.

Statement  (permut (app u t) (app (app v w) z))
        ->(permut (app (cons a u) t) (app (mil a v w) z)).
Proof [h:(permut (app u t) (app (app v w) z))]
    (sym_equal list (app (app v (cons a w)) z) (app v (cons a (app w z)))
    (app_ass v (cons a w) z)
    [x:list](permut (app (cons a u) t) x)
    (permut_cmil a (app u t) v (app w z)
        (app_ass v w z [x:list](permut (app u t) x) h)))).

Remark permut_app1.
Statement (u:list)(permut (app u n2) (app u m2)).
Proof [u:list](list_initial u [v:list](permut (app v n2) (app v m2))
```

```
                    (permut_sym m2 n2 h2)
                    [a:A][v:list](permut_cons a (app v n2) (app v m2))).

Property permut_app_app.
Statement (permut (app m1 m2) (app n1 n2)).
Proof (h1 [u,v:list](permut (app u m2) (app v n2))
          h2
          [u,v,w:list][f1:(permut (app u m2) (app v n2))]
           [f2:(permut (app v m2) (app w n2))]
          (permut_tran (app u m2) (app v n2) (app w n2) f1
            (permut_tran (app v n2) (app v m2) (app w n2)
         (permut_appl v) f2))
       (permut_app_mil m2 n2)
       [a:A][u,v,w:list][f1:(permut (app (app v w) m2) (app u n2))]
       (permut_sym (app (cons a u) n2) (app (mil a v w) m2)
         (permut_app_mil n2 m2 a u v w
         (permut_sym (app (app v w) m2) (app u n2) f1)))).

Property permut_app.
Statement (permut m (app n1 n2)).
Proof (permut_tran m (app m1 m2) (app n1 n2) h0 permut_app_app).

End Equivalence_append.

(*****************************************************)
(* Each element of a list satisfy a given relation *)
(*****************************************************)

Section Rlist.

Variable R : A->Prop.

Inductive Definition Rlist  : list -> Prop =
    Rnil : (Rlist nil)
   |Rcons : (x:A)(l:list)(R x)->(Rlist l)->(Rlist (cons x l)).

Theorem Rlist_app.
Statement (m,n:list)(Rlist m)->(Rlist n)->(Rlist (app m n)).
Proof [m,n:list][hm:(Rlist m)][hn:(Rlist n)]
       (hm [u:list](Rlist (app u n)) hn [a:A][l:list](Rcons a (app l n))).

Subsection Rlist_cons.

Variable a : A.
Variable l : list.
Hypothesis Rc : (Rlist (cons a l)).

Fact Rlist_hd.
Statement (R a).
Local proph [m:list](~(Null m))->(R (m A a [b:A][n:A]b)).
Proof (Rc proph
            [h:~(Null nil)](h (refl_equal list nil) (R a))
            [b:A][m:list][h1:(R b)][h2:(proph m)][h3:~(Null (cons b m))]h1
        (nil_cons a l)).

Fact Rlist_tl.
Statement (Rlist l).
Proof (sym_equal list l (tail (cons a l)) (tail_cons a l) Rlist
        (Rlist_rec (cons a l) Rc [m:list](Rlist (tail m)) Rnil
            [b:A][m:list][h1:(R b)][h2:(Rlist m)/\(Rlist (tail m))]
            (tail_cons b m Rlist <(Rlist m),(Rlist (tail m))>Fst(h2)))).

End Rlist_cons.

Subsection Rlist_append.
Variable n,m : list.

Property Rlist_appl.
Statement (Rlist (app n m))->(Rlist n).
Local propa1 [n,m:list](Rlist (app n m))->(Rlist n).

Proof (list_initial n [u:list](propa1 u m)
                        [h:(Rlist (app nil m))]Rnil
                        [a:A][l:list][h1:(propa1 l m)]
                        [h2:(Rlist (app (cons a l) m))]
                        (Rcons a l (Rlist_hd a (app l m) h2)
                                   (h1 (Rlist_tl a (app l m) h2))))).

Local propa2 [n,m:list](Rlist (app n m))->(Rlist m).

Property Rlist_app2.
Statement (Rlist (app n m))->(Rlist m).

Proof (list_initial n [u:list](propa2 u m)
                        [h:(Rlist (app nil m))]h
                        [a:A][l:list][h1:(propa2 l m)]
                        [h2:(Rlist (app (cons a l) m))]
                        (h1 (Rlist_tl a (app l m) h2))).

End Rlist_append.

(*****************************************)
(* if l eq m then (Rlist l)=>(Rlist m)  *)
(*****************************************)
```

```
Theorem Rpermut.

Variables m,n:list.
Hypothesis heq : (permut m n).

Local propeq [u,v:list](Rlist u)->(Rlist v).

Statement (propeq m n).
Proof (heq propeq
           [h:(Rlist nil)]h
           [u,v,w:list][h1:(propeq u v)][h2:(propeq v w)][h3:(Rlist u)]
            (h2 (h1 h3))
           [a:A][u,v,w:list][h1:(propeq u (app v w))][h2:(Rlist (cons a u))]
            (Rlist_app v (cons a w) (Rlist_app1 v w (h1 (Rlist_tl a u h2)))
                 (Rcons a w (Rlist_hd a u h2)
                          (Rlist_app2 v w (h1 (Rlist_tl a u h2))))))
           [a:A][u,v,w:list][h1:(propeq (app v w) u)][h2:(Rlist (mil a v w))]
            (Rcons a u (Rlist_hd a w (Rlist_app2 v (cons a w) h2))
                          (h1 (Rlist_app v w (Rlist_app1 v (cons a w) h2)
                               (Rlist_tl a w (Rlist_app2 v (cons a w) h2)))))).

End Rlist.

Section Inf_Sup.
(*******************************************)
(* (inf_list x l) == (inf x a) for a in l *)
(* (sup_list x l) == (sup x a) for a in l *)
(*******************************************)

Hypothesis x : A.
Hypothesis l : list.

    Definition inf_list (Rlist (inf x) l).
    Definition sup_list (Rlist (sup x) l).

End inf_sup.

(***************************************************)
(* The process of splitting a list into two parts *)
(***************************************************)

Section Splitting.

Variable a : A.

Inductive Definition Lem_spec [l:list] : Spec =
    Lem_exist : (l1,l2:list)
    (sup_list a l1)->(inf_list a l2)->(permut l (app l1 l2))
    ->(lel l1 l)->(lel l2 l)->(Lem_spec l).

Theorem Lem.
Statement  (l:list)(Lem_spec l).
Proof [l:list](list_ind l Lem_spec
    (Lem_exist nil nil nil (Rnil (sup a)) (Rnil (inf a))
            (permut_nil) (lel_refl nil) (lel_refl nil))
        [b:A][m:list][h:(Lem_spec m)]
        (h (Lem_spec (cons b m))
           [m1,m2:list][h1:(sup_list a m1)][h2:(inf_list a m2)]
           [h3:(permut m (app m1 m2))][f1:(lel m1 m)][f2:(lel m2 m)]
            (inf_sup a b (Lem_spec (cons b m))
                 [t1:(inf a b)]
                 (Lem_exist (cons b m) m1 (cons b m2)
                     h1 (Rcons (inf a) b m2 t1 h2)
                     (permut_cmil b m m1 m2 h3)
                     (lel_cons b m1 m f1)
                     (lel_cons_cons b b m2 m f2))
                 [t2:(sup a b)]
                 (Lem_exist (cons b m) (cons b m1) m2
                     (Rcons (sup a) b m1 t2 h1) h2
                     (permut_cons b m (app m1 m2) h3)
                     (lel_cons_cons b b m1 m f1)
                     (lel_cons b m2 m f2))))).

End Splitting.

(***************************************)
(* Definition for a list to be sorted *)
(***************************************)

Inductive Definition sort : list->Prop =
    sort_nil : (sort nil)
   |sort_mil : (a:A)(l,m:list)(sup_list a l)->(inf_list a m)
        ->(sort l)->(sort m)->(sort (mil a l m)).

(************************************)
(* Proof of the induction principle *)
(************************************)

Theorem induction.

Variable l:list.
Variable P:list->Spec.
Hypothesis hnil : (P nil).
```

```
Hypothesis hrec :(a:A)(m:list)((n:list)(lel n m)->(P n))->(P (cons a m)).

Statement (P l).

Proof
 (list_ind l [m:list](n:list)(lel n m)->(P n)
     [n:list][h:(lel n nil)](eq_spec list nil n (lel_nil n h) P hnil)
     [a:A][m:list][hyp:(n:list)(lel n m)->(P n)][n:list]
     (list_ind n [p:list](lel p (cons a m))->(P p)
         [h:(lel nil (cons a m))] hnil
         [b:A][p:list][h:(lel p (cons a m))->(P p)]
          [f1:(lel (cons b p) (cons a m))]
          (hrec b p
         ([r:list][f2:(lel r p)]
             (hyp r (lel_trans r p m f2 (lel_tail b a p m f1))))))
 l (lel_refl l)).

(****************************)
(* Quicksort Specification *)
(****************************)

Definition Specif [l:list]<list>Sig2(sort,(permut l)).

(***********************)
(* Proof of Quicksort *)
(***********************)

Theorem Quicksort  (l:list)(Specif l)
Proof [l:list]
     (induction l Specif
     (exist2 list sort (permut nil) nil sort_nil permut_nil)
     ([a:A][m:list][hyprec:(n:list)(lel n m)->(Specif n)]
      (Lem a m (Specif (cons a m))
         [m1,m2:list][h1:(sup_list a m1)][h2:(inf_list a m2)]
         [h3:(permut m (app m1 m2))][h4:(lel m1 m)][h5:(lel m2 m)]
         (hyprec m1 h4 (Specif (cons a m))
         ([n1:list][t1:(sort n1)][u1:(permut m1 n1)]
          (hyprec m2 h5 (Specif (cons a m))
           ([n2:list][t2:(sort n2)][u2:(permut m2 n2)]
             (exist2 list sort (permut (cons a m))
             (mil a n1 n2)
             (sort_mil a n1 n2 (Rpermut (sup a) m1 n1 u1 h1)
                  (Rpermut (inf a) m2 n2 u2 h2) t1 t2)
             (permut_cmil a m n1 n2
                 (permut_app m m1 m2 n1 n2 h3 u1 u2)))))))))).

End Quicksort.
```

```
(* #use "CPtac"; V "Prel_Lem"; #use "automatic"; loadf "prel_autom";V "set";;*)

(**************************************************************************)
(*      Development of Warshall's Algorithm                              *)
(*      Christine Paulin-Mohring                                         *)
(*      after F. Pfenning                                                *)
(*      Program Development through Proof Transformation                 *)
(*      Calcul des Constructions V4.10                                   *)
(**************************************************************************)

(*******************************************)
(* V is finite : each set on V is finite  *)
(*******************************************)

Axiom finit_V : (enumerate allV).
#(add_resolve "finit_V").

(******************************************)
(* A decidable relation on V : the edges *)
(******************************************)

Variable E : V->V->Prop.
Variable dec_E : (x,y:V){(E x y)}+{~(E x y)}.
#(add_resolve "dec_E").

(*************************************************************************)
(* Two vertices are connected in W if there exists a path from x to y  *)
(* with inner vertex in W.                                              *)
(*************************************************************************)

(*******************************************************)
(*              Q                                     *)
(*   We write x ---> y   for (connected Q x y)        *)
(*******************************************************)

Inductive Definition connected [W:Set] : V->V->Prop
         = direct : (x,y:V)(E x y)->(connected W x y)
         | one_in : (x,y,z:V)(E x y)->(in W y)->(connected W y z)
                    ->(connected W x z).

#(add_resolve "direct").

(**************************************************************************)
(*    First Part, general lemmas and naive proof                         *)
(**************************************************************************)

(*******************************************************)
(*           Lemmas about connected                    *)
(*******************************************************)

(*****************************************************)
(*           W                          W'          *)
(*  If x ---> y and W included in W' then x ---> y  *)
(*****************************************************)

Goal (W,W':Set)(x,y:V)(incl W W')->(connected W x y)->(connected W' x y).
By (intros THEN elim_last THEN intros_auto).
By (resolve_with <<(one_in y0)>> THEN automatic).
By (resolve <<H>> THEN assumption).
Save connect_incl.

(****************************************************************)
(*  A trivial application to be used as an automatic tactic     *)
(****************************************************************)

Goal (W:Set)(x,y,z:V)(connected W y z)->(connected (add W x) y z).
Intros.
By (resolve <<(connect_incl W)>> THEN automatic).
Save connect_add.
#(add_resolve "connect_add").

(**************************************************************)
(*          W           W                     W              *)
(*   If x ---> y and y ---> z and y in W then x ---> z       *)
(**************************************************************)

Goal (W:Set)(x,y,z:V)
        (connected W x y)->(connected W y z)->(in W y)->(connected W x z).
By (DO 5 intro).
By (elim <<H>> THEN intros).
By (resolve_with <<(one_in y0)>> THEN automatic).
By (resolve_with <<(one_in y0)>> THEN automatic).
Save connect_trans.

(********************************************)
(*      {}              E                  *)
(*   x--->y      then x -> y               *)
(********************************************)

Goal (x,y:V)(connected empty x y)->(E x y).
Intros.
By (elim <<H>> THEN intros_auto).
Do (elim_unfold) H1.
```

```
Save connect_empty.
#(add_resolve "connect_empty").

(*************************************************************************)
(*          Q+y              Q               Q              Q            *)
(* if     x ---> z   then x ---> z  or (x ---> y and y --->z)            *)
(*************************************************************************)

Goal (Q:Set)(x,y,z:V)(connected (add Q y) x z)
    ->((connected Q x z)\/((connected Q x y)/\(connected Q y z))).
By (intros THEN elim_last THEN intros_auto).
By (resolve <<H2>> THEN intro).
By (resolve <<H1>> THEN intro).
Resolve or_introl.
By (resolve_with <<(one_in y0)>> THEN automatic).
By (elim_with <<<V>y0=y>> THEN automatic).
By (elim_last THEN intros).
By (cut <<(connected Q x0 y)>> THEN automatic).
By (resolve <<H1>> THEN intro).
By (resolve_with <<(one_in y0)>> THEN automatic).
By (elim_with <<<V>y0=y>> THEN automatic).
Save connect_lem.


Goal (Q:Set)(x,y,z:V)(~(connected Q x z))->(connected (add Q y) x z)
    ->((connected Q x y)/\(connected Q y z)).
By (intros THEN elim <<(connect_lem Q x y z)>> THEN intros_auto).
By (resolve <<(absurd (connected Q x z))>> THEN automatic).
Save connect_cut.
#(add_resolve "connect_cut").

(*************************************************************************)
(*          Specification                                                *)
(*************************************************************************)

(******************)
(* Case W is empty *)
(******************)

Goal (x,y:V){(connected empty x y)}+{~(connected empty x y)}.
Intros.
By (resolve <<(dec_E x y)>> THEN intros_auto).
Resolve right.
Red.
Intro.
By (resolve <<(absurd (E x y))>> THEN automatic).
Save warshall_empty.
#(add_resolve "warshall_empty").

(*********************************************)
(*  Induction step                 *)
(*********************************************)

Goal (Q:Set)(z:V)
     ((x,y:V){(connected Q x y)}+{~(connected Q x y)})
     ->(x,y:V){(connected (add Q z) x y)}+{~(connected (add Q z) x y)}.

By (intros THEN resolve <<(H x y)>> THEN intros_auto).
  (**********************************)
  (* Case ~(connected Q x y)        *)
  (**********************************)
By (resolve <<(H x z)>> THEN intro).
     (**********************************)
     (* Subcase (connected Q x z)      *)
     (**********************************)
     By (resolve <<(H z y)>> THEN intro).
          (**********************************)
          (* Subsubcase (connected Q z y)  *)
          (**********************************)
          By (resolve <<left>> THEN resolve_with <<(connect_trans z)>>
                               THEN automatic).
          (**********************************)
          (* Subsubcase ~(connected Q z y) *)
          (**********************************)
          By (resolve <<right>> THEN unfold_head THEN intro).
          By (elim <<(connect_cut Q x z y)>> THEN automatic).
     (**********************************)
     (* Subcase ~(connected Q x y)     *)
     (**********************************)
     By (resolve <<right>> THEN unfold_head THEN intro).
     By (elim <<(connect_cut Q x z y)>> THEN intros_auto).
     By (resolve <<H1>> THEN automatic).
Save warshall_ind.
#(add_resolve "warshall_ind").

(********************************************************)
(*  First proof by induction                            *)
(********************************************************)

Goal (x,y:V){(connected allV x y)}+{~(connected allV x y)}.
By (elim <<finit_V>> THEN automatic).
Save warshall1.

(*************************************************************************)
```

```
(*    Second Part, keeping results in a matrice                              *)
(***************************************************************************)

(*************************************)
(*      Arrays on V          *)
(*************************************)


Section Array.
Variable A: V->Spec.

Inductive Definition array : Set->Spec =
    empty_array : (array empty)
  | add_array   : (P:Set)(y:V)(A y)->(array P)->(array (add P y)).

End Array.
#(add_resolve "empty_array").
#(add_resolve "add_array").

(***************)
(* Access      *)
(***************)

Goal (A:V->Spec)(P:Set)(array A P)->(x:V)(in P x)->(A x).
By ((DO 3 intro) THEN elim_last THEN intros).
Resolve except.
By (resolve <<(absurd (in empty x))>> THEN automatic).
By (elim <<(add_null P0 x y)>> THEN intros_auto).
By (incomplet [4] <<(eq_spec H3)>> THEN automatic).
Save acces.

(*******************)
(* Initialisation  *)
(*******************)

Goal (A:V->Spec)((y:V)(A y))->(array A allV).
By (intros THEN resolve <<finit_V>> THEN automatic).
Save array_init.
#(add_resolve "array_init").

(*******************************************************************)
(* Representation of a relation on a finite set by a matrix        *)
(*******************************************************************)

Definition repr_matrix
    [P:V->V->Prop](array [x:V](array [y:V]{(P x y)}+{~(P x y)} allV) allV).

Goal (P:V->V->Prop)((x,y:V){(P x y)}+{~(P x y)})->(repr_matrix P).
By (red THEN automatic).
Save build_matrix.
#(add_resolve "build_matrix").

Goal (P:V->V->Prop)(repr_matrix P)->(x,y:V){(P x y)}+{~(P x y)}.
By (unfold "repr_matrix" THEN intros).
By (incomplet [2;4] <<(acces allV y)>> THEN automatic).
By (incomplet [2;4] <<(acces allV x)>> THEN automatic).
Save acces_matrix.
#(add_resolve "acces_matrix").

(*************************************)
(*      New specification            *)
(*************************************)

Goal (repr_matrix (connected allV)).
By (elim <<finit_V>> THEN automatic).
Save warshall_rep.
#(add_resolve "warshall_rep").

Goal (x,y:V){(connected allV x y)}+{~(connected allV x y)}.
By automatic.
Save warshall2.
```

```
(*****************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3      *)
(*****************************************************************************)
(*                                                                           *)
(*                         Natural numbers                                   *)
(*                                                                           *)
(*                         (uses Prelude)                                    *)
(*                                                                           *)
(*****************************************************************************)

(* #use "CPtac"; V "Prel_Lem"; #use "automatic"; load "prel_autom"; *)
Chapter Natural_numbers.

Section Peano.
Variable n : nat.

Axiom peano : (P:nat->Prop)(P O)->((u:nat)(P u)->(P (S u)))->(P n).

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

End Peano.

Section less_or_equal.
Variable n,m,p:nat.

Theorem le_n_S  (le n m)->(le (S n) (S m))
    Proof [h:(le n m)]
            (h [q:nat](le (S n) (S q))
               (le_n (S n))
               [q:nat](le_S (S n) (S q))).

Theorem le_trans (le n m)->(le m p)->(le n p)
    Proof [h1:(le n m)][h2:(le m p)]
            (h2 [q:nat](le n q) h1 (le_S n)).

Theorem le_n_Sn (le n (S n))
    Proof (le_S n n (le_n n)).

Theorem le_O_n (le O n)
    Proof (peano n (le O) (le_n O) (le_S O)).

End less_or_equal.

Section primitive_recursion.

(* Operator for primitive recursion *)

Subsection primitive_equality.
Variable A : Data.
Variable init : A.
Variable iter : (nat*A) -> A.

Local REC.
Body [l:nat](nat_REC l A init iter).

Local rec.
Body [l:nat](nat_rec l A init iter).

Goal (n:nat)<A>(iter <nat,A><n,(rec n)>)=(rec (S n)).
Intros.
Do (elim_with <<<A>(iter (REC n))=(rec (S n))>> THEN automatic).
Do (incomplet [3]) (f_equal iter).
Do (elim_with) <nat*A>(<nat,A><<nat,A>Fst<(REC n)>,(rec n)>)=(REC n).
By (resolvew <<(peano n)>> THEN automatic).
Do (elim_with <<<nat>n=<nat,A>Fst<(REC n)>>> THEN automatic).
By (resolvew <<(peano n)>> THEN intros_auto).
By (elim_with
    <<<nat>(S <nat,A>Fst<(REC u)>)=<nat,A>Fst<(REC (S u))>>>
    THEN automatic).
Do (incomplet [3]) (f_equal S).
Assumption.
Save rec_S.

End primitive_equality.

Subsection Recnat.
Variable A : Data.

Variable init : A.
Variable iter : nat -> A -> A.

Definition recnat.
Body [n:nat](nat_rec n A init [h:nat*A](h A iter)) : nat->A.

Goal <A>init=(recnat O).
By automatic.
Save recnat_O.

Goal (n:nat)<A>(iter n (recnat n))=(recnat (S n)).
By (unfold) recnat.
By (intro THEN elim <<rec_S>> THEN automatic).
Save recnat_S.
```

```
End Recnat.
End primitive_recursion.

Global pred.
Body (recnat nat O [u:nat][f:nat]u) : nat->nat.

Theorem predS (m:nat)<nat>m=(pred (S m))
Proof (recnat_S nat O [u:nat][f:nat]u).

Lemma le_pred_n (n:nat)(le (pred n) n)
Proof [n:nat](peano n [v:nat](le (pred v) v)
                (le_n O)
                [v:nat][h:(le (pred v) v)]
                (predS v [w:nat](le w (S v)) (le_n_Sn v))).
#(add_resolve "le_pred_n").

Theorem le_S_n (n,m:nat)(le (S n) (S m))->(le n m)
Proof
    [n,m:nat][h:(le (S n) (S m))]
    (sym_equal nat n (pred (S n)) (predS n)
        [u:nat](le u m)
        (sym_equal nat m (pred (S m)) (predS m)
        [u:nat](le (pred (S n)) u)
        (h [v:nat](le (pred (S n)) (pred v))
           (le_n (pred (S n)))
           [v:nat][f:(le (pred (S n)) (pred v))]
            (le_trans (pred (S n)) (pred v) (pred (S v)) f
                   (predS v [u:nat](le (pred v) u ) (le_pred_n v)))))).

Definition gt [n,m:nat](le n m)->{}.

Axiom le_Sn_O : (n:nat)~(le (S n) O).

Theorem O_S (n:nat)~(<nat>O=(S n))
Proof [n:nat][h:<nat>O=(S n)]
        (le_Sn_O n (h [m:nat](le m O) (le_n O))).

End Natural_numbers.

#(add_resolution ["le_n";"le_S";"le_n_S";"le_n_Sn";"le_O_n";"pred_S";
                  "le_pred_n";"le_Sn_O";"O_S"]).
#(add_tac "le" "le_S_n" 1 (resolve <<le_S_n>> THEN trivial)).
```

```
(******************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3       *)
(******************************************************************************)
(*                                                                            *)
(*                          Lists                                             *)
(*                                                                            *)
(*                          (use Prelude)                                     *)
(*                                                                            *)
(******************************************************************************)

(* #use "CPtac"; V "Prel_Lem"; #use "automatic"; load "prel_autom";  V "Nat" *)

(* Some programs and results about lists *)
Chapter Lists_with_parameter.

Parameter A:Data.

Inductive Data list = nil : list | cons : A -> list -> list.

Axiom list_initial :
    (l:list)(P:list->Prop)(P nil)->((x:A)(m:list)(P m)->(P (cons x m)))->(P l).

Global app.
Body [l,m:list](l list m cons) : list->list->list.

Goal (l:list)<list>l=(app nil l).
By (automatic).
Save app_nil.

Goal (a:A)(l,m:list)<list>(cons a (app l m))=(app (cons a l) m).
By (automatic).
Save app_cons.

Goal (l,m,n : list)<list>(app (app l m) n)=(app l (app m n)).
Intros.
By (resolvew <<(list_initial l)>> THEN intros_auto).
By (REPEAT (elim <<app_cons>>)).
By (incomplet [3] <<(f_equal (cons x))>> THEN automatic).
Save app_ass.
#(add_resolve "app_ass").

Global mil.
Body [a:A][l,m:list](app l (cons a m)) : A->list->list->list.

(* Primitive recursion on lists *)
Section Primitive.

Variable B : Data.

Variable init : B.
Variable iter : A -> (list*B) -> B.

Local REC.
Body [l:list](list_REC l B init iter).

Local rec.
Body [l:list](list_rec l B init iter).

Goal (a:A)(l:list)<B>(iter a <list,B><l,(rec l)>)=(rec (cons a l)).
Intros.
Do (elim_with <<<B>(iter a (REC l))=(rec (cons a l))>> THEN automatic).
Do (incomplet [3]) (f_equal (iter a)).
Do (elim_with) <list*B>(<list,B><<list,B>Fst<(REC l)>,(rec l)>)=(REC l).
By (resolvew <<(list_initial l)>> THEN automatic).
Do (elim_with <<<list>l=<list,B>Fst<(REC l)>>> THEN automatic).
By (resolvew <<(list_initial l)>> THEN intros_auto).
By (elim_with
    <<<list>(cons x <list,B>Fst<(REC m)>)=<list,B>Fst<(REC (cons x m))>>>
    THEN automatic).
Do (incomplet [3]) (f_equal (cons x)).
Assumption.
Save rec_cons.

End Primitive.

Section Reclist.
Variable B : Data.

Variable init : B.
Variable iter : A -> list -> B -> B.

Definition reclist.
Body [l:list](list_rec l B init [a:A][h:list*B](h B (iter a))) : list->B.

Goal <B>init=(reclist nil).
By automatic.
Save reclist_nil.

Goal (a:A)(l:list)<B>(iter a l (reclist l))=(reclist (cons a l)).
By (unfold) reclist.
By (intros THEN elim <<rec_cons>> THEN automatic).
Save reclist_cons.

End Reclist.
```

```
Definition tail.
Body (reclist list nil [a:A][m,t:list]m) : list->list.

Goal (a:A)(l:list)<list>l=(tail (cons a l)).
By (intros THEN unfold "tail" THEN elim <<reclist_cons>> THEN automatic).
Save tail_cons.
#(add_resolve "tail_cons").

(* L'ordre des longueurs sur les listes *)

Definition length.
Body [l:list](l nat O [a:A]S) : list->nat.

Section length_order.
Definition lel [l,m:list](le (length l) (length m)).

Variables a,b:A.
Variable l,m,n:list.

    Theorem lel_refl (lel l l)
    Proof (le_n (length l)).

    Theorem lel_trans (lel l m)->(lel m n)->(lel l n)
    Proof (le_trans (length l) (length m) (length n)).

    Theorem lel_cons_cons (lel l m)->(lel (cons a l) (cons b m))
    Proof (le_n_S (length l) (length m)).

    Theorem lel_cons (lel l m)->(lel l (cons b m))
    Proof (le_S (length l) (length m)).

    Theorem lel_tail (lel (cons a l) (cons b m)) -> (lel l m)
    Proof (le_S_n (length l) (length m)).

End length_order.

Definition Null [l:list]<list>nil=l.

Theorem lel_nil (l:list)(lel l nil)->(Null l)
Proof [l:list]
    (list_initial l [m:list](lel m nil)->(Null m)
            [h:(lel nil nil)](refl_equal list nil)
            [a:A][m:list][h:(lel m nil)->(Null m)]
            [h':(lel (cons a m) nil)]
            (le_Sn_O (length m) h' (Null (cons a m)))).


Theorem nil_cons (a:A)(m:list)~(Null (cons a m))
Proof [a:A][m:list][h:(Null (cons a m))]
        (O_S (length m) (f_equal list nat length nil (cons a m) h)).

End Lists_with_parameter.

#(add_resolution ["tail_cons";"lel_refl";"lel_cons_cons";"lel_cons";"lel_nil";
                  "nil_cons"]).
```

```
(****************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3     *)
(****************************************************************************)
(*                                                                          *)
(*            Logical Relations : basic notions for Reynolds paradox        *)
(*                                                                          *)
(****************************************************************************)

(* use the Type:Type system, #use"Reyn_tac" *)


Goal (A,B:Prop)A->(A->B)->B.
Intros.
Exact (H0 H).
Save cut.

(* First, some general definition about relations *)

Inductive Definition equiv [A,B:Prop]:Prop  =
        equiv_intro : (A->B)->(B->A)->(equiv A B).

Section relation.

Definition Rel.
Body [A:Type]A->A->Prop.

Definition sym.
Body [A:Type][R:(Rel A)](x,y:A)(R x y)->(R y x).

Definition trans.
Body [A:Type][R:(Rel A)](x,y,z:A)(R x y)->(R y z)->(R x z).

Definition inclus.
Body [A:Type][R,S:(Rel A)](x,y:A)(R x y)->(S x y).

End relation.

Inductive Definition per [A:Type;R:(Rel A)]:Prop =
    per_intro : (sym A R)->(trans A R)->(per A R).

(* equiv is a partial equivalence relation over Prop *)

Goal (per Prop equiv).
Do res_simpl per_intro.
Do res_simpl H.
Do res_simpl equiv_intro.
Do res_simpl H.
Do res_simpl H0.
Do res_simpl equiv_intro.
Exact (H3 (H1 H5)).
Exact (H2 (H4 H5)).
Save per_equiv.

Section logical_relation.

Definition exp : (A:Type)(Rel A)->(B:Type)(Rel B)->(Rel (A->B)) =
    [A:Type][R:(Rel A)][B:Type][S:(Rel B)][f,g:A->B]
    (x,y:A)(R x y)->(S (f x) (g y)).

Definition power : (A:Type)(Rel A)->(Rel (A->Prop)) =
[A:Type][R:(Rel A)](exp A R Prop equiv).

End logical_relation.
```

```
(***************************************************************************)
(*    Projet Formel - Calculus of Constructions V4.10 - Vernacular V2.3    *)
(***************************************************************************)
(*                                                                         *)
(*             Reynolds paradox, with the Type:Type axiom                  *)
(*                                                                         *)
(***************************************************************************)

(* use the Type:Type system, #use"Reyn_tac";V"Log_Rel" *)

(* this file shows an inconsistency is the logical system U-, defined in Girard's
   thesis, and in which the question of the consistency of U- was raised.
   The system U- may be described with the axioms

        (star,star), (k,star), (k,k), (k',k)

   in Barendregt-Berardi's GTS. The reader can check that we use only these axioms
   in the derivation, and not fully the Type:Type axiom.
   The lambda-term we get does not loop to itself by reduction *)

(* Reynolds operator *)

Definition PHI.
Body [A:Type](A->Prop)->Prop.

(* we extend this map functorialy *)

Definition phi:(A,B:Type)(A->B)->(PHI A)->(PHI B) =
[A,B:Type][f:A->B][z:(PHI A)][u:B->Prop](z [x:A](u (f x))).

(* preinitial PHI-algebra. We need the axiom (Type,Type) *)

Definition A0.
Body (A:Type)((PHI A)->A)->A.

Definition iter_A0.
Body [X:Type][f:(PHI X)->X][u:A0](u X f).

Definition intro : (PHI A0)->A0 =
[z:(PHI A0)][A:Type][f:(PHI A)->A](f (phi A0 A (iter_A0 A f) z)).

(* extension of PHI to relations. We can thus consider PHI as a functor
   on sets, that are types with a relations *)

Definition phi2 : (A:Type)(Rel A)->(Rel (PHI A)) =
[A:Type][R:(Rel A)](power (A->Prop) (power A R)).

(* partial equivalence relation defined on A0, so that the set A0,E0
   is an initial PHI-algebra in the category of sets *)

Definition teta : A0 -> (PHI A0) =
        (iter_A0 (PHI A0) (phi (PHI A0) A0 intro)).

Definition E0 : (Rel A0) = [x1,x2:A0](E:(Rel A0))
     (per A0 E) ->
     ((z1,z2:(PHI A0))(phi2 A0 E z1 z2)->(E (intro z1) (intro z2))) ->
     ((x1,x2:A0)(E x1 x2)->(E x1 (intro (teta x2)))) ->
     (E x1 x2).

Definition F0 : (Rel (A0->Prop)) = (power A0 E0).

Definition G0 : (Rel (PHI A0)) = (power (A0->Prop) F0).

(* the goal of what follows is to show that the set A0,E0 is in one-to-one
   correspondance with the set (PHI A0),(phi2 A0 E0), via intro,teta.
   From this, we deduce a contradiction via Cantor-Russell's argument *)

Goal (sym A0 E0).
By reds.
Do res_simpl H0.
Do res_simpl H3.
Do res_simpl H.
Save sym_E0.

Goal (trans A0 E0).
By reds.
Do res_simpl H1.
Do res_exact H5 y.
Do res_simpl H.
Do res_simpl H0.
Save trans_E0.

Goal (per A0 E0).
Resolve per_intro.
Exact sym_E0.
Exact trans_E0.
Save per_E0.

(* intro is a map from (PHI A0),(phi2 A0 E0) to A0,E0 *)

Goal (z1,z2:(PHI A0))(G0 z1 z2)->(E0 (intro z1) (intro z2)).
By reds.
Do res_simpl H1.
Do res_simpl H.
```

```
Do res_simpl H3.
Do res_simpl H4.
Save lemma1.

Goal (per (A0->Prop) F0).
Unfold F0.
Unfold power.
Do res_simpl per_E0.
Do res_simpl per_equiv.
Resolve per_intro.
By (hyps THEN hyps).
Resolve H1.
Resolve H3.
Do res_simpl H.
By (hyps THEN hyps).
Do res_exact H2 (y y0).
Do res_simpl H3.
Resolve H4.
Do res_exact H0 x0.
Do res_simpl H.
Save per_F0.


Goal (per (PHI A0) G0).
Unfold G0.
Unfold power.
Do res_simpl per_F0.
Do res_simpl per_equiv.
Resolve per_intro.
By (hyps THEN hyps).
Resolve H1.
Resolve H3.
Do res_simpl H.
By (hyps THEN hyps).
Do res_exact H2 (y y0).
Do res_simpl H3.
Resolve H4.
Do res_exact H0 x0.
Do res_simpl H.
Save per_G0.

Goal (x1,x2:A0)(E0 x1 x2)->(E0 x1 (intro (teta x2))).
By (hyps THEN hyps).
Do res_simpl H2.
Do res_simpl H.
Save id_intro_teta.

Goal (z1,z2:(PHI A0))(G0 z1 z2)->(G0 z1 (teta (intro z2))).
By (hyps THEN hyps).
Change (equiv (z1 x) (z2 [x:A0](y (intro (teta x))))).
Do res_simpl H.
Do res_intro H0.
Do res_simpl id_intro_teta.
Save id_teta_intro.

Goal (x1,x2:A0)(E0 x1 x2)->(G0 (teta x1) (teta x2)).
By hyps.
Change ([u,v:A0](G0 (teta u) (teta v)) x1 x2).
Resolve H.
Do res_simpl per_G0.
Do res_simpl per_intro.
Do res_simpl H0.
Do res_exact H1 (teta y).
By (hyps THEN hyps).
Change (equiv (z1 [u:A0](x (intro (teta u))))
          (z2 [u:A0](y (intro (teta u))))).
Do res_simpl H0.
Resolve H1.
Do res_simpl lemma1.
By hyps.
Do res_simpl id_teta_intro.
Save lemma_teta.

Definition psi : (A0->Prop)->A0 =
[u:A0->Prop](intro (F0 u)).

Definition inter : (PHI A0)->A0->Prop =
[C:(PHI A0)][x:A0](P:A0->Prop)(F0 P P)->(C P)->(P x).

Goal (z1,z2:(PHI A0))(G0 z1 z2)->(F0 (inter z1) (inter z2)).
By (hyps THEN hyps).
Do res_intro equiv_intro.

By hyps.
Do cut (equiv (P x) (P y)).
Do res_intro H4.
Resolve H5.
Do res_simpl H1.
Do cut (equiv (z1 P) (z2 P)).
Do res_intro H7.
Exact (H9 H3).
Do res_simpl H.
Do res_simpl H2.
```

```
By hyps.
Do cut (equiv (P x) (P y)).
Do res_intro H4.
Resolve H6.
Do res_simpl H1.
Do cut (equiv (z1 P) (z2 P)).
Do res_intro H7.
Exact (H8 H3).
Do res_simpl H.
Do res_simpl H2.
Save lemma_inter.

(* we follow Cantor-Russell's paradox *)

Definition khi : A0 -> (A0->Prop) = [x:A0](inter (teta x)).

Section paradox.

Variable p:Prop.

Definition u0: A0->Prop = [x:A0](khi x x)->p.

Definition x0: A0 = (psi u0).

Goal (E0 x0 x0).
Unfold x0.
Unfold x0.
Unfold psi.
Unfold psi.
Do res_simpl lemma1.
Do res_simpl per_F0.
Do res_simpl equiv_intro.
Do res_exact H1 x.
Do res_exact H1 y.
Do res_exact H0.
Save lemma4.

Goal (F0 u0 u0).
By hyps.
Do cut (F0 (khi x) (khi y)).
Do cut (equiv (khi x x) (khi y y)).
Do res_simpl H1.
Unfold u0.
Unfold u0.
Do res_simpl equiv_intro.
Exact (H4 (H3 H5)).
Exact (H4 (H2 H5)).
Do res_simpl H0.
Unfold khi.
Unfold khi.
Resolve lemma_inter.
Do res_simpl lemma_teta.
Save lemma3.

Goal (F0 u0 (khi x0)).
Unfold x0.
Unfold khi.
Unfold psi.
Do apply per_F0.
Do res_simpl H.
Do res_exact H1 (inter (F0 u0)).
By hyps.
Do res_simpl equiv_intro.

By hyps.
Do cut (equiv (u0 x) (P y)).
Do res_simpl H6.
Exact (H7 H3).
Do res_simpl H5.

Do apply lemma3.
Do cut (equiv (u0 x) (u0 y)).
Do res_simpl H5.
Resolve H7.
Do res_simpl H3.
Do res_simpl H4.

Do res_simpl lemma_inter.
Do cut (G0 (F0 u0) (teta (intro (F0 u0)))).
Do res_simpl H3.
Do res_simpl id_teta_intro.
Do res_simpl equiv_intro.

Do res_exact H1 x0.
Do res_exact H1 y0.
Do res_simpl H0.

Save lemma5.

Goal (equiv (u0 x0) (khi x0 x0)).
Do apply lemma5.
Resolve H.
Exact lemma4.
Save lemma6.
```

```
Goal (u0 x0).
By hyps.
Do res_simpl lemma6.
Exact (H1 H H).
Save lemma7.

Goal p.
Do res_simpl lemma6.
Exact (lemma7 (H lemma7)).
Save Reynolds.
```

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique