Constructions: A Higher Order Proof System for Mechanizing Mathematics

Thierry Coquand
Gérard Huet

INRIA
Domaine de Voluceau
78150 Rocquencourt
France

ABSTRACT

We present an extensive set of mathematical propositions and proofs in order to demonstrate the power of expression of the theory of constructions.

Introduction

The theory of constructions is a higher order calculus inspired from the work of De Bruijn[11,12], Girard[23,22] and Martin-Löf[39]. It is introduced and motivated in Coquand-Huet[17] and its consistency is proved in Coquand[16]. A prototype implementation in ML has been developed at INRIA for experimentation with the power of expression of the calculus.

We present here an annotated transcript of examples developed on this system. ML is a functional programming language developed originally as the meta-language of the LCF proof assistant[24]. Actually no previous knowledge of ML is required here, since only a few very specific functions are used in the examples.

1. An overview of the language of constructions.

1.1. Constructions: contexts and objects

Constructions are well-typed expressions of a typed lambda-calculus, where the types are lambda-expressions of the same nature. Thus our basic language is basically Nederpelt's Λ[41,42,18]. There are four rules of formation:

	Universe
$[x{:}M]N$	Abstraction
$(M\ N)$	Application
x	Variable

In the formation rule for abstraction, we prefer the Automath notation $[x{:}M]N$ to the more traditional notation $\lambda x_M{\cdot}N$ for two reasons. Firstly, the type M associated to the bound variable x may be quite complex, and thus the subscript notation would be too messy, with subscripting at any level. Secondly, this binding operation is used for representing products $\forall x{\in}M{\cdot}N$ as well as functions $\lambda x{\in}M{\cdot}N$. The name x is of course completely irrelevant, and belongs only to the concrete syntax of the term. Abstractly, the

abstraction operator is binary, and admits two components, M and N. Occurrences of variable x in the concrete syntax of term N will be replaced by its de Bruijn's index, i.e. an integer denoting the reference depth of the occurrence. Thus the string

$$[x{:}A]([y{:}B](x\ y)x)$$

represents concretely the abstract term

$$[A]([B](2\ 1)1)$$

That is, integer n denotes the variable bound at the nth binder upward in the term. As usual in combinatory logic we write $(M\ N)$ for the application of term N to term M.

Our term algebra is completed by a constant $*$, which plays the role of the universe of all types. In Automath languages, $*$ is noted τ, ***prop*** or ***type***.

We shall not explain further the abstract syntax used in the formal manipulation of the constructions. We shall only describe here the concrete syntax, which reflects a richer structure of abbreviations.

First we establish a strong distinction between expressions of the form

$$[x_1{:}M_1][x_2{:}M_2]\ \cdots\ [x_n{:}M_n]^*$$

which we shall call ***contexts***, from all other expressions, which we shall call ***objects***. Intuitively, the contexts are declarations used for linguistic definitions. An object of type the context above is a proposition with free variables of types $M_1, \cdots, M_n$. For instance, the context $[x{:}nat]^*$ is the type of unary predicates over type ***nat***. In order to stress the distinction between contexts and objects, we shall in the following adopt the alternative syntax $\{x \mid M\}N$ for abstraction over a context type M. For instance, we shall declare a unary predicate P over ***nat*** in an expression N by the construction $\{P \mid [x{:}A]^*\}N$.

1.2. Types: propositions and proofs

We shall not in this paper enter the technical details of type-checking in the theory of constructions. The interested reader will find a full technical account of the theory in Coquand's thesis[16]. We assume known the notion of β-reduction, which corresponds to computation in the calculus of constructions. This relation is confluent (i.e. has the Church-Rosser property), noetherian (i.e. strong normalization holds), and preserves the types of expressions. Its equivalence closure is called ***conversion***. All is needed here concerning type-checking is the understanding that an application $(M\ N)$ is well typed only when the expression M has a functional type $[x{:}A]B$ compatible with the type A' of expression N. Compatible means usually that A and A' are equivalent modulo conversion. The resulting type of the application is B, where occurrences of x are replaced by N. This explanation is not quite sufficient, since we also allow for instance $\{A \mid *\}M$ to be applied to P of type $[x{:}nat]^*$. The rationale of this type coercion is to allow the predicate P to stand for the proposition $\forall x \in nat \cdot P(x)$ as well as for a proposition schema with a subterm x of type ***nat***. This departure from the Automath formalisms is of course essential for the expressiveness of the calculus, as the examples below will demonstrate.

We shall call ***proposition*** an object of type a context, and ***proof*** an object of type a proposition. We shall not consider further objects, and therefore restrict abstraction to contexts and propositions. For instance, there are no functionals mapping proofs into proofs. This restriction of the full calculus

presented in Coquand-Huet[17] is in agreement with the theory developed and proved consistent in Coquand's thesis[16]. All examples below belong to this simpler three-level system.

1.3. Abbreviations

We use in our concrete syntax two abbreviations. First, we allow to abbreviate $\{x \mid *\}N$ as $!x.N$. This abbreviation may be iterated, like in $!x,y,z.N$. The symbol ! must be understood as universally quantifying over all propositions. The rationale of this notation lies in the subtype discipline explained above: variable x of type $*$ may be bound by an expression of type an arbitrary context, considered as a quantification prefix. This gives the constant $*$ the role of a free variable standing for an arbitrary proposition.

The second abbreviation is to allow expression $A \to B$ in place of $[x:A]B$, whenever x does not occur in B. The expression $A \to B$, seen as a type, is the type of ordinary functions from A to B. A *dependent* type $[x:A](P\ x)$ is the type of generalized functional objects of domain A, mapping value X in A to some value in $(P\ X)$. Such dependant types appear for instance in Martin-Löf's intuitionistic theory of types[39]. They should not be altogether unfamiliar to computer scientists, though, if one thinks of an Algol procedure with an integer parameter n and returning an array of dimension n.

If one sees $A \to B$ as a proposition rather than as a type, the arrow $\longrightarrow$ may be understood as an (intuitionistic) implication. One can then see the abbreviations ! and $\longrightarrow$ as pointing out as important subcases of constructions the two type constructors of Girard's second order types[23,22].

A few other less important abbreviation mechanisms have been defined. For instance, the *let* construction of ISWIM and ML is allowed. This permits to simplify complex constructions with nested multiple occurrences of some expression X to be abbreviated $let\ x = X\ in\ M_x$ rather than either the expanded form M_X, or the hard-to-read redex $([x:A]M_x\ X)$ which further requires a redundant type A instead of naturally defaulting it to the type of X.

The usual notation of combinatory logic for multiple applications is used, allowing $(A\ B\ C)$ instead of $((A\ B)\ C)$. Also, $\longrightarrow$ associates to the right; thus $A \to B \to C$ abbreviates $[u:A][v:B]C$.

Some constructions are hard to decipher, when too much polymorphism is involved. The polymorphic instanciations needed to get the type right interfere with the real arguments to functional objects. Since very often polymorphic arguments occur before real arguments, it is allowed to group them initially, by abbreviating for instance $(F\ A_1\ A_2\ x)$ into $<A_1,A_2>F(x)$. Note that since the parser does not care about the types of things, this abbreviation may be used for all arguments, allowing the Automath notation $<N>M$ in place of the combinatory logic notation $(M\ N)$.

In the implementation, the concrete syntax of constructions is defined by a Yacc grammar, whose semantic actions generate ML values under the form of trees of abstract syntax. In the examples below, an expression between double quotes "..." is parsed by this specialized parser, and thus denotes an ML value of type *context* or *object* for propositions and proofs.

1.4. Constants

The basic language of constructions is enriched by constants, denoted by identifiers. Every well-typed construction may be named, and later on referred to by that name. The interaction loop of our system is ML's top level, in an environment where a parser, a pretty-printer and a type-checker for constructions are available. Three concrete types are known: ***context***, ***object***, and their sum ***constr=context+object***. A few commands are provided for declaring constants. Such commands may be grouped together in an ML module, providing the user with a rudimentary theory system. Here are the predefined commands:

```
PROP: string -> object -> void
LET: string -> object -> object -> void
LET_SYNTAX: string -> string list -> void
```

The command *PROP* '*p*' *obj* declares a proposition constant *p* corresponding to object *obj*. The construction *obj* is type-checked, and it is verified that its type is a context. Similarly, *LET* '*foo*' *proof prop* type-checks *proof*, verifies that its type is equal (i.e. λ-convertible) to *prop*, and enters it in the current theory under name *foo*. Note that *LET* means "prove" here, in the propositions-as-types isomorphism: we have verified that *prop* had a proof, namely *proof*. Our system may thus be seen as a type-checker for constructive mathematics expressed by constructions: the *PROP* command verifies that a proposition is syntactically meaningful in the current logical language, and the *LET* command verifies that a proposition is true, in the intuitionistic sense of having a proof. Note that we assume no a priori logical connectives, axioms or inference rules. Our system is thus fundamentally different in spirit from inference systems such as LCF's PPλ[25] or Martin-Löf's intuitionistic theory of types[39], although it bears a close relationship with Martin-Löf's earlier theory of types[35].

Finally, a command such as *LET_SYNTAX* '*cond*' [*'if'*;*'then'*;*'else'*;*'fi'*] allows the constant *cond* to be pretty-printed with a mixfix syntax whenever it is applied to enough arguments (3 in the *cond* example).

This completes the description of our prototype implementation. Actually, we must admit a little cheating on the parser's part: the parser knows beforehand about the mixfix syntax of a few constants. This is because we do not know how to modify dynamically the tables of the parser generated by Yacc. This is an unimportant technical detail the reader need not be concerned about, since our parser is consistent with all the *LET_SYNTAX* commands.

2. Logical constructions

Here we start Mathematics from scratch. The purpose of this section is to define the intuitionistic connectives. Note: from now on our text is a commented transcript of a computer session. All the examples given have been machine-checked.

2.1. Arrow

```
Internalizing the -> abbreviation: intuitionistic implication.
let ARROW = "!A,B.[x:A]B";;
PROP '->' ARROW;;
LET_SYNTAX '->' ['('; '->'; ')'];; We write A→B.

Self: -> is reflexive.
```

```
let SELF = "!A.A->A";;
PROP 'self' SELF;;
```

Identity, the single element in self.

```
let ID = "!A.[u:A]u";;
LET 'Id' ID "self";;
```

I.e., Id proves proposition self.

```
LET_SYNTAX 'Id' ['<';'>Id'];;
```

In order to note Id_A as <A>Id.

Modus Ponens = "!A,B.(A->B)->A->B".

```
let MP = "!A,B.(self (A->B))";;
```

Apply = "!A,B.[f:A->B][x:A](f x)".

```
LET 'apply' "!A,B.<A->B>Id" MP;;
```

Application is just functional identity. It proves modus ponens.

Skip.

```
let SKIP = "!A,B.A->B->A";;
LET 'K' "!A,B.[x:A](B->x)" SKIP;;
```

The K constant combinator.

Schonfinkel's generalized composition.

```
let SCHON = "!A,B,C.(A->B->C)->(A->B)->A->C";;
LET 'SC' "!A,B,C.[f:A->B->C][g:A->B][x:A](f x (g x))" SCHON;;
```

Note that K, SC and apply give combinatory logic, the calculus of proofs of implicational intuitionistic logic, whose axioms in Hilbert form are SKIP and SCHON, with MP the sole rule of inference.

The fundamental commutativity of non-dependant hypotheses.

```
let PERMUTE = "!A,B,C.(A->B->C)->(B->A->C)";;
LET 'permute' "!A,B,C.[f:A->B->C][y:B][x:A](f x y)" PERMUTE;;
```

Cut: -> is transitive.

```
let CUT = "!A,B,C.(A->B)->(B->C)->A->C";;
```

Cut is proved by functional composition.

```
let PIPE = "!A,B,C.[f:A->B][g:B->C][x:A](g (f x))";;
LET '|' PIPE CUT;;
LET_SYNTAX '|' ['<';',';';';'>';'|';''];;
```

Note that this is categorical composition, good for following arrows. Its name and syntax "f|g" are derived from the Unix pipe operation on streams.

2.2. Products

We follow Prawitz in the definition of product[50].

```
let PROD = "!A,B,C.(A->B->C)->C";;
```

This may seem like coding-up, and in a way it is. The thread of thought to follow in these very foundational definitions is to look for the operational meaning of the concept one wants to define. For instance, how does one actually USE the product A&B? We can use it to prove any proposition C, given a proof that A and B implies C, i.e. given an element of A->B->C. This is precisely what

PROD says. Here product and conjunction are one and the same concept, whence the & notation below.

```
PROP '&' PROD;;
LET_SYNTAX '&' ['(';'&';')'];;

let and_intro = "!A,B.A->B->A&B";;
```

Pairing : the proof of and_intro. Note how the standard rules of inference of natural deduction map into propositions which admit a proof.

```
let PAIR = "!A,B.[x:A][y:B]!C.[z:A->B->C](z x y)";;
```

Writing out the desired type and_intro of PAIR gives directly its context "!A,B.[x:A][y:B]!C.[z:A->B->C]?". The body (z x y) may be synthesized easily by a simple PROLOG-like backtrack search. This gives hopes to extend the current proof-checker into a more ambitious theorem-prover.

```
LET '<>' PAIR and_intro;;
LET_SYNTAX '<>' ['<';',';'>('; ';')'];;
```

We may of course use all the power of our meta-language ML in order to macro-generate abstract syntax trees of constructions independently from parsing. For instance, using the "apl" function which iterates application on its list argument, we may define:

```
let Pair (obj1,obj2) = apl[ref '<>';A1;A2;obj1;obj2] where
     A1=Type obj1 and A2=Type obj2;;

letrec Tuple lobj = Pair (hd lobj, let ob.lobj' = tl lobj in
     if null lobj' then ob else Tuple (ob.lobj'));;
```

Now for the projections.

```
let and_elim_left = "!A,B.(A&B)->A";;

let FST = "!A,B.[x:A&B](x A [y:A][z:B]y)";;
```

1st proj.

```
LET 'fst' FST and_elim_left;;
LET_SYNTAX 'fst' ['<';',';'>fst(';')'];;

let and_elim_right = "!A,B.(A&B)->B";;

let SND = "!A,B.[x:A&B](x B [y:A][z:B]z)";;
```

2nd proj.

```
LET 'snd' SND and_elim_right;;
LET_SYNTAX 'snd' ['<';',';'>snd(';')'];;

let DEDUCTION_LEFT = "!A,B,C.((A&B)->C)->A->B->C";;
```

Currying.

```
let CURRY = "!A,B,C.[x:(A&B)->C][y:A][z:B](x <A,B>(y,z))";;
LET 'curry' CURRY DEDUCTION_LEFT;;

let DEDUCTION_RIGHT = "!A,B,C.(A->B->C)->(A&B)->C";;
```

Uncurrying.
```
let UNCURRY = "!A,B,C.[x:A->B->C][y:A&B](x <A,B>fst(y) <A,B>snd(y))";;
LET 'uncurry' UNCURRY DEDUCTION_RIGHT;;
```

Equivalence as isomorphism.
```
let EQUIV = "!A,B.(A->B)&(B->A)";;
PROP '<->' EQUIV;;
LET_SYNTAX '<->' ['(';'<->';')'];;

let equiv_intro = "!A,B.[f:A->B][g:B->A]<A->B,B->A>(f,g)";;
LET 'iso' equiv_intro "!A,B.(A->B)->(B->A)->(A<->B)";;
```

It is easy to prove the following isomorphisms:
&-commutative: (A&B)<->(B&A)
&-associative: (A&(B&C))<->((A&B)&C)
<->-commutative: (A<->B)<->(B<->A)
<->-associative: (A<->(B<->C))<->((A<->B)<->C)

The deduction theorem.
```
let DEDUCTION = "!A,B,C.((A&B)->C)<->(A->B->C)";;

LET 'deduction'
    "!A,B,C.(iso ((A&B)->C) (A->B->C) (curry A B C) (uncurry A B C))"
    DEDUCTION;;
```

2.3. Sums

```
let SUM = "!A,B,C.(A->C)->(B->C)->C";;
```
The sum, or intuitionist disjunction, of A and B, is a way to prove any C from a proof of C from A and a proof of C from B.

```
PROP '+' SUM;;
LET_SYNTAX '+' ['(';'+';')'];;
```

Left injection.
```
let sum_intro_left ="!A,B.A->(A+B)";;
let INL = "!A,B.[x:A]!C.[y:A->C][z:B->C](y x)";;
LET 'inl' INL sum_intro_left;;
```

Right injection.
```
let sum_intro_right = "!A,B.B->(A+B)";;
let INR = "!A,B.[x:B]!C.[y:A->C][z:B->C](z x)";;
LET 'inr' INR sum_intro_right;;

let sum_elim = "!A,B.(A+B)->!C.(A->C)->(B->C)->C";;
```

Reasoning by cases.
```
let CASE = "!A,B.[x:A+B]!C.[u:A->C][v:B->C](x C u v)";;
LET 'case' CASE sum_elim;;
```

From the above we get easily:
+-commutative: (A+B)<->(B+A)
+-associative: (A+(B+C))<->((A+B)+C)

2.4. Quantifiers as general product and sum

Universal quantification, or general product.
```
let PI = "!A.{P|A->*}[x:A](P x)";;
PROP 'Pi' PI;;
LET_SYNTAX 'Pi' ['<';'>Pi(';')'];;
```

Instanciation.
```
let Pi_elim = "!A.{P|A->*}[x:A]<A>Pi(P)->(P x)";;
let INST = "!A.{P|A->*}[x:A][p:<A>Pi(P)](p x)";;
LET 'inst' INST Pi_elim;;
```

Universal Generalization.
```
let Pi_intro = "!A.{P|A->*}!B.([x:A]B->(P x))->B-><A>Pi(P)";;
let GEN = "!A.{P|A->*}!B.[f:[x:A]B->(P x)][y:B][x:A](f x y)";;
LET 'gen' GEN Pi_intro;;
```

Existential quantification, or general sum
```
let SIGMA = "!A.{P|A->*}!B.([x:A](P x)->B)->B";;
PROP 'Sig' SIGMA;;
LET_SYNTAX 'Sig' ['<';'>Sig(';')'];;
```

Existential Introduction.
```
let Sig_intro = "!A.{P|A->*}[x:A](P x)-><A>Sig(P)";;
let EXIST = "!A.{P|A->*}[x:A][y:(P x)]!B.[f:[x:A](P x)->B](f x y)";;
LET 'exist' EXIST Sig_intro;;
```
This permits to use a predicate over A as a space over A.

Projection.
```
let Sig_elim = "!A.{P|A->*}<A>Sig(P)->A";;
let WITNESS = "!A.{P|A->*}[p:<A>Sig(P)](p A [x:A](P x)->x)";;
LET 'witness' WITNESS Sig_elim;;
```

Note that this witness is weaker than an epsilon operator: we can't access the proof component justifying that it verifies P. Thus our sum is fundamentally weaker than the one in Martin-Löf's intuitionist theory of types[39].

3. Classical Logic constructions

We are not obliged to stick to intuitionistic connectives, and may define the classical connectives as well, using the standard embedding of classical proofs as intuitionistic refutations.

3.1. Falsity and Negation.

```
let FALSITY = "!A.A";;
```
Falsity as meaningless; there is no proof of "!A.A".
```
PROP '{}' FALSITY;;
```

Negation.
```
let NOT = "!A.A->{}";;
PROP '~' NOT;;
```

```
LET_SYNTAX '~' ['~';''];;

{} implies every proposition.
let CONTR = "!A.{}->A";;
LET 'contr' "!A.[u:{}](u A)" CONTR;;

Classical truth : A statement is true iff it is not meaningless.
let TRUTH = "~{}";;
LET 'truth' "<{}>Id" TRUTH;;

[A] is the construction of the classical meaning of proposition A.
let CLOSURE = "!A.~(~A)";;
PROP '[]' CLOSURE;;
LET_SYNTAX '[]' ['[';']'];;

It is easy to show "!A.~[A]<->[~A]"

let CLOSE = "!A.A->[A]";;
LET 'close' "!A.[x:A][h:~A](h x)" CLOSE;;

Closed (i.e. classical) propositions.
let CLOSED = "!A.[A]->A";;
PROP 'closed' CLOSED;;

{} is closed.
LET 'empty_closed' "[hyp:[{}]](hyp truth)" "(closed {})";;

Negation gives closed propositions.
LET 'neg_closed' "!A.[f:[~A]][x:A](f (close A x))" "!A.(closed (~A))";;

Hence the closure of a proposition is closed.
LET 'closure_closed' "!A.(neg_closed (~A))" "!A.(closed [A])";;
```

And we get easily a proof of "!A.[[A]]<->[A]" . Similarly it is easy to prove that "!A,B.(closed (~A&~B))", and thus that closed propositions are preserved under product.

3.2. Classical disjunction.

```
Union as the closure of sum.
let UNION = "!A,B.[A+B]";;
PROP 'union' UNION;;

A definition easier to deal with: the truth-table disjunction.
let OR = "!A,B.~A->[B]";;
PROP '?' OR;;
LET_SYNTAX '?' ['(';'?';')'];;

Showing the isomorphism between the two definitions
LET 'union_to_or'
    "!A,B.[h:[A+B]][a':~A][b':~B] let u=[x:A+B](x {} a' b') in (h u)"
    "!A,B.[A+B]->(A?B)";;
```

```
LET 'or_to_union'
     "!A,B.[h:A?B][x:~(A+B)]
          let a'=[u:A](x (inl A B u)) in
          let b'=[v:B](x (inr A B v)) in (h a' b')"
     "!A,B.(A?B)->[A+B]";;
```

Identity on closures proves the law of excluded middle.

```
let EXCL_MIDDLE = "!A.(~A?A)";;
LET 'excl_middle' "!A.<[A]>Id" EXCL_MIDDLE;;

let SUM_TO_OR = "!A,B.(A+B)->(A?B)";;
LET 'sum_to_or'
     "!A,B.[u:A+B](union_to_or A B (close (A+B) u))" SUM_TO_OR;;

let OR_INTRO_LEFT = "!A,B.A->A?B";;
LET 'cinl' "!A,B.[x:A](sum_to_or A B (inl A B x))" OR_INTRO_LEFT;;

let OR_INTRO_RIGHT = "!A,B.B->A?B";;
LET 'cinr' "!A,B.[x:B](sum_to_or A B (inr A B x))" OR_INTRO_RIGHT;;
```

3.3. De Morgan's laws and Kuratowski's axioms.

```
LET 'not_or'
     "!A,B.[h:~(A?B)]!C.[f:~A->~B->C]
          let x=[u:A](h (cinl A B u)) in
          let y=[v:B](h (cinr A B v)) in (f x y)"
     "!A,B.~(A?B)->~A&~B";;

LET 'and_not'
     "!A,B.[h:~A&~B][f:A?B]
          let x=<~A,~B>fst(h) in
          let y=<~A,~B>snd(h) in (f x y)"
     "!A,B.(~A&~B)->~(A?B)";;

LET 'De_morgan' "!A,B.(iso ~(A?B) ~A&~B (not_or A B) (and_not A B))"
     "!A,B.~(A?B)<->~A&~B";;
```

This is an intuitionistic isomorphism, not just a classical equivalence.

```
LET 'or_not'
     "!A,B.[f:~A?~B][h:A&B]
          let x=<A,B>fst(h) in
          let y=<A,B>snd(h) in (f (close A x) (close B y))"
     "!A,B.(~A?~B)->~(A&B)";;
```

Beware! The reverse arrow is not intuitionically valid, and so we do not have the second De Morgan's law: "!A,B.~(A&B)<->~A?~B". It is also easy to show:
?-commutative: (A?B)<->(B?A)
?-associative: (A?(B?C))<->((A?B)?C)
     ~A?{}<->~A

As expected from the definition of "union", "A?B" is closed.

```
LET 'or_closed'
     "!A,B.[f:[A?B]][x:~A][y:~B](f [u:A?B](u x y))"
```

```
    "!A,B.(closed (A?B))";;
```

We now prove that or commutes with closure.

```
LET 'or_closure'
    "!A,B.[f:[A]?[B]][x:~A][y:~B](or_not ~A ~B f <~A,~B>(x,y))"
    "!A,B.([A]?[B])->A?B";;

LET 'closure_or'
    "!A,B.[f:A?B][x:[~A]][y:[~B]](f (neg_closed A x) (neg_closed B y))"
    "!A,B.A?B->([A]?[B])";;
```

Now it is easy to show that A?{}<->[A].

```
LET 'Kuratowski'
    "!A,B.(iso [A]?[B] A?B (or_closure A B) (closure_or A B))"
    "!A,B.([A]?[B])<->A?B";;
```

We now have, with close, empty_closed, closure_closed and Kuratowski, proofs of analogues to the Kuratowski axioms, stating that [] is a closure operation, and thus that the propositions form a topological space, in a constructive set theory where -> is inclusion, and ? is union. However here union gives always closed sets, and we do not have an isomorphism (A?{})<->A. This presentation can also be interpreted as building in classical logic in intuitionistic logic, with closure a modality operator. Note the duality between our closure and the classical necessity operator: [A] is to A what in classical modal logic B is to []B.

3.4. Classical implication.

```
let IMPLIES = "!A,B.A->[B]";;
PROP '=>' IMPLIES;;
LET_SYNTAX '=>' ['(';'=>';')'];;
```

Note that by definition A?B is (~A)=>B. We have also (~A?B)->(A=>B) but not the reverse arrow.

```
let IMPLY_INTRO = "!A,B.(A->B)->(A=>B)";;
LET 'imply_intro' "!A,B.[f:A->B][x:A][neg:~B](neg (f x))" IMPLY_INTRO;;

let IMPLY_ELIM = "!A,B.(A=>B)->(~B)->~A";;
LET 'imply_elim' "!A,B.[h:A=>B][neg:~B][x:A](h x neg)" IMPLY_ELIM;;

let NEG_NEG = "!A.[A]=>A";;
LET 'neg_neg' "excl_middle" NEG_NEG;;
```

Same proof as excluded middle!

A positive implicational tautology not provable intuitionistically: Pierce's law.

```
let PIERCE = "!A,B.((A=>B)=>A)=>A";;
LET 'pierce'
    "!A,B.[h1:(A=>B)=>A][h2:~A]
     let AimpB = [u:A][v:~B](h2 u) in (h1 AimpB h2)"
  PIERCE;;
```

This is the first non-trivial proof.

Equivalence.
```
PROP '<=>' "!A,B.(A=>B)&(B=>A)";;

PROP 'xor' "!A,B.~(A<=>B)";;
```

We could alternatively have defined the notions ? and => from:
```
let CONTRADICTORY = "!A,B.A->~B";;
PROP '#' CONTRADICTORY;;
LET_SYNTAX '#' ['(';'#';')'];;
```
It is easy to prove commutativity: "!A.B.(A#B)<->(B#A)" from permute. We may then define A=>B as A#~B and A?B as ~A#~B

3.5. Cantor's theorem.

As a little exercise in classical logic, let us show how to express the proof of Cantor's theorem (Epimenide's liar's paradox).

```
let CANTOR = "!A.{R|A->A->*}~({P|A->*}<A>Sig([x:A][y:A](R x y)<->(P y)))";;

let CANTOR_PROOF =
     "!A.{R|A->A->*}
     [hyp:{P|A->*}<A>Sig([x:A][y:A](R x y)<->(P y))]
     !B.(hyp ([x:A](R x x)->B) B
 ([x:A][iso:[y:A](R x y)<->((R y y)->B)]
  let f=[y:A]<(R x y)->(R y y)->B,((R y y)->B)->(R x y)>fst((iso y)) in
  let g=[y:A]<(R x y)->(R y y)->B,((R y y)->B)->(R x y)>snd((iso y)) in
  let diag=[p:(R x x)](f x p p) in (diag (g x diag))))";;
LET 'Cantor' CANTOR_PROOF CANTOR;;
```
QED

It is interesting to compare this proof with the resolution proof given in Huet[27]. It is fairly obvious that higher-order unification will be a key algorithm for the automated synthesis of such proofs.

4. Equality constructions.

We may *define* equality in Leibniz's fashion. Note that this does not use classical constructions. In particular, "->" is the intuitionistic implication underlying its use as an abbreviation.

4.1. Leibniz's equality.

```
let EQUAL = "!A.[x:A][y:A]{P|A->*}(P x)->(P y)";;
PROP '=' EQUAL;;
LET_SYNTAX '=' ['<';'>';'=';''];;
```
EQUAL = "!A.[x:A][y:A]<A>x=y". Note the infix notation.

4.2. Equality is a congruence relation.

Substitutivity is implicit in the definition of equality. All is left for showing that = is a congruence is that it is an equivalence.

Reflexivity proved by Identity.

```
let REFL_EQUAL = "!A.[x:A]<A>x=x";;
let PROOF_REFL_EQUAL = "!A.[x:A]{P|A->*}<(P x)>Id";;
LET 'refl_equal' PROOF_REFL_EQUAL REFL_EQUAL;;
```

Transitivity proved by Composition.

```
let TRANS_EQUAL = "!A.[x:A][y:A][z:A](<A>x=y)->(<A>y=z)->(<A>x=z)";;
let PROOF_TRANS_EQUAL =
      "!A.[x:A][y:A][z:A][p:<A>x=y][q:<A>y=z]{P|A->*}
      <(P x),(P y),(P z)> (p P)|(q P) ";;
LET 'trans_equal' PROOF_TRANS_EQUAL TRANS_EQUAL;;
```

Symmetry proved by trick.

```
let SYM_EQUAL = "!A.[x:A][y:A](<A>x=y)->(<A>y=x)";;
let PROOF_SYM_EQUAL =
 "!A.[x:A][y:A][p:<A>x=y]{P|A->*}(p [z:A](P z)->(P x) <(P x)>Id)";;
LET 'sym_equal' PROOF_SYM_EQUAL SYM_EQUAL;;
```

4.3. Generalizing to properties of any binary polymorphic relation.

```
let REFL = "{R|!A.A->A->*}!A.[x:A](R A x x)"
and TRANS = "{R|!A.A->A->*}!A.[x:A][y:A][z:A](R A x y)->(R A y z)->(R A x z)"
and SYM ="{R|!A.A->A->*}!A.[x:A][y:A](R A x y)->(R A y x)";;

PROP 'refl' REFL;PROP 'trans' TRANS;PROP 'sym' SYM;;

let EQUIV = "{R|!A.A->A->*}(refl R)&(trans R)&(sym R)";;
```

Let R be a polymorphic binary relation. R is equiv iff R is refl and R is trans and R is sym. Now that we have defined the elementary notions, more complex notions can be expressed in a natural manner.

```
PROP 'equiv' EQUIV;;
```

For instance, = is equiv.

```
let PROOF_EQUIV_EQUAL =
      Tuple[PROOF_REFL_EQUAL;PROOF_TRANS_EQUAL;PROOF_SYM_EQUAL];;
LET 'equiv_equal' PROOF_EQUIV_EQUAL "(equiv =)";;
```

4.4. Andrews' lemma

As an exercise on equality theorem-proving, let us the following property given in Andrews[1]. Theorem: If some iterate of a function admits a unique fixpoint, then the function admits a fixpoint.

Fixpoint.

```
let FIXPT = "!A.[f:A->A][x:A]<A>(f x)=x";;
PROP 'fixpt' FIXPT;;
```

Commutation.

```
let COMMUTE = "!A.[f:A->A][g:A->A][x:A]<A> (g (f x)) = (f (g x))";;
PROP 'commute' COMMUTE;;
```

Unicity.

```
let UNIQUE = "!A.{P|A->*}[x:A][y:A](P x)->(P y)-><A>x=y";;
PROP 'unique' UNIQUE;;
```

If f and g commute, and if g admits a unique fixpoint, then f admits a fixpoint.

```
let LEMMA1 = "!A.[f:A->A][g:A->A] (commute A f g) ->
 <A>Sig((fixpt A g)) -> (unique A (fixpt A g)) -> <A>Sig((fixpt A f))";;

let LEMMA1_PROOF = "!A.[f:A->A][g:A->A][com:(commute A f g)]
   [fix:<A>Sig((fixpt A g))]
   [uni:(unique A (fixpt A g))]
   !B.[h:[x:A](fixpt A f x)->B]
   (fix B [a:A][h':(fixpt A g a)]
      (h a (uni (f a) a
       {P|A->*}[p:(P (g (f a)))](h' [u:A](P (f u)) (com a P p))  h')))";;
LET 'l1' LEMMA1_PROOF LEMMA1;;
```

Inductive definition of power.

```
let POWER = "!A.[f:A->A][g:A->A]
   {P|(A->A)->*}(P f)->([h:A->A](P h)->(P [x:A](f (h x))))->(P g)";;
PROP 'power' POWER;;
```

Commutation is reflexive.

```
let COMMUT_REFL = "!A.[f:A->A](commute A f f)";;
let COMMUT_REFL_PROOF = "!A.[f:A->A][x:A](refl_equal A (f (f x)))";;
LET 'power0' COMMUT_REFL_PROOF COMMUT_REFL;;
```

Commutation is preserved by iteration.

```
let  COMMUT_ITER = "!A.[f:A->A][g:A->A]
   (commute A f g)->(commute A f [x:A](f (g x)))";;

let COMMUT_ITER_PROOF = "!A.[f:A->A][g:A->A][com:(commute A f g)]
   [x:A]{P|A->*}[p:(P (f (g (f x))))]
   (com x [u:A](P (f u)) p)";;
LET 'powerS' COMMUT_ITER_PROOF COMMUT_ITER;;
```

If g is an iterate of f, then f and g commute.

```
let LEMMA2 = "!A.[f:A->A][g:A->A](power A f g)->(commute A f g)";;

let LEMMA2_PROOF = "!A.[f:A->A][g:A->A][pow:(power A f g)]
   (pow [h:A->A](commute A f h)
      (power0 A f)
      [h:A->A](powerS A f h))";;
LET 'l2' LEMMA2_PROOF LEMMA2;;

let ANDREWS = "!A.[f:A->A][g:A->A] (power A f g) ->
   <A>Sig((fixpt A g)) -> (unique A (fixpt A g)) -> <A>Sig((fixpt A f))";;
```

```
let ANDREWS_PROOF = "!A.[f:A->A][g:A->A][pow:(power A f g)]
   [fix:<A>Sig((fixpt A g))]
   [uni:(unique A (fixpt A g))]
   (l1 A f g (l2 A f g pow) fix uni)";;

LET 'Andrews' ANDREWS_PROOF ANDREWS;;
```
QED

5. Relational constructions.

A few elementary properties of binary relations, following Frege[20].

5.1. Context factorization.

Now me must explain further commands of our theory system. First we remark that the proof of Andrews' lemma above was cluttered by irrelevant polymorphism: the common domain A of all the constructions in the proof should have been factored once and for all, instead of being repeatedly abstracted and applied. This is the purpose of the following commands.

```
DECL: string -> context -> void
AXIOM: string -> object -> void
DISCHARGE: string -> void
```

The commands DECL and AXIOM permit to factorize a context common to all constructions in a big proof. Rather than prefixing all those constructions by this context, which further clutters the proof since all constants must be applied to the corresponding parameters, we give the user a way to place itself inside a current context. This is the usual way mathematics is developed in first-order logic: we assume the existence of a domain of discourse, we assume a language of predicates and function letters, and we assume axioms, such as Zermelo-Fraenkel's ones in set theory. The command DECL assumes a propositional construction, of type the given context, the command AXIOM assumes the existence of an object verifying the given proposition, the command DISCHARGE discharges the corresponding notion.

```
DECL 'A' "*";;
DECL 'R' "A->A->*";;
```
We now are in the theory of a binary relation R over a domain A.

5.2. General properties of a binary relation R over set A.

Reflexivity.
```
let REFLEXIVITY = "{R'|A->A->*}[x:A](R' x x);;
PROP 'Refl' REFLEXIVITY;;
```

Transitivity.
```
let TRANSITIVITY = "{R'|A->A->*}[x:A][y:A][z:A](R' x y)->(R' y z)->(R' x z)";;
PROP 'Trans' TRANSITIVITY;;
```

R-hereditary.
```
let HER = "{P|A->*}[x:A][y:A] (P x)->(R x y)->(P y)";;
PROP 'Her' HER;;
```

Inductive definition of R^+.

```
let TRANS_CLOSURE =
    "[x:A][y:A]{P|A->*}(Her P)->([u:A](R x u)->(P u))->(P y)";;
PROP 'Rplus' TRANS_CLOSURE;;

R+ is R-hereditary.
let HER_RPLUS = "[x:A] (Her (Rplus x))";;

let HER_RPLUS_PROOF = "[x:A][y:A][z:A][h1:(Rplus x y)][h2:(R y z)]
      {P|A->*}[h3:(Her P)][h4:[u:A](R x u)->(P u)]
      (h3 y z (h1 P h3 h4) h2)";;
LET 'Her_Rplus' HER_RPLUS_PROOF HER_RPLUS;;

R+ is increasing.
let RPLUS_INCREASING = "[x:A][y:A] (R x y)->(Rplus x y)";;

let RPLUS_INCREASING_PROOF = "[x:A][y:A][h1:(R x y)]{P|A->*}
    (Her P) -> [h2:[z:A](R x z)->(P z)](h2 y h1)";;
LET 'Rplus_increasing' RPLUS_INCREASING_PROOF RPLUS_INCREASING;;

R+ is transitive.
let TRANS_RPLUS = "(Trans Rplus)";;

let TRANS_RPLUS_PROOF = "[x:A][y:A][z:A][h1:(Rplus x y)][h2:(Rplus y z)]
    let H = (Her_Rplus x) in
        (h2 (Rplus x) H ([u:A][h3:(R y u)](H y u h1 h3)))";;
Note the curious double use of (Her_Rplus x)
LET 'Trans_Rplus' TRANS_RPLUS_PROOF TRANS_RPLUS;;

R*
let TRANS_REFL_CLOSURE  = "[x:A][y:A] (<A>y=x) + (Rplus x y)";;
PROP 'Rstar' TRANS_REFL_CLOSURE;;
Note that Frege chose y=x for simplicity in proof below.

let RSTAR_TO_RPLUS = "[x:A][y:A][z:A](Rstar x y)->(R y z)->(Rplus x z)";;

let RSTAR_TO_RPLUS_PROOF = "[x:A][y:A][z:A][h1:(Rstar x y)][h2:(R y z)]
    (h1 (Rplus x z)
       [h3:<A>y=x](Rplus_increasing x z (h3 [u:A](R u z) h2))
       [h3:(Rplus x y)](Her_Rplus x y z h3 h2))";;
LET 'Rstar_to_Rplus' RSTAR_TO_RPLUS_PROOF RSTAR_TO_RPLUS;;

R* is reflexive.
let REFL_RSTAR = "(Refl Rstar)";;

let REFL_RSTAR_PROOF  = "[x:A]!P.[h1:(<A>x=x)->P]
    ((Rplus x x)->P) -> (h1 (refl_equal A x))";;
LET 'Refl_Rstar' REFL_RSTAR_PROOF REFL_RSTAR;;

let RPLUS_TO_RSTAR = "[x:A][y:A] (Rplus x y) -> (Rstar x y)";;

let RPLUS_TO_RSTAR_PROOF = "[x:A][y:A][h1:(Rplus x y)]
    !P.((<A>y=x)->P) -> [h3:(Rplus x y)->P](h3 h1)";;
LET 'Rplus_to_Rstar' RPLUS_TO_RSTAR_PROOF RPLUS_TO_RSTAR;;
```

5.3. Tarski's theorem

As an exercise in relational constructions, let us now show Tarski's theorem : If a function is monotonous over a complete upper-semi-lattice, it admits a fixpoint[57].

x is an R-upper bound of subset M.
```
let BOUND = "[x:A]{M|A->*}[y:A](M y)->(R y x)";;
PROP 'bound' BOUND;;
```

R-increasing function.
```
let MONOTONOUS = "[f:A->A][x:A][y:A](R x y)->(R (f x) (f y))";;
PROP 'mon' MONOTONOUS;;
```

f (u)≡u with ≡ the symmetric quotient of R.
```
let FIX = "[f:A->A][u:A](R u (f u))&(R (f u) u)";;
PROP 'fix' FIX;;

let COMPL_SEMI_LATTICE =
    "{M|A->*}!B.([u:A](bound u M)->([x:A](bound x M)->(R u x))->B)->B";;
PROP 'csl' COMPL_SEMI_LATTICE;;

let TARSKI = "(Trans R) -> csl -> [f:A->A](mon f) -> <A>Sig((fix f))";;

let TARSKI_PROOF = "[trans:(Trans R)][ub:csl][f:A->A][mo:(mon f)]
    !B.[h:[u:A]((R u (f u))&(R (f u) u))->B]
    (ub ([u:A](R u (f u))) B
       ([x:A][h1:[u:A](R u (f u))->(R u x)]
           [h2:[u:A]([v:A](R v (f v))->(R v u))->(R x u)]
  let p = ([y:A][h':(R y (f y))](trans y (f y) (f x) h' (mo y x (h1 y h'))))
    in let xRfx = (h2 (f x) p) in
    (h x <(R x (f x)),(R (f x) x)>(xRfx,(h1 (f x) (mo x (f x) xRfx))))))";;
LET 'Tarski' TARSKI_PROOF TARSKI;;
```
QED

6. Frege's lemma

This section is devoted to prove that injectivity implies linearity, following the proof of Frege[20].

R is injective.
```
let INJECTIVE = "[x:A][y:A][z:A] (R x y) -> (R x z) -> <A>y=z";;
PROP 'Injective' INJECTIVE;;
```

R is R-hereditary.*
```
let HER_RSTAR = "[x:A] (Her (Rstar x))";;

let HER_RSTAR_PROOF = "[x:A][y:A][z:A][h1:(Rstar x y)][h2:(R y z)]
    (Rplus_to_Rstar x z (Rstar_to_Rplus x y z h1 h2))";;
LET 'Her_Rstar' HER_RSTAR_PROOF HER_RSTAR;;

let LINEARITY_LEMMA =
    "Injective->[x:A][y:A](R y x)->[z:A](Rplus y z)->(Rstar x z)";;
```

```
let LINEARITY_LEMMA_PROOF =
   "[inj:Injective][x:A][y:A][yRx:(R y x)][z:A][h:(Rplus y z)]
   (h (Rstar x) (Her_Rstar x)
     ([u:A][yRu:(R y u)]
     (inj y x u yRx yRu (Rstar x) (Refl_Rstar x))))";;
LET 'linearity_lemma' LINEARITY_LEMMA_PROOF LINEARITY_LEMMA;;

let CASE_RSTAR = "[x:A][y:A](Rstar y x)->((Rplus y x)+(Rstar x y))";;
let CASE_RSTAR_PROOF  = "[x:A][y:A][h1:(Rstar y x)]
     !P.[h2:(Rplus y x)->P][h3:(Rstar x y)->P]
     (h1 P ([h4:<A>x=y](h3 (h4 (Rstar x) (Refl_Rstar x))))
          ([h5:(Rplus y x)](h2 h5)))";;
LET 'Case_Rstar' CASE_RSTAR_PROOF CASE_RSTAR;;

R-connected
let CONNECTED = "[x:A][y:A] (Rplus y x)+(Rstar x y)";;
PROP 'Connected' CONNECTED;;

We show that (R injective & xR+y & xR+z) -> (Connected z y)
let LINEARITY1 =
     "Injective->[x:A][y:A](R y x)->[z:A](Rplus y z)->(Connected z x)";;

let LINEARITY1_PROOF =
     "[h1:Injective][x:A][y:A][h2:(R y x)][z:A][h3:(Rplus y z)]
     (Case_Rstar z x (linearity_lemma h1 x y h2 z h3))";;
LET 'linearity1' LINEARITY1_PROOF LINEARITY1;;

let HER_CONNECTED =
     "Injective->[x:A][y:A][z:A](Connected x y)->(R y z)->(Connected x z)";;

let HER_CONNECTED_PROOF =
     "[h1:Injective][x:A][y:A][z:A][h2:(Connected x y)][h3:(R y z)]
     !P.[h4:(Rplus z x)->P][h5:(Rstar x z)->P]
     (h2 P ([h6:(Rplus y x)](linearity1 h1 z y h3 x h6 P h4 h5))
     ([h6:(Rstar x y)]
          (h5 (Rplus_to_Rstar x z (Rstar_to_Rplus x y z h6 h3)))))";;
LET 'her_linear' HER_CONNECTED_PROOF HER_CONNECTED;;

let LINEARITY =
"Injective -> [x:A][y:A][z:A](Rplus x z) -> (Rplus x y) -> (Connected z y)";;

let LINEARITY_PROOF =
     "[h1:Injective][x:A][y:A][z:A][h2:(Rplus x z)][h3:(Rplus x y)]
     (h3 (Connected z) (her_linear h1 z) ([u:A][h4:(R x u)]
          (linearity1 h1 u x h4 z h2)))";;
LET 'Linearity' LINEARITY_PROOF LINEARITY;;
QED
```

7. Newman's Lemma

This section uses the relational constructions from section 5. We show here Newman's lemma: A Noetherian relation is confluent iff it is locally confluent[43]. The method of proof follows Huet[29].

We first need to show that $x\ R^+\ y$ implies $\exists z\ x\ R\ z\ \&\ z\ R^*\ y$. We thus give a dual definition Rplus' of the transitive closure of R.

In order to avoid unduly currification, we give a preliminary construction for $\exists x\ P(x)\ \&\ Q(x)$.

```
let SIGMA2 = "!C.{P|C->*}{Q|C->*}!B.([x:C](P x)->(Q x)->B)->B";;
PROP 'Sig2' SIGMA2;;

let RPLUS' = "[x:A][y:A](Sig2 A [z:A](R x z) [z:A](Rstar z y))";;
PROP 'Rplus" RPLUS';;
```

Now we prove intermediate lemmas needed for RPLUS_TO_RPLUS'.

```
let R_TO_RPLUS' = "[x:A][y:A] (R x y)->(Rplus' x y)";;

let R_TO_RPLUS'_PROOF =
     "[x:A][y:A][h1:(R x y)]!B.[h2:[z:A](R x z)->(Rstar z y)->B]
     (h2 y h1 (Refl_Rstar y))";;
LET 'R_to_Rplus" R_TO_RPLUS'_PROOF R_TO_RPLUS';;

let HER_RPLUS' = "[x:A] (Her (Rplus' x))";;

let HER_RPLUS'_PROOF =
     "[x:A][y:A][z:A][h1:(Rplus' x y)][h2:(R y z)]
     !B.[h3:[u:A](R x u)->(Rstar u z)->B]
     (h1 B ([v:A][h4:(R x v)][h5:(Rstar v y)]
        (h3 v h4 (Rplus_to_Rstar v z (Rstar_to_Rplus v y z h5 h2)))))";;
LET 'Her_Rplus" HER_RPLUS'_PROOF HER_RPLUS';;

let RPLUS_TO_RPLUS' = "[x:A][y:A] (Rplus x y)->(Rplus' x y)";;

let RPLUS_TO_RPLUS'_PROOF =
     "[x:A][y:A][h1:(Rplus x y)]
     (h1 (Rplus' x) (Her_Rplus' x) (R_to_Rplus' x))";;
LET 'Rplus_to_Rplus" RPLUS_TO_RPLUS'_PROOF RPLUS_TO_RPLUS';;

let RSTAR_CASES = "[x:A][y:A] (Rstar x y)->((<A>x=y)+(Rplus x y))";;

let RSTAR_CASES_PROOF =
     "[x:A][y:A][h:(Rstar x y)]
     !B.[h1:(<A>x=y)->B][h2:(Rplus x y)->B]
     (h B ([eq:(<A>y=x)](h1 (sym_equal A y x eq))) h2)";;
```

Remark that we pay here the orientation of = in the definition of Rstar.

```
LET 'Rstar_cases' RSTAR_CASES_PROOF RSTAR_CASES;;
```

R^* is transitive.

```
let TRANS_RSTAR = "(Trans Rstar)";;

let TRANS_RSTAR_PROOF =
     "[x:A][y:A][z:A][h1:(Rstar x y)][h2:(Rstar y z)]
     (h1 (Rstar x z) ([h3:(<A>y=x)](h3 [u:A](Rstar u z) h2))
        ([h3:(Rplus x y)](Rstar_cases y z h2 (Rstar x z)
           [h4:(<A>y=z)](Rplus_to_Rstar x z (h4 ([u:A](Rplus x u)) h3))
         [h4:(Rplus y z)](Rplus_to_Rstar x z (Trans_Rplus x y z h3 h4)))))" ;;
```

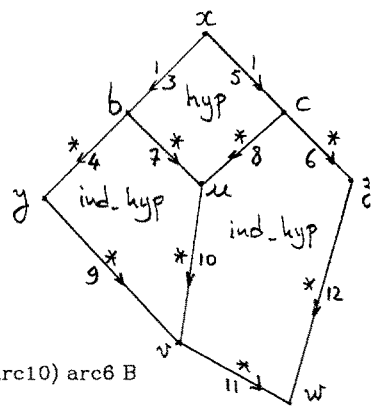
```
LET 'Trans_Rstar' TRANS_RSTAR_PROOF TRANS_RSTAR;;

let COHERENT = "[x:A][y:A] (Sig2 A (Rstar x) (Rstar y))";;
PROP 'Coherent' COHERENT;;

let CONFLUENT = "[x:A][y:A][z:A] (Rstar x y)->(Rstar x z)->(Coherent y z)";;
PROP 'Confluent' CONFLUENT;;

let LOCALLY_CONFLUENT = "[x:A][y:A][z:A] (R x y)->(R x z)->(Coherent y z)";;
PROP 'Locally_Confluent' LOCALLY_CONFLUENT;;
```

The diagram.

```
let DIAGRAM =
    "[x:A]([u:A](R x u) -> (Confluent u)) -> (Locally_Confluent x) ->
    [y:A][z:A](Rplus' x y) -> (Rplus' x z) -> (Coherent y z)";;
```

Here is the important part of the proof: filling out the diagram. Remark how the existential quantifications allow us to progressively "draw" the diagram.

```
let DIAGRAM_PROOF =
    "[x:A]
    [induction_hyp:[u:A](R x u)->(Confluent u)]
    [hyp:(Locally_Confluent x)]
    [y:A][z:A]
    [arc1:(Rplus' x y)]
    [arc2:(Rplus' x z)]
    !B.[coher:[u:A](Rstar y u)->(Rstar z u)->B]
    (arc1 B
    ([b:A][arc3:(R x b)][arc4:(Rstar b y)]
    (arc2 B
    ([c:A][arc5:(R x c)][arc6:(Rstar c z)]
    (hyp b c arc3 arc5 B
      ([u:A][arc7:(Rstar b u)][arc8:(Rstar c u)]
       (induction_hyp b arc3 y u arc4 arc7 B
          ([v:A][arc9:(Rstar y v)][arc10:(Rstar u v)]
           (induction_hyp c arc5 v z (Trans_Rstar c u v arc8 arc10) arc6 B
              ([w:A][arc11:(Rstar v w)][arc12:(Rstar z w)]
       (coher w (Trans_Rstar y v w arc9 arc11) arc12))))))))))";;
LET 'diagram' DIAGRAM_PROOF DIAGRAM;;
```




```
let COROLLARY =
    "[x:A]([a:A](R x a)->(Confluent a))->(Locally_Confluent x)->
    [y:A][z:A](Rplus x y)->(Rplus x z)->(Coherent y z)";;

let COROLLARY_PROOF =
    "[x:A][induction_hyp:[a:A](R x a)->(Confluent a)]
    [hyp:(Locally_Confluent x)][y:A][z:A][arc:(Rplus x y)][arc':(Rplus x z)]
    (diagram x induction_hyp hyp y z
        (Rplus_to_Rplus' x y arc)
        (Rplus_to_Rplus' x z arc'))";;
LET 'corollary' COROLLARY_PROOF COROLLARY;;
```

Newman's lemma assuming proper induction hypotheses.

```
let NEWMAN_IND =
    "[x:A]([a:A](R x a) -> (Confluent a)) -> (Locally_Confluent x) ->
    [y:A][z:A](Rstar x y) -> (Rstar x z) -> (Coherent y z)";;
```

```
let NEWMAN_IND_PROOF =
     "[x:A]
     [induction_hyp:[a:A](R x a)->(Confluent a)]
     [hyp:(Locally_Confluent x)]
     [y:A][z:A]
     [arc:(Rstar x y)]
     [arc':(Rstar x z)]
     (Rstar_cases x y arc (Coherent y z)
      ([case1:(<A>x=y)]!B.[coher:[v:A](Rstar y v)->(Rstar z v)->B]
                (coher z
                   (case1 ([u:A](Rstar u z)) arc')
                   (Refl_Rstar z)))
   ([case2:(Rplus x y)](Rstar_cases x z arc' (Coherent y z)
       ([case21:(<A>x=z)]!B.[coher:[v:A](Rstar y v)->(Rstar z v)->B]
                (coher y
                   (Refl_Rstar y)
                   (case21 ([u:A](Rstar u y)) arc)))
        ([case22:(Rplus x z)]
           (corollary x induction_hyp hyp y z case2 case22)))))";;
LET 'Newman_ind' NEWMAN_IND_PROOF NEWMAN_IND;;
```

The proofs in this section may appear formidable at first sight. Actually, they follow quite naturally the mathematical reasoning, and one may argue that the language of constructions is the ideal medium for natural deduction in higher order predicate calculus. What is most needed at this point is a good interactive system allowing to "debug" its proof progressively, with machine assistance for automating the trivial steps.

We now introduce Noetherian induction.

```
let NOETHERIAN = "{P|A->*} ([x:A]([y:A](R x y)->(P y))->(P x))->[x:A](P x)";;
PROP 'Noetherian' NOETHERIAN;;
```

Noetherian induction is both a proposition stating that relation R is Noetherian (i.e. there are no infinite R-chains), and the type of functional objects which may be applied to a property P in order to show $\forall x \in A \cdot P(x)$ by induction. Once again we stress operational meanings: the proposition "R is Noetherian" is the type of induction principles.

Newman's lemma.

```
let NEWMAN = "Noetherian -> Locally_Confluent -> Confluent";;

let NEWMAN_PROOF =
     "[h1:Noetherian][h2:Locally_Confluent]
     (h1 ([u:A](Confluent u))
     ([x:A][ind:[y:A](R x y)->(Confluent y)] (Newman_ind x ind (h2 x))))";;
LET 'Newman' NEWMAN_PROOF NEWMAN;;
```

QED.

We believe this is the first machine-checked proof of Newman's lemma.

8. Categorical constructions

It is instructive to axiomatize the notion of category in the language of constructions. The most natural definition is to parameterize the notion of category on a class Obj, a binary relation Hom on Obj, and operations id and o verifying the usual laws of generalized monoid:

```
CATEGORY = "!Obj.(Hom|[A:Obj][B:Obj]*)[id:[A:Obj](Hom A A)]
  [o:[A:Obj][B:Obj][C:Obj](Hom A B)->(Hom B C)->(Hom A C)]
   ([A:Obj][B:Obj][f:(Hom A B)]<(Hom A B)> (o A A B (id A) f) = f)
  & ([A:Obj][B:Obj][f:(Hom A B)]<(Hom A B)> (o A A B f (id B)) = f)
  & ([A:Obj][B:Obj][C:Obj][D:Obj][f:(Hom A B)][g:(Hom B C)][h:(Hom C D)]
      <(Hom A D)> (o A C D (o A B C f g) h) = (o A B D f (o B C D g h)))";;
```

This definition is not quite general enough, since it assumes intensional equality for morphisms. A more general definition would replace the equality = by a relation E postulated to be an equivalence relation compatible with composition. We shall not develop further those categorical constructions here, but we just remark that there is no intrinsic difference in our view between Hom and a relation on Obj. This is because we have an intuitionistic view of models: the interpretation of a relation is not just "true" or "false", but the set of meanings which prove such a relation. Note that from this "realizability" point of view, stating the existence of identity and composition is just postulating the relation Hom to be reflexive and transitive.

9. Algebraic constructions

The following section needs only the logical constructions from section 2. We show here how to develop the standard notions from abstract algebra in a natural way. First we show how to embed the signatures of homogeneous algebras into simple propositions.

9.1. Finite domains

The empty set, already seen as Falsity.

```
let EMPTY = "!A.A";;
PROP '{}' EMPTY;;
```

The unit set.

```
let UNIT = "!A.A->A";;
PROP 'unit' UNIT;;
```

Already seen as 'self'; contains as unique element Id ="!A.[u:A]u"

Booleans.

```
let BOOL = "!A.A->A->A";;
PROP 'bool' BOOL;;
```

Church's Booleans : $\lambda x,y.y$ *and* $\lambda x,y.x$ [13].

```
let TRUE = "!A.[u:A][v:A]u" and FALSE = "!A.[u:A][v:A]v";;

LET 'true' TRUE "bool";;
LET 'false' FALSE "bool";;
```

Complement.
```
let COMPL = "[b:bool](b bool false true)";;
LET 'not' COMPL "bool->bool";;
```

Conditional.
```
let IF = "!A.[b:bool](b A)";;
LET 'if' IF "!A.bool->A->A->A";;
```

Note that, for p:A&A, <A,A>fst(p)=(p (true A)) and similarly with snd/false

Standard Boolean functions.
```
let AND = "[b:bool][b':bool](b bool b' false)";;
LET 'And' AND "bool->bool->bool";;

let OR = "[b:bool][b':bool](b bool true b')";;
LET 'Or' OR "bool->bool->bool";;
```

It is easy to generalize the constructions above to define the canonical polymorphic structure with n elements.

9.2. Natural numbers

```
let NAT = "!A.(A->A)->A->A";;
PROP 'nat' NAT;;

let ZERO = "!A.(A->A)->(Id A)";;
LET '0' ZERO "nat";;

let SUCC = "[n:nat]!A.[s:A->A][z:A](s (n A s z))";;
LET 'S' SUCC "nat->nat";;
LET_SYNTAX 'S' ['S(';')'];;
```

"Concrete" natural numbers completely expanded out.
```
let ONE = apply(SUCC,ZERO);;
letrec NATURAL n = if n<0 then fail
                   if n=0 then ZERO else apply(SUCC,NATURAL (n-1));;
```

"Abstract" natural numbers in terms of the constants 0 and S.
```
let Zero = "0" and Succ = "S";;
let ONE = "(S 0)";;
LET '1' ONE "nat";;
```

More generally:
```
letrec Natural n = if n=0 then Zero else apply(Succ,Natural (n-1));;
let Decl_Nat n = LET (string_of_int n) (Natural n) "nat";;
```
This permits to declare any nat with its usual representation.

Note that the untyped λ-expressions corresponding to proofs of type nat are exactly Church's integers[13]. However, we need the power of second-order types to give them a uniform type, that of polymorphic iterators.

Since all our constructions are functional objects, there is no distinction between data structures and control structures. A data structure is just a control structure waiting for additional arguments in order to project itself.

Thus booleans implement conditionals, integers are for loops, etc...

If one sees nat as denoting the initial algebra in the class of all algebras with signature <A,s:A->A,z:A>, the role of the initiality morphism is here taken by application, since the homomorphic image of n:nat in the structure <A,s:A->A,z:A> is simply (n A s z). This should clarify the definition of SUCC above. We shall develop these algebraic ideas more systematically in the next section.

9.3. The general case of homogeneous varieties.

In this section we show how to synthesize in ML the constructions of abstract algebra.

Abstracting k times on the carrier, which we assume is bound at level m

```
let arity m n = arityrec n
   whererec arityrec k = let carrier = rel(n+m-k) in
      if k=0 then carrier
           else function(carrier,arityrec (k-1));;
```

The canonical structure with n elements.

```
let FINITE n = if n<0 then fail else polymorph(star,arity 1 n);;
```

The case statement.

```
let CASE(n,m) = if n<0 or m<1 or m>n then fail else
   polymorph(star,caserec n) whererec caserec k =
      if k=0 then rel(m) else function(rel(n+1-k),caserec(k-1));;
```

We have the following identities:

```
 EMPTY=FINITE(0)
 UNIT=FINITE(1)   ID=CASE(1,1)
 BOOL=FINITE(2)   FALSE=CASE(2,1)  TRUE=CASE(2,2)
```

More generally, we define a signature as a list of non-negative integers which determines a functionality as follows.

```
letrec functionality m sig obj =
   if sig=[] then (obj m)
   else function(arity m (hd sig),functionality (m+1) (tl sig) obj);;

let free_algebra sig = polymorph(star,functionality 1 sig rel);;
```

So we could have defined NAT as free_algebra[1;0]. In a similar fashion, we could synthesize the constructors of such free algebras. Let us now turn to the non-homogeneous case.

9.4. Heterogeneous algebras

Lists

```
let LIST = "!A,B.(A->B->B)->B->B";;
PROP 'list' LIST;;
```

```
let NIL = "!A,B.(A->B->B)-><B>Id";;
LET 'nil' NIL "list";;
```
*Compare with ML, where nil:*list*

```
let CONS = "!A.[x:A][y:(list A)]!B.[u:A->B->B][v:B](u x (y B u v))";;
LET 'cons' CONS "!A.A->(list A)->(list A)";;
```

Assume proposition A given, and let us abbreviate, for $a_1, \cdots, a_n$ of type A, the construction "!B.[u:A->B->B][v:B](u a_1 (u a_2 ... (u a_n v)...))" as $[a_1;...;a_n]$. An element $[a_1;...;a_n]$ of (list A) is thus a control structure which may operate on any algebra <B,U:A->B->B,V:B> in order to compute the element of B (U a_1 (U a_2 ... (U a_n V)...)). This is exactly the role of the ML *itlist* primitive, or the APL *reduce* operator. Similarly, we may define trees as tree iterators, as follows.

Trees = !A.(A->A->A)->A->A

```
let BINTREE = "!A.(list A A)";;
PROP 'bintree' BINTREE;;

let NIL_TREE = "!A.(nil A A)";;
LET 'nil_tree' NIL_TREE BINTREE;;

let MKTREE =
"[left:bintree][right:bintree]!A.[c:A->A->A][n:A](c (left A c n) (right A c n))";;
LET 'mktree' MKTREE "bintree->bintree->bintree";;
```

We may treat in the same way more complex tree structures, strings, and more generally all the standard data structures which correspond to free algebras. Note that all the propositions needed here correspond to Girard's second order types restricted to degree 2, similarly to the treatment in Berarducci-Böhm[4].

9.5. Non-free structures

All the standard free algebras on one carrier are of the form "!A.P1->P2-> ... ->Pn->A" with Pi of the form (A->A->...->A). That is, we are restricting ourselves to second-order types of degree 2. Let us now consider algebras defined by types of degree 3.

Binding.

```
let BIND = "!A.((A->A)->A)->A";;
PROP 'bind' BIND;;
```

The algebra bind has only one constructor, which may be thought of as a quantifier, which introduces locally a new generator. The proofs of bind are all expressions of the form

$$\forall x_1.\forall x_2.\cdots\forall x_n.x_i \quad with \ 1 \le i \le n$$

i.e. the algebra bind is isomorphic to the relation $\geq$ over **N**. For instance, the formula $\forall x_1.\forall x_2.x_1$, or isomorphically $2 \geq 1$, is represented by the following proof of type bind.

```
LET '2>=1' "!A.[quant:(A->A)->A](quant [x1:A](quant [x2:A]x1))" BIND;;
```

Lambda expressions.
```
let LAMBDA = "!A.((A->A)->A)->(A->A->A)->A";;
PROP 'lambda' LAMBDA;;
```

This is the algebra of ordinary lambda expressions. The first operator is λ, the second is apply. For instance, $\lambda x \cdot \lambda y \cdot (x\ y)$ is represented as the following proof of type lambda:

```
LET '\\(2 1)'
    "!A.[lam:(A->A)->A][ap:A->A->A](lam [x:A](lam [y:A](ap x y)))" LAMBDA;;
```

Typing.
```
let TYPE = "!A.(A->(A->A)->A)->A->A";;
PROP 'type' TYPE;;
```

This is the algebra of one typing operator similar to our own abstraction operator, and one base type, similar to our own star. For instance, the expression [x:*][y:x]y is represented as the following construction of type type:

```
LET '[*][1]1'
    "!A.[typ:A->(A->A)->A][star:A](typ star [x:A](typ x [y:A]y))" TYPE;;
```

Typed λ-calculus.
```
let DELTA = "!A.(A->(A->A)->A)->(A->A->A)->A->A";;
PROP 'delta' DELTA;;
```

This algebra, obtained by mixing the operators of lambda with those of type, is in a sense the simplest typed lambda calculus structure. This is exactly Nederpelt's Λ, and thus the algebra of our own language. For instance, the construction "[B:*][x:B][f:[u:B]B](f x)" is internalized as the following proof of type delta.

```
LET '[*][1][[2]3](1 2)'
  "!A.[abs:A->(A->A)->A][ap:A->A->A][star:A]
     (abs star [B:A](abs B [x:A](abs (abs B [u:A]B) [f:A](ap f x))))"
  DELTA;;
```

We could for instance present the type-checker of constructions as a construction in "delta->delta+unit", but this is out of the scope of the present paper.

Note how the constructions above are a natural generalization of the standard definitions from abstract algebra, where the notion of signature has been enriched to be not just a list of integer arities, but more generally a list of signatures of lower level algebras. We have the finite structures at level 1, the standard term algebras at level 2, the various lambda calculi at level 3. This quick investigation of the simplest of Girard's second order types[22] should give an idea of the richness of expression of the full language. The next section gives an example where we use an inner type quantification.

9.6. Ordinals.

Ordinals are defined in a similar way to natural numbers, adding a limit operation. A sequence over type A is of type (nat->A), and thus a limit operator over A is of type (nat->A)->A.

```
let ORD = "!A.((nat->A)->A)->(A->A)->A->A";;
PROP 'ord' ORD;;

let ORDZERO = "!A.[lim:(nat->A)->A][s:A->A][z:A]z";;
LET 'ordzero' ORDZERO "ord";;

let ORDSUCC = "[a:ord]!A.[lim:(nat->A)->A][s:A->A][z:A](s (a A lim s z))";;
LET 'ordsucc' ORDSUCC "ord->ord";;

Limit of a sequence of ordinals.
let ORDLIM = "[seq:nat->ord]!A.[lim:(nat->A)->A][s:A->A][z:A](lim [n:nat](seq n
A lim s z))";;
LET 'ordlim' ORDLIM "(nat->ord)->ord";;

Finite ordinals are obtained by coercing nats into ords.
let ORDNAT = "[n:nat](n ord ordsucc ordzero)";;
LET 'ordnat' ORDNAT "nat->ord";;

ω
let OMEGA = "(ordlim ordnat)";;
LET 'omega' OMEGA "ord";;
```

10. Programming constructions.

We finally show how we can use the algebraic constructions as basic data structures and control structures of a very-high level programming language.

10.1. Arithmetic on natural numbers in unary notation.

```
Addition.
let ADD = "[n:nat](n nat S)";;
LET 'add' ADD "nat->nat->nat";;
LET_SYNTAX 'add' ['(';'+';')'];;
```
Other possible def: "*[m:nat][n:nat](n nat S m)*". *This last one generalizes to ordinal addition.*

```
Multiplication.
let MULT = "[n:nat][m:nat](n nat (add m) 0)";;
LET 'mult' MULT "nat->nat->nat";;
```
A less abstract one is composition (the B Curry combinator)
"*[n:nat][m:nat]!A.[f:A->A](n A (m A f))*". *Or its dual:*
"*[n:nat][m:nat]!A.[f:A->A](m A (n A f))*".

```
Exponentiation.
let EXP = "[n:nat][m:nat](m nat (mult n) 1)";;
LET 'exp' EXP "nat->nat->nat";;
```

Super-Exponential: n^(n^ ... (n^n)...).
```
let SUPEXP = "[n:nat][k:nat](k nat (exp n) n)";;
LET 'supexp' SUPEXP "nat->nat->nat";;
```

10.2. Primitive recursion

We want to implement primitive recursive equations of the form:

$$g(0)=a \quad , \quad g(n+1)=f(n,g(n))$$

For this, we build in for loops, that is flowcharts such as :

$$Iterate(f,a,n)=z:=a\,;\ \text{for } i:=0 \text{ to } n-1 \text{ do } z:=f(i,z)$$

We get such a construct by iterating on a pair: $g(n)=snd(n\ h\ <0,a>)$ with

$$h<i,g(i)> = <i+1,g(i+1)> = <i+1,f(i,g(i))>$$

which we generalize as $h<i,x> = <i+1,f(i,x)>$. More generally, flowcharts iterate commands, in store->store, for an appropriate value of store (i.e., the product of the types of the relevant identifiers). These remarks lead directly to the iter construction:

```
let ITER = "!A. let store=nat&A in
            let mkstore=[i:nat][a:A]<nat,A>(i,a) in
            let index=[st:store]<nat,A>fst(st) in
            let result=[st:store]<nat,A>snd(st) in
     [f:store->A][a:A]
     let com=[st:store](mkstore (S (index st)) (f st)) in
     let inist=(mkstore 0 a) in
     [n:nat](result (n store com inist))";;
LET 'iter' ITER "!A.((nat&A)->A)->A->nat->A";;
```

Exemple: predecessor, defined as pred(0)=0, pred(n+1)=n
```
let PRED = "let index=[st:nat&nat]<nat,nat>fst(st) in (iter nat index 0)";;
LET 'pred' PRED "nat->nat";;
```

*Factorial: fact(0)=1, fact(n+1)=(n+1)*fact(n)*
```
let FACT =
 "let factcom=[st:nat&nat](mult (S <nat,nat>fst(st)) <nat,nat>snd(st)) in
     (iter nat factcom 1)";;
LET 'fact' FACT "nat->nat";;
```

It is now easy to imagine a compiler which macro generates the call to iter, with a command part containing identifiers compiled as projection functions in the store. More ambitiously, we may access variables through an environment, and have more complex commands using continuations. A program will be a construction in the algebra of the abstract syntax of some programming language, whose semantics is given by constructions in the traditional style of denotational semantics.

Equality to zero
```
let EQZERO = "[n:nat](n bool [b:bool]false true)";;
LET 'eq0' EQZERO "nat->bool";;
```

A more complicated recursive schema: Fibonacci.

$$fibo(0)=a, fibo(1)=b, fibo(n+2)=F(fibo(n), fibo(n+1))$$

```
let FIBO = "!A.let store=A&A in
      let mkstore=[x:A][y:A]<A,A>(x,y) in
      let res=[st:store]<A,A>snd(st) in
      [F:store->A]
      let com=[st:store](mkstore (res st) (F st)) in
      [a:A][b:A]
      let inist=(mkstore a b) in
      [n:nat](eq0 n A a (res (n store com inist)))";;
LET 'fibo' FIBO "!A.((A&A)->A)->A->A->nat->A";;
```

Exemple: the standard Fibonacci function.

```
let FIB =
      "let F=[st:nat&nat](add <nat,nat>fst(st) <nat,nat>snd(st)) in
      (fibo nat F 1 1)";;
LET 'fib' FIB "nat->nat";;
```

Beyond primitive recursion: Ackerman's function.

```
let ACK = "[n:nat](n (nat->nat) [f:nat->nat][n:nat](n nat f n) S)";;
LET 'ack' ACK "nat->nat->nat";;
```

This shows the power of iteration on functional types.

10.3. List processing

List concatenation

```
let APPEND = "!A.[l1:(list A)][l2:(list A)](l2 (list A) (cons A) l1)";;
LET 'append' APPEND "!A.(list A)->(list A)->(list A)";;
```

Insert at end of list

```
let POST = "!A.[a:A][l:(list A)](l (list A) (cons A) (cons A a (nil A)))";;
LET 'post' POST "!A.A->(list A)->(list A)";;

let REVERSE = "!A.[l:(list A)](l (list A) (post A) (nil A))";;
LET 'reverse' REVERSE "!A.(list A)->(list A)";;
```

Example : Hanoi's towers

```
let HANOI_TYPE = "!A.nat->A->A->A->(list (A & A))";;

let HANOI_PROGRAM = "!A.[n:nat]
   let pair=[x:A][y:A]<A,A>(x,y) in
   (n (A->A->A->(list (A & A)))
      [H:(A->A->A->(list (A & A)))][a:A][b:A][c:A]
      (append (A & A) (H a c b) (cons (A & A) (pair a c) (H b a c)))
      ([a:A][b:A][c:A](nil (A & A))))";;
LET 'prog' HANOI_PROGRAM HANOI_TYPE;;
```

10.4. Ordinal programming

Ordinal sum.

```
let ORDSUM = "[a:ord][b:ord](b ord ordlim ordsucc a)";;
LET 'ordsum' ORDSUM "ord->ord->ord";;
```

Note: we have more than ordinals, i.e. ordinals presented with a fundamental sequence. Thus (ordsum ordone omega) is different from omega.

Ordinal multiplication.
```
let ORDMULT = "[a:ord][b:ord](b ord ordlim (ordsum a) ordzero)";;
LET 'ordmult' ORDMULT "ord->ord->ord";;
```

ordone = (ordnat 1)
```
let ORDONE = "(ordsucc ordzero)";;
LET 'ordone' ORDONE "ord";;
```

Ordinal exponentiation.
```
let ORDEXP = "[a:ord][b:ord](b ord ordlim (ordmult a) ordone)";;
LET 'ordexp' ORDEXP "ord->ord->ord";;
```

ε_0
```
let EPSILON0 = "(omega ord ordlim (ordexp omega) ordzero)";;
LET 'epsilon0' EPSILON0 "ord";;
```

We now show how to use ordinals to describe functional hierarchies.

Schwichtenberg's hierarchy[19]
```
let FAST = "let natfun=nat->nat in
     let natfunzero=S in
     let natfunsucc=[f:natfun][n:nat](n nat f n) in
     let natfunlim=[seq:nat->natfun][n:nat](seq n n) in
     [a:ord](a natfun natfunlim natfunsucc natfunzero)";;
LET 'fast' FAST "ord->nat->nat";;
```
Note: (fast epsilon0) is not provably total in Peano's arithmetic.

```
let SLOW = "let natfun=nat->nat in
     let natfunzero=S in
     let natfunsucc=[f:natfun][n:nat](S (f n)) in
     let natfunlim=[seq:nat->natfun][n:nat](seq n n) in
     [a:ord](a natfun natfunlim natfunsucc natfunzero)";;
LET 'slow' SLOW "ord->nat->nat";;
```
Note that only natfunsucc changes from the fast to the slow hierarchy.

Conclusion

Leibniz was the first to seek a universal language for mathematical reasoning. The graphical proof notation of Frege[20] anticipated the works on natural deduction[21] and constructive mathematics[5,6]. The first real attempt at mechanizing mathematics was the Automath effort[11,30,54]. Martin-Löf made a fundamental contribution to the problem in presenting his 1971 theory of types[35]. This first attempt was shown inconsistent by J.Y. Girard. This prompted Martin-Löf to develop predicative theories[36,37,38,39]. Various implementations of intuitionistic type theory are now under development[55,56,44,14,15,3]. On the other hand, impredicative systems such as Girard's second-order types[22,23] discovered independently by Reynolds[51,52,53] have been recently the subject of close investigation[19,34,4]

A related field is of course that of automated theorem-proving. Powerful systems able to machine-check significant theorems are scarce. The Boyer-Moore theorem prover has been able to mechanize successfully a substantial

part of recursive arithmetic[7,8,9,10]. Several attempts have been made at mechanizing Church's type theory[2,1,49,48,28,27,26,40]. A lot of research has been put into mechanically verifying computer programs. The most impressive effort so far is the LCF project[25,45,47,46]. Another interesting theorem-proving system is Ketonen's EKL, who recently obtained a mechanical proof of Ramsey's theorem[32,33,31].

The current proposal is a tentative to blend together Martin-Löf's 1971 theory of types and Girard's second order types as a generalization of the Automath's formalisms. We have shown that the resulting language was well suited to expressing in a concise way notions taken from various mathematical fields.

It is also possible to regard our constructions as a very high level programming language, where propositions/specifications/types are used to decorate proofs/programs. To each proposition corresponds a canonical data-structure and its associated control-structure. To valid propositions correspond proofs which may be considered as programs obeying the corresponding specification. Executing the program corresponds to normalizing the proof to a cut-free direct proof. Such normalization is always possible for verified programs, and the computation does not involve the specification part, which may be considered a simple comment meaningless for computation.

This points out to the two essential problems that need to be solved in order to apply this work to the development of verifiably reliable software: firstly to establish programming methodologies implementing correctness verification as part of program development, with proper programming environments to support such methodologies in a user-friendly manner. Secondly, to develop computer architectures adequate for efficient lambda-calculus computation.

References

1. P. B. Andrews, *Resolution in Type Theory*, Journal of Symbolic Logic 36,3 pp. 414-432 (1971).
2. P. B. Andrews, Dale A. Miller, Eve Longini Cohen, and Frank Pfenning, *Automating higher-order logic* , Dept of Math. University Carnegie-Mellon (1983 January).
3. R. Backhouse, *Algorithm development in Martin-Löf's type theory*, University of Essex (July 1984).
4. A. Berarducci and C. Böhm, *Toward an Automatic Synthesis of Polymorphic Typed Lambda Terms*, ICALP (1984).
5. E. Bishop, *Foundations of Constructive Analysis*, McGraw-Hill, New-York (1967).
6. E. Bishop, *Mathematics as a numerical language*, Intuitionism and Proof Theory, Edited by J. Myhill, A.Kino and R.E.Vesley, North-Holland, Amsterdam, pp. 53-71 (1970).
7. R. Boyer and J Moore, *A Lemma Driven Automatic Theorem Prover for Recursive Function Theory* , 5th International Joint Conference on Artificial Intelligence, pp. 511-519 (1977).
8. R. Boyer and J. Moore, *A mechanical proof of the unsolvability of the halting problem* , Report ICSCA-CMP-28, Institute for Computing Science - University of Texas at Austin (July 1982).

9. R. Boyer and J. Moore, *Proof Checking the RSA Public Key Encryption Algorithm* , Report ICSCA-CMP-33, Institute for Computing Science - University of Texas at Austin (September 1982).
10. R. Boyer and J. Moore, *Proof checking theorem proving and program verification* , Report ICSA-CMP-35, Institute for Computing Science - University of Texas at Austin (January 1983).
11. N.G. de Bruijn, *Automath a language for mathematics*, Les Presses de l'Universite de Montréal (1973).
12. N.G. de Bruijn, *A survey of the project Automath*, Curry Volume, Academic Press (1980).
13. A. Church, *The Calculi of Lambda-Conversion*, Princeton U. Press, Princeton N.J. (1941).
14. R.L. Constable and J.L. Bates, *Proofs as Programs*, Dept. of Computer Science, Cornell University. (Feb. 1983).
15. R.L. Constable and J.L. Bates, *The Nearly Ultimate Pearl*, Dept. of Computer Science, Cornell University. (Dec. 1983).
16. Th. Coquand, *Une théorie des constructions*, Thèse de troisième cycle, Université Paris VII (Janvier 85).
17. Th. Coquand and G. Huet, *A Theory of Constructions*, Preliminary version, presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis (June 84).
18. D. Van Daalen, *The language of Automath*, Ph. D. Dissertation, Technological Univ. Eindhoven (1980).
19. S. Fortune, D. Leivant, and O'Michael Donnell, *The Expressiveness of Simple and Second-Order Type Structures*, Journal of the Association for Computing Machinery, Vol 30, No 1, pp 151-185 (January 1983).
20. G. Frege, *Begriffschrift*, in Heijenoort, From Frege to Gödel (1879).
21. G. Gentzen, *The Collected Paper of Gerhard Gentzen*, edited by E. Szabo, North-Holland, Amsterdam, 1969 (1969).
22. J.Y. Girard, *Une extension de l'interprétation de Gödel à l'analyse, et son application a l'élimination des coupures dans l'analyse et la théorie des types*, Proceedings of the Second Scandinavian Logic Symposium, Ed. J.E. Fenstad, North Holland, pp. 63-92 (1970).
23. J.Y. Girard, *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure*, Thèse d'Etat, Université Paris VII (1972).
24. M. Gordon, R. Milner, and C. Wadsworth, *A Metalanguage for Interactive Proof in LCF*, Internal Report CSR-16-77, department of Computer Science, University of Edinburgh (Sept. 1977).
25. M.J. Gordon, A. J. Milner, and C.P. Wadsworth, *Edinburgh LCF*, Springer-Verlag LNCS 78 (1979).
26. G. Huet, *Constrained Resolution: a Complete Method for Type Theory*, Ph.D. Thesis, Jennings Computing Center Report 1117, Case Western Reserve University (1972).
27. G. Huet, *A Mechanization of Type Theory*, Proceedings, 3rd IJCAI, Stanford (Aug. 1973).
28. G. Huet, *A Unification Algorithm for Typed Lambda Calculus*, Theoretical Computer Science, 1.1, pp. 27-57 (1975).

29. G. Huet, *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*, J. Assoc. Comp. Mach. 27,4 pp. 797-821 (Oct. 1980).

30. L.S. Jutting, *A translation of Landau's "Grundlagen" in AUTOMATH*, Eindhoven University of Technology, Dept of Mathematics (October 1976).

31. J. Ketonen, *A mechanical proof of Ramsey theorem*, Stanford Univ. CA (October 1983).

32. J. Ketonen, *EKL-A Mathematically Oriented Proof Checker*, 7th International Conference on Automated Deduction, Napa, California. LNCS 170, Springer-Verlag (May 1984).

33. J. Ketonen and J. S. Weening, *The language of an interactive proof checker*, Stanford University (1984).

34. D. Leivant, *Polymorphic type inference*, 10th POPL (1983).

35. P. Martin-Löf, *A theory of types*. October 1971.

36. P. Martin-Löf, *About models for intuitionistic type theories and the notion of definitional equality*, Paper read at the Orléans Logic Conference (1972).

37. P. Martin-Löf, *An intuitionistic Theory of Types, predicative part*, Logic Colloquium 73, pp. 73-118, North-Holland (1974).

38. P. Martin-Löf, *Constructive Mathematics and Computer Programming*, Logic, Methodology and Philosophy of Science VI, pp. 153-175, North-Holland (1980).

39. P. Martin-Löf, *Intuitionistic Type Theory*, Studies in Proof Theory, Bibliopolis (1984).

40. D.A. Miller, *Proofs in Higher-order Logic*, Ph. D. Dissertation, Carnegie-Mellon University (Aug. 1983).

41. R.P. Nederpelt, *Strong normalization in a typed lambda calculus with lambda structured types*, Ph. D. Thesis, Eindhoven University of Technology (1973).

42. R.P. Nederpelt, *An approach to theorem proving on the basis of a typed lambda-calculus*, Lecture Notes in Computer Science 87 : 5th Conference on Automated Deduction, Les Arcs, France, Springer-Verlag (1980).

43. M.H.A. Newman, *On Theories with a Combinatorial Definition of "Equivalence"*, Annals of Math. 43,2 pp.223-243 (1942).

44. B. Nordström, *Programming in Constructive Set Theory: Some Examples*, Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Portmouth, New Hampshire, p . 141-154 (Oct. 1981).

45. L. Paulson, *Recent Developments in LCF : Examples of structural induction*, Technical Report No 34, University of Cambridge, England (Janvier 1983).

46. L. Paulson, *Tactics and Tacticals in Cambridge LCF*, Technical Report No 39, Computer Laboratory, University of Cambridge (July 1983).

47. L. Paulson, *Verifying the unification algorithm in LCF*, Technical report No 50, Computer Laboratory, University of Cambridge (March 1984).

48. T Pietrzykowski and D.C Jensen, *A complete mechanization of* ω-order type theory, Proceedings of The ACM Annual Conference (1972).

49. T. Pietrzykowski, *A Complete Mechanization of Second-Order Type Theory*, JACM 20, pp. 333-364 (1973).
50. D. Prawitz, *Natural Deduction*, Almqist and Wiskell, Stockolm (1965).
51. J.C. Reynolds, *Towards a Theory of Type Structure*, Programming Symposium, Paris. Springer Verlag LNCS 19, pp. 408-425 (Apr. 1974).
52. J. C. Reynolds, *Types, abstraction, and parametric polymorphism*, IFIP Congress'83, Paris (September 1983).
53. J. C. Reynolds, *Polymorphism is not set-theoretic*, International Symposium on Semantics of Data Types, Sophia-Antipolis (June 1984).
54. D. Scott, *Constructive validity*, Symposium on Automatic Demonstration, Lecture Notes in Mathematics, vol. 125 (1970).
55. J. Smith, *Course-of-values recursion on lists in intuitionistic type theory*, Göteborg (September 1981).
56. J. Smith, *The identification of propositions and types in Martin-Lof's type theory : a programming example*, University of Goteborg Sweden (November 1982).
57. A. Tarski, *A lattice-theoretical fixpoint theorem and its applications*, Pacific J. Math. 12,2/3, pp. 242-248 (1955).