LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Playing with Lego:

Some Examples of Developing Mathematics

in the Calculus of Constructions

by

Paul Taylor

LFCS Report Series

ECS-LFCS-89-89
(also published as CSR-306-89)

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

August 1989

Copyright © 1989, LFCS

Playing with Lego: Some Examples of Developing Mathematics in the Calculus of Constructions

Paul Taylor

1 Introduction

The Calculus of Constructions (CC) of Coquand and Huet [1] is a lambda calculus with dependent types, devised to provide an expressive language for the formal development of mathematics. By exploiting the notion of propositions as types it is possible to view the type theory as a higher order intuitionistic logic with a universal quantifier and implication. This would not be a very rich theory, were it not for the fact that the type theory is impredicative. This means the definition of an object may implicitly refer to itself. New propositions can be created by universally quantifying over propositions, thus the proposition `True` is defined by the universal property $\forall P : \mathit{proposition}.P \Rightarrow P$. But impredicativity has to be treated with care, as the danger of paradox and inconsistency lies close at hand. Coquand [3] proved strong normalisation for the Calculus of Constructions, so CC is consistent and the forms of impredicativity it allows are safe. Allowing impredicativity has its reward: it is now possible to extend the calculus with definitions and rules for all of the connectives needed for intuitionistic propositional logic: `and`, `or`, `not` and `exists`. Moreover the same techniques provide the means for defining all the usual forms of data: natural numbers, lists, trees etc. Thus a minimal kernel provides us with a logic having all the ingredients necessary for doing mathematics.

A proof by natural deduction of a proposition in this logic, is a lambda calculus term whose type represents the proposition. And the top down development of such a proof, by refinement from an initial goal, corresponds to a top down construction of a functional program. Thus the proof task has been translated into a programming task. Now programming would be hopeless without tools for syntax and type checking, and similarly developing mathematics within Constructions would not get far were we confined to the use of pencil and paper. The LEGO system provides a set of tools to support Constructions: a syntax and type checker,

a mechanism to maintain a context of definitions and assumptions, and a proof assistant for refinement proofs. A full description may be found in [12, 9].

Surprisingly perhaps, mathematics may be expressed in Constructions in a fairly conventional way. In [2] Coquand and Huet show how several interesting fragments of mathematics may be developed in Constructions. In this paper we discuss a further collections of examples - in the main distinct from those of [2] developed using the LEGO implementation. The examples cover a variety of topics and hopefully give a broad insight into the style of formalisations allowed in Constructions. We will outline the key definitions and lemmas (formulae introduced by the keyword `Goal`, all of which have been successfully proved). It would be nice to also indicate the detailed proofs, however documenting a formal proof presents difficulties as neither the proof term nor the script of the proof commands, provides adequate informal documentation of the reasoning in the proof.

The examples are of moderate size, depending on a longer chain of definitions than most of the examples of [2], but by the standards of conventional mathematical practice all our examples are elementary. Also all of the examples are still to some extent toy examples produced to confirm some idea or to illustrate some technique and as such not necessarily susceptible to further development or application as they stand. However one of the challenges of formal mathematics is to devise the appropriate formalisation, and this is to some extent necessarily an experimental process. LEGO provides a system wherein such experimentation starts to become enjoyable. You can state definitions and prove the theorems in a natural manner. Of course simple theorems will frequently require more work than initially envisaged, but perhaps this just reflects how much is taken for granted in conventional presentations of mathematics.

1.1 A quick review of some basic notions

In a type theory certain terms of the theory define the types of other terms. Thus the terms in the theory are stratified into layers, where the terms in one level are typed by terms in the level above. The original Calculus of Constructions has three levels of entities. Terms originate from a single constant term `Prop`, representing the type of propositions, lying in the top layer of the type hierarchy.

> At the base is the level of proofs or objects,
> which verify propositions or propositional functions
> which are classified by terms at the level of `Prop`

So to give a specific example, (the details of the syntax are explained overleaf)

The Identity function [A:Prop][x:A]x
verifies the True proposition {A:Prop}A->A
which being a proposition has the type Prop

By adding an infinite sequence of constants Type(0), Type(1), etc. the levels in the type hierarchy may be continued upwards indefinitely. This Extended Calculus of Constructions (ECC) has been proved to be consistent [8, 7], but it is predicative at the Type levels e.g. universally quantifying over the objects of Type(0) results in an object of Type(1). Of course definitions can be made generically for Type's independently of the absolute level, and indeed it is possible to introduce a notion of typical ambiguity [5] so that we can use a single constant Type to stand for all of constants Type(i). A type checking algorithm that prevents the level constraints from being subverted has been implemented. In addition Prop may be viewed as signifying a particular class of types, hence all definitions relevant to Type will also be applicable for Prop. LEGO implements these extensions to Constructions, and this paper will exploit them.

Starting from Prop, further closed objects, denoting functions or entities that type functions (function spaces or Pi-Types), may be formed by abstraction. A typing in square brackets e.g. [x:A] represents functional abstraction (lambda binding), whilst functional type formation (Pi binding) is represented by a typing in curly brackets, {x:A}. Thus the identity function [A:Prop][x:A]x is typed by {A:Prop}{x:A}A. We have already seen that the last expression is more usually written as {A:Prop}A->A, as conventionally {x:A}B is represented by A->B when the type of the output value is independent of the input value. Constructions is a functional calculus, so we may also form objects by applying functions to arguments with appropriate types. As usual this is indicated syntacticly by juxtaposition: all operations will be prefixed apart from ->.

When Constructions is viewed as a logic, with propositions as the types of proofs, then {x:A}(B x) represents a universally quantified predicate and X->Y represents an implication. To proceed further, definitions of the logical connectives are needed. The definition of logical connective and is typical

```
[and [A,B:Prop] = {C:Prop}(A->B->C)->C]
```

Here = binds the identifier and to the term [A,B:Prop]{C:Prop}(A->B->C)->C Note LEGO's syntax for definitions allows binding variables to appear to the left of = by analogy with the usual syntax for declaring functions in programming languages. Strictly we should write

```
[and = [A,B:Prop]{C:Prop}(A->B->C)->C]
```

To see how this definition captures the logical notion of *And*, we must consider what is the operational relevance of this logical connective. Now a knowledge of

the *truth* of and A B will allow you to proceed to a proof a proposition C, when you can form a proof of C under the assumption of both the *truth* of A and the *truth* of B. Note this is one of the forms of the *And Elimination* rule. Our definition certifies a term to be proof of and A B exactly when it embodies such an operation. Of course the usual method for asserting the truth of and A B is to prove it by the means *And Introduction* rule. This rule is expressed by the proposition

```
{A,B:Prop}A->B->(and A B)
```

which may be proved, as this type is inhabited by the term

```
[A,B:Prop][a:A][b:B][C:Prop][prf:A->B->C]prf a b
```

so this term implements And_Intro. This term is actually given the name pair as it may also be viewed as an operation which pairs a and b.

Similarly the other logical connectives may be defined by the elimination rule they embody, and the dual introduction rule will be expressed by a provable proposition. We give below the forms defining the remaining logical operations. It is a useful exercise to write down the statement and proofs of the introduction rules.

```
[or [A,B:Prop] = {C:Prop}(A->C)->(B->C)->C]
(* Existential quantifier *)
[Ex [A:Type][B:A->Prop] = {C:Prop}({x:A}(B x)->C)->C]

[absurd = {C:Prop}C]
[not [A:Prop] = A->absurd]

(* Leibnitz Equality *)
[Q [A:Type] [a1,a2:A] = {P:A->Prop}(P a1)->(P a2)]
```

In practice as these and a few other facts of logic are so widely used they are predefined in a file and loaded at the start of a session. Thus proofs will usually be carried out in the context of some definitions. Moreover, it is frequently the case that several theorems will depend upon the same set of assumptions: e.g. they all concern rings and so assume the ring axioms. So the system allows for global proof contexts containing assumptions, as well as definitions. After a sequence of proofs, the assumptions may be discharged and the system will make the appropriate abstractions. Examples of this process occur in the following sections.

2 The Binomial Theorem

This is the well known identity expressing the sum of x and y raised to an *n-th* power as a sum of products of powers of x and y. i.e.

$$(x+y)^n = \sum_{i=0}^{n} B_i^n x^{n-i} y^i$$

Where B_i^n denotes the *i-th* binomial coefficient of degree n. This identity is valid for any commutative semiring, and our proof will be relative to this general setting. The presentation of the identity in LEGO syntax, takes the following form:-

```
Goal {x,y|U}{n|nat}(ind_nat n)
     ->
     eq (power_mult n (add x y))
        (add_to  (Suc n)
                 [i:nat]power_add (binomial n i)
                                  (mult (power_mult (Sub n i) x)
                                        (power_mult i y)))
```

This looks unfamiliar and much more verbose, but perhaps is not really so bad. The problems mainly stem from the uniformity of our syntax: all operations are explicit, are prefixed and have names (which could in this case be shorter). It could be argued that in other respects such a uniform notation makes things clearer. For instance, the expression nx is represented by `power_add n x` with this notation, which explicitly recognises it as n-fold repeated ring addition, and analagous to x^n which is `power_mult n x`, an n-fold repeated ring multiplication.

The explicitness demanded by the formal presentation is felt even more strongly in the development of the proof. This is a fairly straightforward manipulation of the equality, via the use of many well known identities. The chief task of the proof is to formulate and prove these, and indeed this involves more work than one would initially anticipate, as informal proofs take for granted, in the very notation, certain manipulations (e.g. applications of the associative laws). Thus whilst in some ways this is the easiest example, it embodies the longest proof achieved to date with the LEGO system. It was necessary to develop libraries of lemmas for natural numbers, monoids and summation of series etc. Building a useful library of lemmas requires some skill: it is a demand driven task, as it is impossible to anticipate every lemma one will need, and yet one tries to select a form for each lemma which will be of most general application. Our description of the proof is an overview of the contents of these libraries.

2.1 A Context for Semirings

The formulation and proof of the binomial theorem occurs within a context, of objects and assumptions sufficient to define a commutative semiring. This is built in stages, as a semiring consists of both an additive and a multiplicative monoid. Thus it makes sense to develop a general theory of monoid operations, which can be specialized for the two operators.

Throughout the proof we keep the context below to represent the carrier type with equality for an algebra, (which will be in turn a monoid and a semiring).

```
[U|Prop]
[eq|U->U->Prop]
[eq_refl:{x:U}eq x x]
[eq_sym:{x,y|U}(eq x y)->(eq y x)]
[eq_trans:{x,y,z|U}(eq x y) -> (eq y z) -> (eq x z)]
```

In addition we must set up an additional fragment of context to describe an associative binary operation with identity i.e. a monoid.

```
[BinaryOp = U->U->U]
[op|BinaryOp]

[Congruence
   ={x1,y1,x2,y2|U}
       (eq x1 y1)->(eq x2 y2)->eq (op x1 x2) (op y1 y2)]
[congr:Congruence]
[Associativity = {x,y,z|U}eq (op x (op y z)) (op (op x y) z)]
[assoc:Associativity]

[id|U]
[LeftIdentity = {x|U}eq x (op id x)]
[ident_left:LeftIdentity]
```

Within this context we prove the theorems, concerning powering and iterating op over a series of values (i.e. summation of a series), discussed in section 2.3. By discharging the context back to op, using the command `Discharge op`, the definitions and theorems are rendered parametric with respect to any associative binary operation. So we may set up the the ultimate context for the semiring as follows:-

```
(* Monoid Structure for add *)
[add|BinaryOp]
[congr_add:Congruence|add]
```

```
[assoc_add:Associativity|add]
[zero|U]
[ident_left_add_zero:LeftIdentity|add|zero]

(* Monoid Structure for mult *)
[mult|BinaryOp]
[congr_mult:Congruence|mult]
[assoc_mult:Associativity|mult];
[unit|U];
[ident_left_mult_unit:LeftIdentity|mult|unit]

(* Additional axioms for commutativity of add and mult*)
[Commutes [op|BinaryOp] = {x,y|U}eq (op x y) (op y x)]
[comm_add:Commutes|add]
[comm_mult:Commutes|mult]
(* Distributivity Axiom *)
 [Distributes [op1,op2|BinaryOp]
  = {x,y1,y2|U}
      eq (op1 x (op2 y1 y2)) (op2 (op1 x y1) (op1 x y2))]
 [distrib_mult_add:Distributes|mult|add];
```

All the definitions and theorems created relative to the monoid context may be specialised for `add` and `mult`, because of the unification mechanism of LEGO there is no necessity to do this for everything. However it is convenient for the statement of theorems to have some specific names for certain specializations. e.g.

```
[power_add = power|add|zero]

[add_to = iterate_to|add|zero]
[add_from_to = iterate_from_to|add|zero]

[power_mult = power|mult|unit]
```

2.2 The natural numbers

Even though we are proving the binomial theorem for general case of commutative semirings, the result depends strongly upon the theory of natural numbers. It requires n-th powers for the ring addition and multiplication, and the result itself may be viewed as sequence of identities indexed by the natural numbers.

In the Calculus of Constructions, following the ideas of Church, the natural numbers may be defined as a family of combinators, i.e. the natural number n is the

operation of n-fold composition f^n, where f is a single argument function. The type of natural numbers, and the zero and successor objects have the definitions:

```
[nat = {C|Prop}(C->C)->C->C]
[Zero = [C|Prop][step:C->C][init:C]init:nat]
[Suc[n:nat] = [C|Prop][step:C->C][init:C]step (n step init):nat]
```

This specification of natural numbers allows us to define all the functions we might need and to perform computation by normalizing lambda terms. However it does not guarantee that we have the canonical natural numbers, we cannot be sure that all objects in nat can be constructed using Zero and Suc, hence Peano's induction axiom is not provable. Nevertheless we may restrict our attention to the subset of nat generated by Zero and Suc, and the definition of this subset is essentially Peano's axiom. Hence we call it the set of inductive nat's.

```
[Suc_invariant [P:nat->Prop] = {n:nat}(P n)->(P(Suc n))]
[n_ind_prop [P:nat->Prop] = and (P Zero) (Suc_invariant P)]
[ind_nat [n:nat] = {P|nat->Prop}(n_ind_prop P)->(P n)]
```

Whilst the definition of ind_nat encodes Peano's axiom it is not quite strong enough: we will prove properties not for arbitrary nat's but for inductive nat's, thus ind_nat m must occur as a side condition for all natural number variables m, which occur in our proofs. This situation is captured by the following definition and theorem, which may be thought of as the Peano's induction principle.

```
[natInductionProp [P:nat->Prop]
        = {n:nat}(ind_nat n)->(P n)->P (Suc n)]

Goal {P|nat->Prop}{n|nat}(ind_nat n)->
        (P Zero) -> (natInductionProp P)->(P n)
```

Just as the definition of inductive nat doesn't initially appear strong enough to enable all inductive proofs, similarly the definition of nat's as iterators seems insufficient to encompass all functions definable by primitive recursion. However, just as we may derive a rule for proof by induction, so we may define a combinator to implement primitive recursion.

```
[C|Prop]
[aux [rec_step:nat->C->C] [init:C] [n:nat]
   = [step = [store:and nat C]pair (Suc(fst store))
                                   (rec_step (fst store) (snd store))]
     [initial_store = pair Zero init]
     n step initial_store]
```

```
[prim_rec_nat [rec_step:nat->C->C] [init:C] [n:nat]
     = snd (aux rec_step init n)]
```

We can prove, by induction, that functions defined via this combinator satisfy the constraint for primitive recursion.

```
Goal {rec_step|nat->C->C}{init|C}{n|nat}
       (ind_nat n)
        ->
        Q (rec_step n (prim_rec_nat rec_step init n))
          (prim_rec_nat rec_step init (Suc n))
```

With these preliminaries, a host of standard elementary number theoretic identities may be established via the traditional inductive proofs. A table of some of the more familiar of these is given below. Note that certain of the results for subtraction have to be conditional upon the standard ordering relation for natural numbers, `Less_or_Eq`. This may defined and shown to have the desired properties, within the framework established above. Here is its definition.

```
[Less_or_Eq [n,m:nat]
  = {P|nat->Prop}(Suc_invariant P)->(P n)->(P m)]
```

2.3 Properties of Monoids: Powers and Summation of Series

Crucial to the statement and proof of the binomial theorem, is an understanding of the repeated application of the binary operation of a monoid: both with multiple copies of a single element (powering) and with sequences of elements (iterating, which is summation of a series when the operation is addition). Of course the first operation may be defined as the specialization of the second to a constant sequence. However proving properties in the special case first is technically much simpler, and yet provides the essential form of the more general proofs.

Powering is defined directly via the iterator implicit in a natural number. Iterating needs primitive recursion, `iterate_to n seq` will iterate the binary operation over the elements of the sequence indexed by numbers less than n.

```
[power [n:nat][x:U] = n (op x) id]

[iterate_to  [n:nat] [seq:nat->U]
   =  [op_elem = [index:nat] [accum:U]op (seq index) accum]
      prim_rec_nat op_elem id n]
```

```
(* Definition and Identities for Addition *)
 [Add [n,m:nat] = n Suc m]

 Goal {n|nat}Q n (Add Zero n)
 Goal {n,m|nat}Q (Suc(Add n m)) (Add (Suc n) m)
 Goal {n|nat}(ind_nat n)->Q n (Add n Zero)
 Goal {n,m,o|nat}(ind_nat n)
       ->Q (Add n (Add m o)) (Add(Add n m) o)
 Goal {n,m|nat}(ind_nat n)->(ind_nat m)
      ->Q (Add n m) (Add m n)

 (* Definition and Identities for Multiplication *)
 [Mult [n,m:nat] = n (Add m) Zero]

 Goal {n|nat}Q Zero (Mult Zero n)
 Goal {n,m|nat}Q (Add m (Mult n m)) (Mult (Suc n) m)
 Goal {n|nat}(ind_nat n)->Q Zero (Mult n Zero)
 Goal {n|nat}(ind_nat n)->Q n (Mult n One)
 Goal{n,m,l|nat}(ind_nat n)->(ind_nat l)
      ->Q  (Add (Mult n l) (Mult m l))(Mult (Add n m) l)
 Goal {n,m,l|nat}(ind_nat n)->(ind_nat m)->(ind_nat l)
        ->Q (Mult n (Mult m l)) (Mult (Mult n m) l)
 Goal {n,m|nat}(ind_nat n)->(ind_nat m)
       ->Q (Mult m n) (Mult n m)

(* Definitions and Identities for Predecessor and Subtraction *)
[Pred [n:nat] = [K = [x,y:nat]x]prim_rec_nat K Zero n]
[Sub  [n,m:nat] = m Pred n]

 Goal {m,n|nat}(ind_nat m)->(ind_nat n)
       ->Q  n (Sub (Add m n) m)
 Goal {n,m|nat}(ind_nat m)->
       (Less_or_Eq m n)->Q n (Add (Sub n m) m)
 Goal {n,m|nat}(ind_nat n)->(ind_nat m)
       ->Q Zero (Sub n (Add n m))
 Goal {n,m|nat}(ind_nat m)->
      (Less_or_Eq m n) ->Q Zero (Sub m n)
```

Figure 1: Identities for Natural Numbers

We have to prove that powering respects the equality of monoid elements, this will be an inductive proof which in effect exploits, n times, the fact that op is a congruence. There is a corresponding property for `iterate_to`, note that this is conditional on the equality of the sequences for values less than n

```
Goal {n|nat}{x,y|U}(ind_nat n)
     ->
    (eq x y)->eq (power n x) (power n y)
```

```
Goal {n|nat}{seq,seq':nat->U}(ind_nat n)
    ->
    ({i:nat} (ind_nat i)->
             (Less_or_Eq i (Pred n))->eq (seq' i) (seq i))
    ->
    eq (iterate_to n seq') (iterate_to n seq)
```

We need the standard identities, which show how powering interacts with the arithmetic operations i.e. $x^{n+m} = x^n x^m$ and $x^{nm} = (x^n)^m$. It is interesting to note that these identities are simply generalisations the associativity law. Their statement in LEGO is the following:

```
Goal {n,m|nat}{x|U}(ind_nat n)
    ->
    eq (op (power n x) (power m x)) (power (Add n m) x)
```

```
Goal {n,m|nat}{x|U}(ind_nat n)->(ind_nat m)
     ->
     eq (power n (power m x)) (power (Mult n m) x)
```

The corresponding result for `iterate_to` explains how op'ing over a sequence may be decomposed into op'ing over adjacent subintervals of the sequence. But to do this, the iterated operation must be defined for subintervals starting at indices other than Zero; this is the operation `iterate_from_to`. So we have:

```
[iterate_from_to [n1,n2:nat] [seq:nat->U]
      = [seq' = [i:nat]seq (Add n1 i)]
        iterate_to (Sub n2 n1) seq']
```

This definition applies `iterate_to` to `seq'`, a locally defined sequence which translates seq to the start of the interval. The identities which show how `iterate_to` and `iterate_from_to` may be decomposed follow.

```
[seq|nat->U]
Goal {n,m|nat}(ind_nat n)->(ind_nat m)
      ->
      eq (op (iterate_from_to m (Add m n) seq)
             (iterate_to m seq))
         (iterate_to (Add m n) seq)

Goal
 {n,d1,d2|nat}(ind_nat n)->(ind_nat d1)->(ind_nat d2)
 ->
 eq (op (iterate_from_to (Add  n d1) (Add(Add n d1) d2) seq)
        (iterate_from_to n (Add  n d1 ) seq))
    (iterate_from_to n (Add n (Add d1 d2)) seq)
```

These results are straightforward, but their formal proofs in LEGO require care, they would probably be eased if the system had higher order pattern matching.

2.4 The Binomial Coefficients

The binomial theorem revolves upon the fact that it is possible to give an explicit expression for the integer coefficients of the powers of x and y in the expansion of $(x+y)^n$. Conventionally these binomial coefficients are expressed via the factorial function so $B_m^n = !n/!m!(n-m)$. Whilst it would be possible to do this, in our presentation we take the more direct route of defining the coefficients by the recursion `binomial n m = binomial (n-1) m + binomial (n-1) (m-1)`. To do this formally we exploit the fact that the natural numbers are a polymorphic type and correspond to iterated composition over all types. So by defining an initial function for `binomial Zero`, which has type `nat->nat`, and thinking of the recurrence relation as a step function with type `(nat->nat)->(nat->nat)`, the binomial coefficients of degree n will be the n-fold composition of the recurrence relation. So we have:-

```
[binomial_zero = [i:nat] if (is_positive i)  Zero
                         One]

[binomial_step [f:nat->nat]
    = [i:nat]if (is_positive i)
                    (Add (f i) (f (Pred i)))
             One]

[binomial [n:nat] = n binomial_step binomial_zero];
```

```
Goal {n:nat}(ind_nat n)->Q One (binomial n Zero)

Goal {n,m:nat}(Positive m)->
       Q (Add (binomial n m) (binomial n (Pred m)))
         (binomial (Suc n) m)

Goal {n:nat}(ind_nat n)->
     {m:nat}(Less n m)->Q Zero (binomial n m)

Goal {m|nat}(ind_nat m)->
     {n|nat}(Less_or_Eq m n)->Positive (binomial n m)

Goal {n|nat}(ind_nat n)->Q One (binomial n n)
```

Figure 2: Identities for Binomial Coefficients

Note that these definitions depend upon a theory of Boolean data values, they use the conditional `if` and a test for positive natural numbers, `is_positive`. Defining Booleans in CC uses the same techniques as defining nat's: Bool's embody the computation process which selects one value from two. Previous lemmas have involved the predicate `Positive`, we can prove that this predicate and the test `is_positive` are related in the expected way i.e.

```
{n:nat}(Positive n)->Q true (is_positive n)
```

Again a number of standard identities must be proved for the binomial coefficients, these are listed in figure 2.

2.5 Semirings and the Binomial Theorem

It was noted previously that the identities for manipulating power and `iterate_to` were generalised forms of the associative rule for monoids; similarly, to cope with sequences of elements from a commutative semiring, generalized forms of the commutative law and distributive law are required.In traditional notation these would be:

$$\sum_{i=0}^{n} x_i + \sum_{i=0}^{n} y_i = \sum_{i=0}^{n} x_i + y_i$$

$$x \sum_{i=0}^{n} y_i = \sum_{i=0}^{n} x y_i$$

Their expression in LEGO's notation is as follows:

```
(* The generalised Commutative Law for add *)
Goal
{n|nat}{seq1,seq2|nat->U}(ind_nat n)
     ->
     eq (add (add_to n seq1) (add_to n seq2))
        ([seq' = [i:nat]add (seq1 i) (seq2 i)]
         add_to n seq')

(* The generalised Distributive Law *)
Goal
{n|nat}{x|U}{seq|nat->U}(Positive n)
   ->
   eq (mult x (add_to n seq))
      ([seq' = [i:nat]mult x (seq i)]
       add_to n seq')
```

Note that as we are working in a semiring (so we don't assume an additive inverse) we do not know that `eq (mult x zero) zero` so the distributive law is certain only for summation over non-empty intervals.

We have indicated most of the identities needed for the proof of the binomial theorem, (plus a few added for completeness and minus others not quoted for obscurity). A straightforward proof by induction of the binomial theorem can now proceed. First rewrite $(x+y)^n$ as $(x+y)(x+y)^{n-1}$, then using the induction hypothesis $(x+y)^{n-1}$ is rewritten as the summation of a series. Next, by the use of distributivity, the resulting formula is split into four chunks, which can be massaged back into the binomial expansion for n. As this verbal sketch suggests, this process, whilst notionally quite straightforward, becomes very messy when described in precise formal detail. The fact that it is so easy in a conventional proof indicates how well adapted classical mathematical notation is for the calculations of analysis.

As LEGO has no automatic tactics all the rewrites had to be explicitly specified, an automatic rewriting tactic might have been of assistance. However for this theorem a sophisticated tactic would be required: it would need higher order matching to recognise sequence forms, controlled use of commutativity, and an ability to deal with conditional equalities.

3 Some elementary topology

In Constructions it is possible to do naive set theory, a set being thought of as the predicate (propositional function), which selects elements of the set. Here we show how two elementary results of general topology may be formulated in this set theory: the first verifies that the topology defined by a basis is indeed a topology, and the second that a topology on a type induces a topology upon any subtype. The proofs in essence, revolve about the fact that we can reorder quantifiers, however the proofs are somewhat tricky because of the levels of types involved, we have families of families of sets.

As these are probably the two simplest results of general topology it is easy to list other results we could tackle. However it should be noted that the logic of Constructions is intuitionist and lacks the excluded-middle, so certain identities between sets will be missing. Thus further development of topology would quickly run into problems. Of course the excluded middle could be assumed as an axiom, or alternatively we could develop topology within an encoding of some classical set theory.

3.1 Naive Set Theory

Sets defined over a type are the propositional functions over that type, so:

```
[U|Type]
[set = U->Prop]
```

Operations upon sets are defined by applying the standard logical operation pointwise, and we get the usual definitions for intersection and union etc.

```
[insct [A,B:set] = [x:U]and (A x) (B x) : set ]
[union [A,B:set] = [x:U]or (A x) (B x) : set ]
[compl [A:set] = [x:U]not (A x) : set]

[void = [x:U]absurd : set ]
[universe =  compl void ]
```

Similarly we may readily define relations of inclusion and equality of sets.

```
[incl [A,B:set] = {x:U}(A x)->(B x) : Prop ]
[eq [A,B:set] = and (incl A B) (incl B A) : Prop ]
```

These are defined relations, so it is necessary to verify that the first is a partial order and the second an equivalence, i.e.

```
Goal {a|set}incl a a
Goal {a,b,c|set}(incl a b)->(incl b c)->(incl a c)

Goal {a|set}eq a a
Goal {a,b|set}(eq a b)->(eq b a)
Goal {a,b,c|set}(eq a b)->(eq b c)->(eq a c)
```

However the proofs are straightforward, the properties being essentially inherited from the corresponding properties of the logical operations underlying their definitions. Similarly a whole host of trivial identities (e.g. commutativity of intersection) may be stated and proved for the set operations. Many of the standard identities may be proved, but as noted previously some will be missing. Thus we can prove

```
{A:set}incl A (compl(compl A))
```

but can't prove

```
{A:set}incl (compl(compl A)) A
```

Our ultimate aim is to do equational reasoning with sets. Thus in addition to proving the basic identities we must prove that each operation we define respects set equality. For instance, for intersection we have

```
Goal {A1,A2,B1,B2|set}(eq A1 A2)->(eq B1 B2)
                        ->eq (insct A1 B1) (insct A2 B2)
```

3.2 Classes and Topologies

Sets may be defined over any type, thus as `set` is itself a type we may form sets of sets, i.e. classes. But first we must discharge `U` from the context, to allow all the set definitions and theorems to take a type as a parameter. However it is still useful to work in a context, which has a name for the underlying universe. Thus we invoke the commands

```
Discharge U; [U|Type]
```

or alternatively the single command

```
DischargeKeep U
```

Classes may now be introduced via the following definition

```
[class = set|(set|U)]
```

Of course all the operations and properties of sets will also hold for classes, but further operations may be defined. For instance all the subsets of a set form a class

```
[Pwr_cls [A:set|U] = [B:set|U]incl B A : class]
```

More importantly for our purposes, we can define the construction of the set which is the global intersection of the sets in a class, and similarly the global union of the sets in a class.

```
[Insct [C:class] = [x:U]{A:set|U}(C A)->(A x) : set|U ]
[Union [C:class] = [x:U]Ex [A:set|U]and (C A) (A x) : set|U ]
```

We have now framed sufficient definitions to express the standard notion of a topology; i.e. a class of sets, the open sets, that contains the empty set and the universal set and which is closed with respect to intersection and unions over classes of open sets. With our notation, this is written

```
[topology [open:class]
  = and4 (open (universe|U)) (open (void|U))
         ({a,b:set|U}(open a)->(open b)->open (insct a b))
         ( {cv:class}(incl cv open) -> open (Union cv))]
```

(Note that all the topologies forms a set of classes, so we could proceed to further levels of generalisation if we wished.) Similarly a basis is a class constrained by certain properties: the union of all the basic sets is the universe and the intersection of any pair of basic sets is covered by the union of a collection of basic sets. We express this formally by

```
[basis [basic:class]
    =   and (eq (universe|U) (Union basic))
            ({a,b:set|U}(basic a)->(basic b)
                  ->Ex [cv:class]and (eq (insct a b) (Union cv))
                                     (incl cv basic))]
```

Now from a basis it is possible to construct another class of sets, which may be shown to define a topology. (This is their significance: for whilst it is generally simpler to state and prove results using the viewpoint of a topology, it usually more convenient to use a basis to define the open sets of a topology.) The construction for basic topology from a basis is defined as follows:

```
[basic_top [basic:class]
    =  [s:set|U]
         Ex [cv:class]and (eq s (Union cv)) (incl cv basic)
           :class]
```

and the theorem we need, so as to verify that this construction defines a topology is

```
Goal {basic:class}(basis basic)->(topology (basic_top basic))
```

3.3 The Key Lemmas

The informal proof that a basis defines a topology is quite short, it relies on two observations. Firstly, that the intersection of the Unions of two classes of sets is the Union of intersections of pairs of sets, one from each class. Secondly that a Union of Unions of classes is the Union of a single aggregated class. The formal expression of these properties requires some care as we must provide an explicit formula for the class over which we form the Union. Here is formulation of the first of these.

```
Goal {cv1,cv2|class}
     [icv =[s:set|U]Ex2  [a,b:set|U]and (eq s (insct a b))
                                        (and (cv1 a) (cv2 b))]

     eq_set (insct (Union cv1) (Union cv2)) (Union icv)
```

The local definition icv is the class of intersections of pairs, it uses Ex2 a two valued existential quantifier. (This is just for convenience, we could have used two nested Ex's.)

Here is the formulation of the second lemma.

```
Goal {attr|class}{cvo|class}
     [ucv = [si:set|U]
            Ex [so:set|U]
                and (cvo so) (Ex [cvi':class]
                                 and3 (eq so (Union cvi'))
                                      (cvi' si)
                                      (attr si))
     ]
      ({s:set|U}
       (cvo s)-> Ex [cvi:class]and (eq s (Union cvi)) (incl cvi attr))
       ->
      eq_set (Union cvo) (Union ucv)
```

The local definition ucv is a class which is the aggregate of classes. It is difficult to express informally the content of this definition. The idea is that a set has an outer cover cvo which is a class of sets, each of which has an inner covering by a class of sets cvi. Our definition turns this situation inside out, so we refer to sets si lying in an innermost cover in the outermost scope. The situation is further complicated by the fact that we have to maintain as a side condition that each of our sets has the attribute attr. (In the topology proof this will be that the sets belong to the basis.)

The proof of both of these, and the second in particular, require some care as the search space becomes very wide at certain points. It is in this situation that a knowledge of the semantics of what we are trying to prove, here the distinction between inner and outer sets, prevents us from exploring fruitless paths. In some sense, this difficulty is surprising, for as always with set identities we are really restating straightforward logical identities: in this case that the existential quantifier distributes over disjunction and commutes with itself.

3.4 The subspace topology

Subtypes

The subspace topology over a set P of points from type U equipped with a topology is formed by restricting open sets to their intersection with P. The proof that these restricted open sets also form a topology, is essentially transmitted through this construction. However, a formal demonstration of this presents an interesting technical problem: the definition of topology requires that the space of points U be a Type, so we must define a subtype corresponding the set P.

A natural notion for an object typed by the subtype corresponding to P, would be a pairing of an object of U along with a proof that it lies in the set P i.e. a pair (x,p) where p:P x. However such a pair cannot be typed by a simple product type, as the typing of its second component depends upon the value of the first component. We require a dependent product type constructor, analogous to the Σ type constructor (a form of parametrised set summation, hence Σ) of Martin-Löf type theory. Whilst such a constructor may be safely added to the ECC, this example exploits an alternative approach. The proof rule for existential-introduction packages an object x and a proof of P x into a single object which is a proof of Ex P. But such proofs are inadequate for representing dependent pairs, as it is impossible to define the projection functions which extract the components of the pair. In the original CC this behaviour was essential, as Coquand showed that a dependent product with both projections allowed for the construction of an explicit inconsistency. But having the just the first projection is safe: and indeed a slight variation of the existential quantifier will have the first projection. i.e. we

have a weak sum.

```
(* Existential quantifier *)
[Ex [A:Type][B:A->Prop] = {C:Prop}({x:A}(B x)->C)->C]

(* Weak Sum *)
[Sig [T|Type][P:T->Type] = {B|Type}({t:T}(P t)->B)->B : Type]

[T|Type][P:T->Prop]
(* dependent pairing *)
[d_pair [wit|T][prf:P wit]
        = [B|Type][gen:{t:T}(P t)->B](gen wit prf) : Sig P]

(* first projection *))
[proj1 [p:Sig P] = p  [x:T][y:P x]x : T];
 Discharge T
```

The subtyping construction is simply a renaming of Sig i.e.

```
[Subtype [P:set|U] = Sig P]
```

In general this is not a good notion of subtype, as the inaccessibility of the second component prevents us from exploiting the constraint which defines the subtype. Thus, for instance, we could not prove that the subtype of square matrices with vanishing determinant forms a group, if we used this subtype construction. However for the subspace topology this is not a problem: any set P leads to a subspace topology and the exact form of P is irrelevant to all our definitions and proofs.

Restricting Sets and Classes

With this notion of subtype, we can define the restriction of a U set to a Subtype P set, by testing that the first projection of an element lies in the overlying set.

```
[P:set|U]
[UP = Subtype P]
[restrict [S:set|U] = [u':UP]S (proj1 u'):set|UP]
```

A slightly more obscure definition must be provided for the restriction of classes of sets.

```
[c_rstrct [C:class|U]
     = [S:set|UP]
         Ex [S':set|U]and (eq S (restrict S')) (C S')
       :class|UP]
```

This captures the following informal definition: the set S lies in the restriction of the class C, if it is the restriction of some set S' that lies in the class C. Other formulations of class restriction are possible but this is the one needed to obtain the subspace topology theorem i.e.

```
[open:class|U];
Goal (topology|U open) -> (topology|UP (c_rstrct open))
```

The observation made earlier that properties of the topology are essentially transmitted through the subspace restriction, is formally captured by the lemmas which show that set operations commute with the appropriate restrictions. In particular for intersection of sets and Unions of classes we have:

```
Goal
{S,T:set|U}eq (restrict(insct S T)) (insct (restrict S) (restrict T))

Goal {C:class|U}eq (Union(c_rstrct C)) (restrict(Union C))
```

These are readily proved, as the definitions of the restrictions refer back a set construction in the subtype UP to the construction in the type U: so essentially both sides of the equalities are identical. However the proof of the main theorem does hold a small surprise. It is essential to prove the following

```
Goal
  {c':class|UP}{c:class|U}
   (incl c' (c_rstrct c))
    ->
    Ex  [lftc':class|U]and (eq c' (c_rstrct lftc')) (incl lftc' c)
```

i.e. a class c' included in the restriction of a class c may be covered by a class lftc' included in c. This is needed to ensure that families of open sets in the subspace, may be placed in correspondence with a families of open sets in the original space. The proof can proceed as an explicit candidate for the lifted class may be defined. However to complete the proof we need the hypothesis shown below.

```
[c_resp_eq:{U|Type}{c:class|U}{s,t:set|U}(eq s t)->(c s)->(c t)]
```

This lets us assume that equal sets lie in the same classes. That this seemingly obvious property requires proof is a symptom of the constructive nature of our mathematics. Individual sets and classes must be defined by explicit constructions, and two equal sets may be the result of distinct constructions. Thus we will require a proof that they lie in the same class: a proof which we assume in this instance.

4 Warshall's Algorithm

The results described here are a translation into LEGO of some work of Christine Paulin-Mohring [10], who used the INRIA implementation of Constructions. It thus served as a comparison between proof styles in the two systems. The INRIA proof was cast in terms of appeals to tactics, and was of little help in deciding the refinements necessary for the LEGO proof. However the chain of lemmas needed to achieve the main result was much the same in both cases. Ultimately the LEGO proof was quickly discovered, as it corresponds quite closely to an informal presentation of the proof. However the resulting proof script is probably as impenetrable, as that of the original tactic based proof: illustrating the difficulties that proof documentation presents.

The result is quite interesting in itself. It was developed as an example of how from different proofs of the same proposition, alternative programs may be extracted. It is proved that for a decidable relation defined over a finite set the transitive closure is also a decidable relation. In terms of graphs this means the connectivity relation can be computed from the edge relation. As our proof is constructive it embodies an implicit algorithm, which for the most direct proof will have exponential order. Walshall's algorithm is an $O(n^3)$ algorithm, which may be generated by using an array of boolean values to encode the relation. The representation of decidable relations by arrays may be formulated in Constructions. And reproving the decidability result in this setting, results in a proof for which the extracted program corresponds to Warshall's $O(n^3)$ algorithm.

4.1 Connection in a Graph: Transitive Closure

The notion of a graph may be formulated using a relation, the edge relation which defines the vertices directly connected by edges of the graph. The relation of connection between vertices is then defined inductively: two vertices are either directly connected via an edge, or the first vertex is directly connected to a vertex which is connected to the second (i.e. there is a finite chain of direct connections linking the two vertices.) In the more general setting of relations this is the transitive closure of a relation. Actually the definition we use is slightly more specific than the usual definition, as we need to restrict the vertices through which the connection may be made. Thus the relation `connected` will be parameterised by a set `W`, in which the connecting vertices must lie.

The formal definition is given below. As usual for an inductive property, we define a least fixed point. (You should compare this definition with that for inductive natural numbers). We first define the class of connected relations: those that contain the edge relation, and which are preserved by adjoining edges. Then `connected` is the minimal connected relation. Note that the definition encodes

the induction principle implicit in the connection relation.

```
[Edge:U->U->Prop]

[connected_rel [W:set] [P:U->U->Prop]
   = (and ({x,y:U}(Edge x y)->(P x y))
          ({x,y,z:U}(Edge x y)->(W y)->(P y z)->(P x z)))]

[connected [W:set] [x,y:U]
  = {P:U->U->Prop}(connected_rel W P)->(P x y)]
```

Some elementary properties of connection must be proved. There is a contravariant relationship between inclusion of the vertex sets W, and inclusion of the classes of connected relations; which in turn enables us to verify that the relation connected is monotonic with respect to set inclusion i.e. adding vertices can only increase the number of pairs we can connect. The formal statement of these follows, note we have specialised the monotonicity result to the case of adding a single vertex.

```
[W|set]

Goal {P|U->U->Prop}
       (incl W W')->(connected_rel W' P)->(connected_rel  W P)

Goal {x,y|U}(incl W W')->(connected W x y)->(connected W' x y)

Goal {a,x,y|U}(connected W x y)->(connected (add W a) x y)
```

Also by using the induction principle implicit in the definition of connected, basically induction over the chain of vertices in a connection, we can prove that the relation is transitive. Similarly we can show that if the set of vertices through which we must make the connection is empty then the connected relation is the same as the edge relation.

```
Goal {x,y,z|U}
      (connected W x y)->(connected W y z)->(W y)->connected W x z

Goal {x,y|U}(connected void x y)->(Edge x y)
```

We now have sufficient background to prove the key lemma for the proof of the Warshall algorithm. The monotonicity and transitivity of the relation connected means that if a vertex y is added to the set through which the connections are made then x is connected to z, if they were connected before, or if they both previously had a connection to y. By induction we can prove that the converse of this is also holds.

```
Goal {x,y,z:U}
       (connected (add W y) x z)
                ->(or (connected W x z)
                      (and(connected W x y) (connected W y z)))
```

The proof of this reaches a stage where choosing the appropriate move becomes confusing. Again our geometric (semantic) understanding of the problem provides the clue: the proof depends upon actually defining how we may construct a path through the graph, and it is quite clear how this may be achieved.

4.2 Finite Sets and Decidable Relations

To proceed further we need to postulate additional properties for our graphs. Firstly, although the edge relation tells us when two vertices are directly connected, additional information is needed to exclude an edge connection. Thus we must assume the edge relation is decidable by introducing the law of the excluded middle for this special case.

```
[decidable [P:Prop] = or P (not P)]
[dec_E:{x,y:U}decidable (Edge x y) ]
```

We aim to prove that, relative to this assumption, the defined relation connected is also decidable.

```
Goal {x,y:U}decidable (connected W x y)
```

Now the relation connected is established by a finite computation, which verifies that a finite chain of edges links two vertices. However unless the choice of edges is finite i.e. unless the set W is finite we stand no chance of chance of checking every path. Thus to establish the theorem we must also formalise the notion of finiteness. This is achieved by encoding the inductive definition of finite set (a finite set is either empty or formed by adding a new element to a finite set), via the standard fixed point technique.

```
[finite_attr [P:set->Prop]
  =   and (P void)
          ({A:set}{x:U}(P A)->(not(A x))->(P (add A x)))]

[finite [A:set] = {P:set->Prop}(finite_attr P)->(P A)]
```

We must postulate that all sets are finite.

```
[finite_U:{A:set}finite A]
```

As usual the inductive definition embodies an induction principle, which we can use to readily establish the decidability of connected. The base case, when the set W is void, is our assumption. For the step case we must show, under the assumption of decidability for connected W, that connected (add W y) is decidable . The proof is a nested case analysis. If connected W x z then we immediately know connected (add W y) x z. But for not (connected W x z) we must do a further case analysis with connected W x y and connected W y z. Here the following theorem will generate the necessary contradictions.

```
Goal {x,y,z|U}
      (not (connected W x z))
          ->(connected (add W y) x z)
              ->(and(connected W x y) (connected W y z))
```

(This is the key lemma noted earlier, transformed via the tautalogy identity (or A B)->(not A)->B))

4.3 The Array Algorithm

If a proposition is decidable a proof can be constructed for it or its negation: hence its truth can be encoded by a boolean truth value. This observation can be formally expressed in the theory of Constructions. Of course we must define the data type of booleans, but this is by now a fairly routine procedure. Additionally we must assume no junk and no confusion constraints for the boolean data. We can then prove

```
Goal {P|Prop}(Ex [b:bool]iff (Q b true) P)->(or P (not P))
```

```
Goal {P|Prop}(or P (not P))->(Ex [b:bool]iff (Q b true) P)
```

There should be corresponding results for decidable predicates, such that if for each x:X we can show or (P x) (not(P X)) then then the predicate P may be represented by a function with type X->bool This can be formulated in Constructions. But is proof is possible only if we introduce an additional axiom, a form of the axiom of choice.

```
[ax_of_choice:{A,B|Type}{S:A->B->Prop}
  ({x:A}Ex [y:B]S x y)->(Ex [f:A->B]{x:A}S x (f x))]
```

This axiom in quite innocuous as a constructive proof of {x:A}Ex [y:B]S x y must implicitly construct such a function, so no inconsistency can be introduced into the system. However as Ex is only an existential quantifier and not a strong

sum we are unable to prove this (in Martin-Löf type theory it may be proved). Paulin-Mohring's [11] notion of realisability for Constructions gives a rigorous justification of these observations. As the Edge and connected relations are both binary relations we actually require the binary form of the axiom of choice.

```
Goal {A,B,C|Type}{P:A->B->C->Prop}
   ({a:A}{b:B}Ex [c:C]P a b c)
    ->(Ex [f:A->B->C]{a:A}{b:B}P a b (f a b))
```

Fortunately this may be proved from the unary form of the axiom, so we need only postulate a single version of the axiom of choice.

Then the idea of representing a relation by a mapping into bool, which formalises the notion of a 2-dimensional array of bool, is captured by the following definitions and theorems.

```
[matrix = U->U->bool]
[represents [R:U->U->Prop] [M:matrix]
  = {x,y:U}iff (Q (M x y) true) (R x y)]

Goal (Ex [M:matrix]represents R M)
                    ->{x,y:U}or (R x y) (not(R x y))

Goal ({x,y:U}or (R x y) (not(R x y)))
                    ->Ex [M:matrix]represents R M
```

Thus we see that a relation is representable by an array of bool, if and only if it is decidable, and we may reformulate the decidability of the connected relation by:-

```
Goal Ex [M:matrix]represents (connected W) M
```

The proof is almost identical in form to the proof of the previous formulation of this result: it uses induction upon the construction of the finite set W and uses the same lemmas to compute the extension of the relation. However the intent of the proof is subtly different, since at each stage we construct a new array from a previous array. The case analysis is now over the two boolean values true and false, and the proof uses the specification of the input matrix to verify that the output matrix meets its specification.

5 Bisimulation

The final example is of a proof via the principle of coinduction, which is applicable to objects defined as maximal fixed points. In this case the relation of bisimulation between graphs, which is the maximal fixed point of an operation acting on relations. Coinduction is the dual notion to the more familiar induction. A proof by induction lets you assert for elements of the least fixed point any property which may be proved to be stable; dually coinduction lets you verify that an element lies in maximal fixed point by showing that has some costable property.

5.1 Maximal Fixed Points

When we have a type `X`, an equivalence relation `eq:X->X->Prop` and a mapping `F:X->X`, an object `x` of `X` is a fixed point of `F` if it is invariant under the action of `F`. i.e `eq x (F x)`. An equality may be defined from a partial ordering `le` by the definition

```
eq [x,y:X] = and (le x y) (le y x)
```

In this case the fixed points of `F` will also be ordered by `le`. So we have minimal fixed points, greatest lower bounds for the collection of fixed points , and maximal fixed points, which are least upper bounds. Objects defined as minimal fixed points are frequently used in computer science: in this report the inductive numbers, the finite sets and the transitive closure of a relation are defined by minimal fixed point constructions. Indeed the minimal fixed point will be the simplest object consistent with the constraint implicit in `F`. Similarly a maximal fixed point is the largest object which does not contradict the constraint implied by `F`. Of course the objects with type `X` often are structured: calling them points is somewhat misleading.

Tarski [13] pointed out a construction for maximal (minimal) fixed points when there is a general construction for a least upper bound (greatest lower bound) and `F` preserves the ordering. Moreover this construction embodies the proof principle of coinduction (induction).

Here we indicate the proof of Tarski's theorem in the particular case of binary relations and inclusion: of course a general proof could be produced and specialised to this case (see for instance [2]).

First we must repeat certain of the definitions of set operations for binary relations over a type `U`, as unfortunately for technical reasons we cannot exploit the fact that a binary relation is essentially a set of pairs.

```
[U:Type]
[rel = U->U->Prop]
[inclr [R,S:rel] = {x,y:U}(R x y)->(S x y)]
[Unionr [cR:rel->Prop] = [x,y:U]Ex [R:rel]and (cR R) (R x y):rel]
```

Now the Unionr of a family of relations is the a least upper bound for the family, i.e. we can prove

```
(* The theorem 'Unionr_ub' *)
Goal {cr|rel->Prop}{R|rel}(cr R)->(inclr R (Unionr cr));

Goal {cR|rel->Prop}{UB|rel}
       ({R:rel}(cR R)->(inclr R UB))
        ->
       (inclr (Unionr cR) UB)
```

If we assume a mapping F upon relations, can define the notion of stable and costable relations: note that a relation is fixed point of F if and only if it is both stable and costable.

```
[F:rel->rel]
[costable [R:rel] = inclr R (F R)]
[stable [R:rel] = inclr (F R) R]
```

Then the maximal fixed point may be defined as the Unionr of the costable relations, and if we assume that F is monotonic with respect to inclusion of relations we may show that it is a fixed point.

```
[max_fixpoint = Unionr costable]
[order_preserving:{R,S|rel}(inclr R S)->(inclr (F R) (F S))];
```

The proof is quite simple. Firstly we show

```
Goal costable max_fixpoint
```

Which will be an instance of the least upper bound property of Unionr, as from the properties assumed of F it is easy to show that F max_fixpoint is an upper bound.

The other inclusion needed for the fixed point property

```
Goal stable max_fixpoint
```

now follows easily from the upper bound property of Unionr, since by means of the lemma below, we may show that F max_fixpoint is also a costable relation.

```
Goal {R|rel}(costable R)->(costable (F R))
```

The principle of coinduction is essentially the upper bound property of maximal fixed point for costable relations. i.e.

```
[coinduction = Unionr_ub |costable]
```

5.2 Graphs

As we noted previously, a graph may be formalised by the binary edge relation on a universe U, which may be thought of as providing names for the vertices. An element of U represents a vertex if it is an endpoint of an edge. Note with this formulation graphs are directed, so a vertex is either an incoming vertex or an outgoing vertex. Thus we have the following definitions

```
[Graph = rel]
[Vertex [G:Graph] = [x:U]Ex [y:U]or (G x y) (G y x)]
[OutVertex [G:Graph] = [x:U]Ex [y:U]G x y]
[InVertex [G:Graph] = [x:U]Ex [y:U]G y x]
```

Our result concerns cyclic graphs. To define the notion of cyclicity we must define the transitive closure TClos of a relation, which determines the directed paths in a graph. Then a graph is cyclic, every vertex lies in a cycle, so whenever there is a path from x to y there is a return path from y to x.i.e:

```
[Cyclic [G:Graph] = {x,y:U}(TClos G x y)->(TClos G y x)]
```

In this example the definition of transitive closure is simpler than that needed for Warshall's lemma, the set of connecting vertices is no longer needed. However we still have essentially the same minimal fixed point definition.

```
[TC_p  [R:rel] [P:rel]
  = and ({x,y|U}(R x y)->(P x y))
        {x,y,z|U}(R x y)->(P y z)->(P x z)]

[TClos [R:rel][x,y:U]
  = {P:rel}(TC_p R P)->(P x y)]
```

By induction we prove that the transitive closure has the expected properties, for instance it is a transitive relation. In particular, for the bisimulation proof the following property is required.

```
[R:rel];
Goal {x,y|U}(TClos R x y)->Ex [y':U](R x y')
```

For graphs, this means any path from x to y has an initial edge. This obvious fact implies an essential property of cyclic graphs: in a cyclic graph every vertex is an out-vertex, so there are no dead ends.

```
Goal {G|Graph}(Cyclic G)
     ->{x|U}(Vertex G x)->(OutVertex G x)
```

5.3 Bisimulation

The concept of bisimulation comes from the theory of transition systems. It formalises the idea of the observational equivalence of processes, whereby two processes are indistinguishable if for any sequence of observations of the one a parallel sequence of observations may be elicited from the other. Transition systems may be modelled by labelled graphs, and directed graphs are the degenerate case of transition systems which have just a single label. So a pair of vertices from two graphs bisimulate, if for each path originating at one vertex a corresponding path originates from the other vertex. Any pair of vertices from cyclic graphs will bisimulate, as all paths are segments of cycles and may be extended indefinitely.

The formal expression of bisimulation is as the maximal fixed point of the following monotonic operation F on relations. Essentially F makes a single edge comparison between the graphs G1 and G2. Of course we must prove that F is monotonic (the theorem order_preserving_F).

```
[G1,G2:Graph]

[F [equiv:U->U->Prop]
   = [x,y:U]
      and ({x':U}(G1 x x')->Ex [y':U]and (G2 y y') (equiv x' y'))
          ({y':U}(G2 y y')->Ex [x':U]and (G1 x x') (equiv x' y'))]

[bisimulates  = max_fixpoint F]
```

Our general theory will now ensure that bisimulation is a fixed point i.e.

```
[costable_bis = costable_max_fix F (order_preserving_F)]
[stable_bis = stable_max_fix F (order_preserving_F)]
```

And give us the principle coinduction for this maximal fixed point.

We must also define the relation, which relates arbitrary vertices from the graphs G1 and G2.

```
[weakest_rel [x1,x2:U] = and (Vertex G1 x1) (Vertex G2 x2)]
```

The bisimulation between cyclic graphs is reflected in the logical equivalence of the bisimulation relation to this weakest relation. i.e.

```
Goal {x,y:U}(Cyclic G1)->(Cyclic G2)->(Vertex G1 x)
        ->
       (iff (max_fixpoint F x y) (weakest_rel x y));
```

The proof is straightforward, relying on several applications of the property of cyclic graphs noted earlier. To prove

```
(max_fixpoint F x y)-> (weakest_rel x y)
```

we note that the costability of the maximal fixed point forces the existence of an edge of G2 originating at y. The converse implication

```
(weakest_rel x y)-> (max_fixpoint F x y)
```

requires the use of coinduction: we show that `weakest_rel` is costable under F. This follows easily from the key property of cyclic graphs, as informally we must show that from both vertex x and vertex y we can proceed along an outgoing path.

References

[1] Coquand, T., Huet, G. The Calculus of Constructions, Information and Computation, vol 76 1988.

[2] Coquand, T., Huet, G. Constructions: A Higher Order Proof System for Mechanising Mathematics, Proc. EUROCAL 85, LNCS 203, Springer 1985.

[3] Coquand, T., Une Theorie des Constructions, PhD thesis, University of Paris 1985.

[4] Coquand, T., An Analysis of Girard's Paradox, 1st Symposium on Logic in Computer Science, IEEE 1986.

[5] Harper, R., Pollack, R., Type Checking, Universe Polymorphism, and Typical Ambiguity, Proc. TAPSOFT'89, LNCS 352, Springer 1989

[6] Huet, G. Induction Principles Formalised in the Calculus of Constructions, TAPSOFT'87, LNCS 249.

[7] Luo, Z. ECC, An Extended Calculus of Constructions, 4th Symposium on Logic in Computer Science, IEEE 1989.

[8] Luo, Z. C_{ξ}^{∞} and Its Meta Theory LFCS Report 58, Department of Computer Science, Edinburgh University 1988

[9] Luo, Z., Pollack, R., Taylor, P. How to Use LEGO: a preliminary user's manual. Preprint, LFCS, Department of Computer Science, Edinburgh University 1989

[10] Paulin-Mohring, C., A Development of Warshall's Algorithm, in Calcul des Constructions: Quelques examples de developpement de preuves, Preprint INRIA 1989

[11] Paulin-Mohring, C., Extracting Programs from Proofs in the Calculus of Constructions, Proc. 16th ACM Symp. Princ. Prog. Lang., ACM 1989.

[12] Pollack, R., The Theory of LEGO, Preprint LFCS, 1988.

[13] Tarski, A., A Lattice Theoretical Fixpoint Theorem and its Applications, Pacific J. Math. 5 (1955), 285-309.

Copyright © 1989, Laboratory for Foundations of Computer Science, University of Edinburgh. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.