Technische Universität Dresden

Fakultät Informatik

Diplomarbeit

zum Thema

Mechanizing Coinduction with Maude

Abstract

While induction and coinduction both are higher order principles, their underpinnings, initiality of algebras and finality of coalgebras, can be expressed in membership equational logic. They become conditional equations of morphisms in a category with the appropriate structure. By that "element free" approach, inductive and coinductive properties can be proved within membership equational logic. Maude is a tool that mechanizes reasoning in membership equational theories by rewriting. In my thesis I start to investigate this equational approach to coinductive theorem proving, and, in particular, how it can be supported by Maude.

von

Kai Brünnler

geboren am 28. Mai 1975 in Karl-Marx-Stadt (jetzt Chemnitz)

Verantwortlicher Hochschullehrer: Prof. Dr. rer. nat. habil. Horst Reichel

Contents

1 Introduction

Data types, like natural numbers, finite lists etc., are initial algebras. As such, they allow to define functions on them by induction and to prove properties of those functions by induction. A typical example are finite lists (or words) over some set A denoted A^*. The constructors of A^* usually are denoted $nil\colon \to A^*$ and $cons\colon A \times A^* \to A^*$. A function from the set of finite lists over A into some set can be defined by giving its values on the constructors. Consider for example the length of a list $len\colon A^* \to \mathbb{N}$, inductively defined as

$$len(nil) = 0 \qquad \text{and} \qquad len(cons(a, l)) = len(l) + 1.$$

In addition to the definition principle there also is a proof principle of induction. Proving that some predicate P holds for all $l \in A^*$ can be done by proving for all $l \in A^*$ and $a \in A$

$$P(nil) \qquad \text{and} \qquad P(l) \implies P(cons(a, l)).$$

Behaviours of systems that involve states and transitions between them (automata, transition systems, processes: what could be called process types) are final coalgebras. As such, dually to initial algebras, they provide the definition and proof principle of coinduction. A typical example are outputs of nonterminating processes, infinite lists (streams) over some set A denoted $A^{\mathbb{N}}$. Instead of constructors we have what could be called destructors: $head\colon A^{\mathbb{N}} \to A$ and $tail\colon A^{\mathbb{N}} \to A^{\mathbb{N}}$. Coinduction as a definition principle allows to define functions into $A^{\mathbb{N}}$ by giving the value of the destructors on it. Consider the definition of a function $even\colon A^{\mathbb{N}} \to A^{\mathbb{N}}$ that returns the stream containing every other element from the original stream:

$$head(even(s)) = head(tail(s)) \qquad \text{and} \qquad tail(even(s)) = even(tail(tail(s))).$$

That *even* is well-defined follows from $A^{\mathbb{N}}$ with *head* and *tail* being a final coalgebra, just as the well-definedness of *len* follows from A^* with *nil* and *cons* being an initial algebra. The proof principle of coinduction says that, to prove the equality of two streams s_1 and s_2, it suffices to exhibit some bisimulation relation R, such that $R(s_1, s_2)$. Usually the definitions of s_1 and s_2 determine R. A proof by coinduction then amounts to showing that R is a bisimulation, that is, for any streams s_1, s_2 if $R(s_1, s_2)$ then

$$head(s_1) = head(s_2) \qquad \text{and} \qquad R(tail(s_1), tail(s_2))$$

The proof principle of coinduction follows from the finality of $A^{\mathbb{N}}$ with *head* and *tail*, just as the proof principle of induction follows from the fact that A^* with *nil* and *cons* is an initial algebra. Proofs of that, the examples from above treated in more depth and many more are found in [JR97].

Considerable work has been done in the area of inductive theorem proving, either by using the induction principle as an axiom in a higher order setting or by making induction an explicit inference rule in a first order setting. Coinductive theorem proving has been done usually by describing final coalgebras as greatest fixpoints of endofunctors in higher order logic, see [P97] and [HJ97].

If one is interested in using both induction and coinduction in the same proof, a natural question to ask is whether the proof then should be carried out at a level where the duality between both principles is readily apparent. That level is the equality of morphisms in a category with structure, namely initial algebra objects and final coalgebra objects, together with their respective unique homomorphisms. By moving to that "element free" abstract level it is possible to greatly simplify the logic used to express the induction and coinduction principles, because here functions are first order elements. Proofs by induction or coinduction become purely algebraic.

Membership equational logic appears to be well-suited for reasoning in those categories because partiality can be easily encoded. The Maude system [Maude99] mechanizes reasoning in membership equational logic theories by rewriting. In this thesis I start to investigate this approach to coinductive theorem proving and in particular how the Maude system can help.

This work considers coinductive proofs only. But it should be stressed that the general approach dualizes straightforwardly to inductive proofs, and that the ultimate goal is to mechanize proofs that contain induction and coinduction nested one into another.

In the next section some coalgebraic peliminaries are defined and a short introduction to membership equational logic and Maude is given. In section 3 categories are specified in membership equational logic. They are enriched by products, sums and a terminal object in section 4. Final coalgebra objects are considered in section 5, the examples treated are streams, (completed) natural numbers and sequences. Maude will be used to prove some simple coinductive properties of those. In section 6 I briefly outline what a coinduction strategy is in the context of reasoning in a structured category and present some examples.

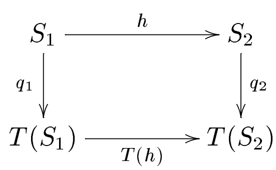
2 Preliminaries

I assume familiarity with the notions of *set*, *function*, *category*, *functor*, *product*, *coproduct* and the *free category generated by a graph.* A good reference for all of the above is [BW99]. The concept of *term rewriting* [DJ90], and in particular *termination* and *confluence* of a term rewriting system will be used to decide a *word problem.* In the next subsection, membership equational logic is introduced as in [M98], for that some basic notions from many-sorted universal algebra are needed.

2.1 Coinductive Reasoning

Definition 2.1 (Coalgebra). For an endofunctor $T\colon \mathbf{Set} \to \mathbf{Set}$, a coalgebra of T is a pair (S, q) consisting of a set S and a function $q\colon S \to T(S)$ called structure of the coalgebra.

Definition 2.2 (Coalgebra Homomorphism). For an endofunctor $T\colon \mathbf{Set} \to \mathbf{Set}$, a coalgebra homomorphism between two coalgebras (S_1, q_1) and (S_2, q_2) of T is a function $h\colon S_1 \to S_2$ such that the following diagram commutes:



Definition 2.3 (Final Coalgebra). For an endofunctor $T\colon \mathbf{Set} \to \mathbf{Set}$, a final coalgebra of T is a coalgebra (F, next) with $\mathit{next}\colon F \to T(F)$ such that for every coalgebra (S, q) of T there is exactly one coalgebra homomorphism $h\colon S \to F$.

Given the finality of some coalgebra (F, next) of T we have at the same time a

definition principle: any function $q\colon S \to T(S)$ uniquely determines a function $h\colon S \to F$, and a

proof principle: Given two functions $h_1, h_2\colon S \to F$ we can prove them equal by exhibiting some coalgebra structure $q\colon S \to T(S)$ that turns both h_1 and h_2 into coalgebra homomorphisms.

From now on, a coinductive definition of a function $h\colon S \to F$ is not a a couple of equations as in the definition of *even* in the introduction. Instead, it is a function $q\colon S \to T(S)$. Similarly, proof by coinduction will not mean a proof using the notion of bisimulation as seen in the introduction but using the proof principle just given.

2.2 Membership Equational Logic and Maude

Membership Equational Logic (abbreviated MEL) [M98] was conceived in the effort to find “the right” formalism for equational specification. As such, it should strike a good balance between expressiveness and simplicity. MEL does so by conservatively extending many- and order sorted equational horn clauses by so called membership axioms. This allows to express partiality while maintaining good properties of total equational logics, e.g. an initial algebra semantics.

Basic Definitions (from [M98])

Definition 2.4 (Signature). A signature Ω in membership equational logic is a triple (K, Σ, π) where K is a set whose elements are called kinds, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$ is a $K^* \times K$-indexed family of function symbols, and π is a function $\pi\colon S \to K$ that assigns to each element of a set S of sorts its corresponding kind. We denote by S_k the set $\pi^{-1}(k)$ for $k \in K$.

Intuitively, the above signatures can be seen as an extension of the usual many sorted signatures (whose sorts are now called kinds) by unary predicates (here called sorts).

Definition 2.5 (Ω-algebra, Ω-homomorphism). Given a signature Ω, an Ω-algebra consists of

1. a many-kinded Σ-algebra A, together with
2. an assignment to each sort $s \in S$ of a subset $A_s \subseteq A_{\pi(s)}$.

Given two Ω-algebras A and B, an Ω-homomorphism $h\colon A \to B$ is a Σ-homomorphism $h\colon A \to B$ such that for each sort $s \in S$ we have $h_{\pi(s)}(A_s) \subseteq B_s$.

Definition 2.6 (Sentences). There are two types of atomic formulas, namely:

1. equations of the form $t = t'$ where $t, t' \in T_\Sigma(X)_k$ for some $k \in K$, for X a K-kinded set of variables and $T_\Sigma(X)$ the free Σ-kinded algebra on X, and
2. membership assertions of the form $t : s$, where $s \in S$ and $t \in T_\Sigma(X)_{\pi(s)}$.

Sentences are universally quantified Horn clauses on these atomic formulas, that is, sentences of the form

$$(i) \quad (\forall Y) \quad t = t' \quad \Leftarrow \quad u_1 = v_1 \wedge ... \wedge u_n = v_n \wedge w_1 : s_1 \wedge ... \wedge w_m : s_m$$
$$(ii) \quad (\forall Y) \quad t : s \quad \Leftarrow \quad u_1 = v_1 \wedge ... \wedge u_n = v_n \wedge w_1 : s_1 \wedge ... \wedge w_m : s_m$$

where Y is a K-kinded set containing all the variables appearing in t, t' (resp. t) and the u_i, v_i, and w_j.

Definition 2.7 (Satisfaction). Given an Ω-algebra A and a K-kinded set of variables X such that $t, t' \in T_\Sigma(X)$ (resp. $t \in T_\Sigma(X)$), then satisfaction of an atomic formula $t = t'$ (resp. $t : s$ relative to a K-kinded function $a\colon X \to A$, called an assignment of values in A to the variables X, is defined in the obvious way, that is,

$$A, a \models_\Omega t = t' \text{ iff } \overline{a}(t) = \overline{a}(t')$$
$$A, a \models_\Omega t : s \text{ iff } \overline{a}(t) \in A_s$$

where $\overline{a}\colon T_\Sigma(X) \to A$ is the unique K-kinded Σ-homomorphism from $T_\Sigma(X)$ to A as a Σ-algebra extending the assignment a. Similarly, for φ a sentence of type (i) (resp. (ii)), we say that A satisfies φ, written $A \models_\Omega \varphi$, iff for all K-kinded assignments $a\colon X \to A$ such that $A, a \models_\Omega u_i = v_i, 1 \le i \le n$, and $A, a \models_\Omega w_j : s_j, 1 \le j \le m$, we have $A, a \models_\Omega t = t'$ (resp. $A, a \models_\Omega t : s$). For Γ a set of sentences we write $A \models_\Omega \Gamma$ iff for all $\varphi \in \Gamma, A \models_\Omega \varphi$. A theory is a pair $\Omega, \Gamma)$, with Γ a set of Horn Ω-sentences. Such a theory defines a subclass of Ω-algebras, that is, those A such that $A \models_\Omega \Gamma$.

Execution in Maude Maude's functional modules are membership equational logic theories with an intended initial semantics. Existence of an initial Ω-algebra in a membership equationally defined class of Ω-algebras is proved in [M98]. Intuitively, the underlying Σ-algebra of the initial Ω-algebra satisfies only those ground equations that hold in all models. In addition to that, the initial Ω-algebra only satisfies those ground memberships that hold in all models.

Consider the following specification of lists in Maude. The keyword `fmod` declares that the MEL theory contained has an initial semantics. The usual constructors `nil` and `cons` are declared as operators as well as some functions `len` and `getElAt`. Keywords `eq, mb, ceq, cmb` declare equations, memberships, conditional equations and conditional memberships, respectively. Sort `MachineInt` together with functions `_<_`, `_+_` and `_<=_` are imported from module `MACHINE-INT`. Note that `getElAt` is actually a partial operation, defined only for pairs of an integer and a list, where the length of the list is at least as long as the integer.

```
fmod LIST is

  including MACHINE-INT .

  sorts El El? List .
  subsorts El < El? .

  op  cons  : El List -> List .
  op  nil   : -> List .
  ops a b c : -> El .

  op  len     : List -> MachineInt .
  op  getElAt : MachineInt List -> El? .

  var L : List .
  var E : El .
  var I : MachineInt .

  eq   len(cons(E,L))  =  len(L) + 1 .
  eq   len(nil)        =  0 .


  ceq  getElAt(I,cons(E,L)) = getElAt(I - 1, L)  if  I > 1 .
  eq   getElAt(1,cons(E,L)) = E .

  cmb  getElAt(I,L) : El  if  I > 0 and I <= len(L) .

endfm
```

Kinds are now given implicitly by the set of maximal sorts in each connected component of the poset formed by the sorts with the subsort relation. That is, in our example there are two connected components: {`El, El?`} and {`List`}. Their maximal elements are `El?` and `List`, and the corresponding kinds are denoted `Error(El?)` and `Error(List)`, respectively. Those names derive from the intuition that terms that cannot be assigned a sort are error expressions, i.e. contain functions called with arguments for which they are not defined.

The `subsorts` declaration is actually syntactic sugar for

```
  cmb X : El? if X : El .
```

just as

```
  var E : El .
  eq   len(cons(E,L))  =  len(L) + 1 .
```

is syntactic sugar for

```
  ceq   len(cons(X,L))  =  len(L) + 1  if  X : El .
```

where X is a variable of kind `Error(El?)`.

The above specification can be executed in Maude by rewriting. The equations are thus seen as rewrite rules oriented from left to right. During the rewriting process, not only the rewrite rules, but also the membership axioms are applied to lower the sort of an expression. When this process terminates, i.e. neither rewrite rules nor membership axioms can be applied, the simplified expression together with its sort is returned:

```
Maude> red getElAt(2,cons(a,cons(b,cons(c,nil)))) .
rewrites: 5
result El: b
Maude> red getElAt(4,cons(a,cons(b,cons(c,nil)))) .
rewrites: 13
result El?: getElAt(1, nil)
Maude> red cons(getElAt(1,nil),nil) .
rewrites: 4
result Error(List): cons(getElAt(1, nil), nil)
```

Module `LIST` demonstrates the key advantage of membership equational logic over just conditional equational logic: we easily can express partiality. While partial operations could be encoded in conditional equational logic by introducing an error expression such as

```
 op   error : -> El .
 ceq  getElAt(I,L)  =  error  if  I > len(L)  or  I < 1 .
```

the membership equational approach frees us from having to think about what functions with domain `El` should do with `error` and is of course less questionable regarding the semantics of `El`. While this could be achieved with in an order-sorted approach as well (by choosing `El?` as the domain of `error`, there still is a problem: now all undefined terms are equal and reduced to `El? : error`. Of course `El?: getElAt(1, nil)` is a much more informative error message.

3 Categories

The idea in this work is to use not proof of equality by bisimulation to show theorems in final coalgebras but instead directly to use their uniqueness property. Neither of those two proof principles is a first order statement. While the first quantifies over a relation (the bisimulation), the second quantifies over functions. How then, can I express them in a weak logic as MEL and mechanize them with Maude? The short answer is: to reason in equalities of morphisms in a category with structure. The long answer is the remainder of this thesis. To this end, categories are specified in MEL, see module `CATEGORY` page 6.

A (small) category can be seen as an algebra of functions. Functions are the elements of its carrier set and the operations of the algebra work on those. As such, a category is a generalization of a monoid: composition of functions is associative, there is an identity element, but composition now in general is a *partial* operation, defined only for composable functions, i.e. two functions where the codomain of the first coincides with the domain of the second. While total algebras can be easily specified in equational horn logic, specification of categories requires the additional strength of MEL to express partiality. We introduce a sort `Arrow?` on which the (total) composition operator is defined and a subsort `Arrow`. A conditional membership asserts that terms with composition as their main constructor, i.e. of sort `Arrow?` are of sort `Arrow` if codomain and domain of its respective arguments are the same.

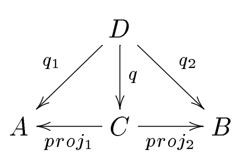
4 Product and Sum

This section introduces some structure that is used to build endofunctors for the final coalgebras considered in section 5. This structure consists of products, coproducts (sums) and a terminal object. First, I specify a cartesian category and present a confluent and terminating rewrite system that decides equalities of well-formed morphisms in the free cartesian category generated by a graph. It forms the basis for proving coinductive properties by rewriting. Secondly, I show how partiality of the pairing constructor is expressed in MEL and how Maude can thus check the well-formedness of morphisms. Then I introduce what will be called a "strictly associative product", a product that allows us not to worry about associativity isomorphisms when dealing with nested products. Finally, to enable case analysis, distributivity of the product over the sum is specified and booleans are exhibited as sum of the terminal object with itself.

```
fmod CATEGORY is

  sorts   Object Arrow Arrow? .

  subsorts   Arrow < Arrow? .

  op id           :  Object -> Arrow .
  ops dom cod     :  Arrow -> Object  .
  op _;_          :  Arrow? Arrow? -> Arrow? [ assoc ] .

  vars F G        :  Arrow .
  vars A          :  Object .

  cmb   F ; G     : Arrow       if      cod(F) == dom(G) .

  ceq   id(A) ; F = F           if      dom(F) == A .
  ceq   F ; id(A) = F           if      cod(F) == A .

  eq    dom( F ; G ) = dom(F) .
  eq    cod( F ; G ) = cod(G) .

  eq    dom(id(A)) = A .
  eq    cod(id(A)) = A .

endfm
```

Module 1: A Category

Definition 4.1 (Product). Let A and B be two objects in a category. A (*not* the) product of A and B is an object C together with arrows $proj_1 \colon C \to A$ and $proj_2 \colon C \to B$ such that for any object D and arrows $q_1 \colon D \to A$ and $q_2 \colon D \to B$, there is a unique arrow $q \colon D \to C$ that makes the following diagram commute:



A category in which for any two objects A and B there is a product of A and B is said to have binary products.

Definition 4.2 (Terminal Object). An object A is called terminal if for every object B (that may be equal to A) there is exactly one arrow from B to A.

Definition 4.1 gives a binary product. It can be generalized to an n-ary product in the obvious way. A category having n-ary products for any $n \in \mathbb{N}$ is said to have *finite* products. Since n-ary products for $n > 2$ can be built using binary products, a category that has binary products and a terminal object has finite products. A category is called *cartesian* if it has finite products.

4.1 Specifying a Cartesian Category

The definition of a product involves two aspects: existence of such an arrow $q : D \longrightarrow C$ and uniqueness thereof. We specify its existence by simply giving it a name, i.e. introducing the binary constructor

```
op <_,_>               : Arrow Arrow -> Arrow .
```

called *pairing* which, given two arrows with the same domain, yields the arrow whose existence is required by the universal mapping property. The same is done to the product object: given two objects, applying constructors

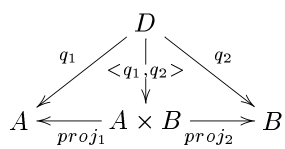
```
op _x_                 : Object Object -> Object .
```

and

```
ops proj1 proj2        : Object Object -> Arrow .
```

yield an object that will satisfy the definition of a product and its projections, respectively.

The diagram in Definition 4.1 can thus be instantiated as:



and will in the following permit to talk about *the* product.

Commutativity of the diagram is easily expressed by equations of morphisms:

```
var A B : Object .
var F G : Arrow .

eq < F,G > ; proj1(A,B) = F .                                   (ExP1-2)

eq < F,G > ; proj2(A,B) = F .
```

Uniqueness can be expressed by a conditional equation:

```
ceq < F,G > = H  if   H ; proj1(cod(F), cod(G)) = F and
                      H ; proj2(cod(F), cod(G)) = G .           (UniP')
```

Now the product of two arrows can be specified by simply identifying it with its equivalent representation as a pairing:

```
op _x_                 : Arrow Arrow -> Arrow .

eq  F x G = < proj1( dom(F), dom(G) ) ; F ,
              proj2( dom(F), dom(G) ) ; G > .
```

Defining domain and codomain of the introduced morphisms is straightforward:

```
eq  dom( proj1(A,B) ) = A x B .
eq  dom( proj2(A,B) ) = A x B .
eq  cod( proj1(A,B) ) = A .
eq  cod( proj2(A,B) ) = B .

eq  dom( < F,G > ) = dom(F) .
eq  cod( < F,G > ) = cod(F) x cod(G) .
```

Now we just have a category with binary products. For a cartesian category we have to add a terminal object (the empty product) with its unique morphisms (called bang):

```
op 1  : -> Object .
op !  : Object -> Arrow .

eq  dom(!(A)) = A .
eq  cod(!(A)) = 1 .
```

Expressing the universal mapping property of the terminal object in MEL is easy:

```
ceq G = !(dom(G)) if cod(G) == 1 .
```
(Trm)

All that is said above, dualizes to the cocartesian category. Thus it is possible to specify a cocartesian category in MEL as well. But since the initial object and its unique maps seem rather useless in building endofunctors for my purpose, I will contend myself with binary coproducts. The signature looks as follows:

```
op _+_              : Object Object -> Object .
ops inc1 inc2       : Object Object -> Arrow .
op [_,_]            : Arrow Arrow -> Arrow .
```

The axioms are straightforward duals to the ones for the product. See modules PRODUCT and SUM pages 36 and 39 for the full specification.

Notation Equations will be referred to by names put in parentheses such as (ExP1) which denotes the first equation specifying the existence of the pairing. The dual equation for the copairing will be denoted (ExS1).

4.2 Deciding the Equality of Morphisms

Every functional Module in Maude has an operational semantics as a rewrite system. Under the usual confluence and termination assumptions, they may constitute decision procedures for the word problem in the respective MEL theory. Since I specify categories with additional structure, the word problems correspond to the question whether two morphisms in the free category with that structure are equal.

Module CATEGORY (page 6) constitutes a decision procedure for the word problem in the free category generated by a graph. The reasoning is as follows:

1. well-formedness of morphisms is decidable: just check for every composition operator that it is defined on its arguments. A decision procedure for the word problem may thus rely on well-formedness.
2. Knowing that only well-formed arrows are given, and that all the equations in module CATEGORY preserve well-formedness, the conditions of the cmb and the two ceqs will always be fulfilled. We can thus replace Arrow? by Arrow, drop the cmb and the conditions of the ceqs, turning them into plain equations.
3. On well-formed arrows, the resulting reduction system (that consists just of the rules for identity and associativity) is confluent and terminating.

Because in my restricted framework well-formedness of morphisms is decidable, it turns out that memberships are not necessary to give decision procedures. The decidability proof for the free cartesian category generated by a graph can thus be based on a conventional term rewriting system without memberships or conditions. The reason why in practice I use memberships as well as equations to prove equality of morphisms is that now Maude is checking the well-formedness of my input – and much better than myself.

Deciding the word problem in cartesian categories The problem with conditional equation (UniP') (page 7) is that it cannot be oriented into a conditional rewrite rule, since, either way, the right-hand-side contains variables that do not occur in the left-hand-side. Thus, it cannot be executed with Maude.

Fortunately, there is an equivalent equation which can be oriented.

```
eq < F ; proj1(A,B), F ; proj2(A,B) > = F .
```
(UniP)

Lemma 4.3. Assuming the correct definitions of *dom* and *cod*, (UniP) is equivalent to (UniP').

Proof.

"⇐": Instantiate (UniP') with `F = F' ; proj1(A,B), G = F' ; proj2(A,B)` and `H = F'`, the conditions can be proven using the definitions of dom and cod, now we have (a variant of) (UniP).

"⇒": To prove (UniP'), apply the equations in its condition to its conclusion, this yields an instance of (UniP).

□

The resulting term rewriting system is not confluent yet, completion yields 2 more rules:

```
eq F ; < G,H > = < F ; G, F ; H > .
```
(CompP)

and

```
eq < proj1(A,B), proj2(A,B) > = id(A x B) .
```
(idP)

With (Trm) (page 8) again the problem is that the conditional equation cannot be executed as a conditional rewrite rule because as such it is obviously not terminating. And again, there is a solution to this problem:

Lemma 4.4. Assuming the correct definitions of *dom* and *cod* and that no arrows with codomain 1 are introduced in addition to the ones that can be built from the signature so far (identities, compositions, pairings, projections, bang), the following equations are equivalent to (Trm):

$$proj_1(1, A) = !_{1\times A} \quad (!1)$$

$$proj_2(A, 1) = !_{A\times 1} \quad (!2)$$

$$!_1 = id_1 \quad (!3)$$

$$F \; ; \; !_A = !_{dom(F)} \quad (!4)$$

Proof.

"⇒": Trivial .

"⇐": Let G be a term of sort `Arrow` with $cod(G) = 1$ and prove $G = !_{dom(G)}$ by induction on G: the induction base is given by equations (!1),(!2) and (!3). For the induction step pairings `<_,_>` need not be considered because they never have codomain 1. A composition $G = G_1 \; ; \; G_2$ can have codomain 1, then G_2 has codomain 1, apply the induction hypothesis, now $G_2 =!_{dom(G_2)}$ and by (!4) $G = !_{dom(G)}$.

□

Assuming that there are no arrows with codomain 1 except the ones mentioned in Lemma 4.4 is of course very reasonable because any additional arrow $f\colon A \to 1$ is equal to $!_A$ anyway.

The equations given in Lemma 4.4 can be oriented into rewrite rules and added to the ones for the binary product. Some critical pairs arise, they are oriented as well, giving rise to the rewrite system in module CARTESIAN (page 35).

Remark 4.5. The difference between module CARTESIAN and the union of modules CATEGORY, PRODUCT, and TERMINAL (pages 6, 36, 38, respectively) is that the first is a simple term rewriting system, while the second is a conditional class rewriting system of congruence classes of terms induced by associativity of the composition operator. The second also uses memberships and conditional rewrite rules to check the well-formedness of morphisms, while the first does not.

Theorem 4.6. Equality of morphisms in the free cartesian category generated by a graph is decidable.

Proof.

By Lemma 4.3 and Lemma 4.4 the equational theory in module CARTESIAN indeed specifies the free cartesian category generated by whatever morphisms one adds as constants.

Termination can be proven by a recursive path ordering with status (to deal with associativity) and the following ordering on function symbols:

```
dom = cod > _;_ > <_,_> > proj1 = proj2 > ! > id
```

For well-formed arrows, the system is locally confluent. That can be seen by checking that all critical pairs converge. Confluence (for well-formed arrows) follows from termination and local confluence, as usual. □

The same holds for the free cocartesian category. Interestingly, the associativity rule has to be reversed, i.e. shift parentheses to the right rather than to the left as in module CARTESIAN. To reason in the union of both theories, I resort to rewriting modulo associativity, as shown in module CATEGORY. Still: the resulting system is not confluent (nor confluent modulo associativity when the associativity rule is removed), the problem is a redex of the form

```
[ f,g ] ; < h,j >
```

which can be reduced using (CompP) as well as (CompS), yielding two terms that are not reducible to one another. Completion modulo associativity turned out to be next to impossible to do by hand since critical pairs are numerous and complex.

There is a very recent result by Cockett and Seely [CS00]. They use cut-elimination to prove the decidability of equality of morphisms in the free category with finite products and sums generated by a graph.

Partiality Of course, just as `_;_`, the pairing operator `<_,_>` is partial, defined only on arrows the domains of which coincide. In module CARTESIAN there is no distinction between defined (well-formed) and undefined arrows. Just as in most categorical literature it is assumed implicitly that equations only apply to well-formed arrows, i.e. whenever a user wants to check the equality of two morphisms with module CARTESIAN, she should first check whether both of them are well-formed. This is a reasonable assumption at least in our restricted framework, since this property is decidable. But on one hand this is an error-prone task nevertheless and on the other hand, because of the undecidability of the word problem in equational theories in general, one can imagine categories with equationally specified structure that makes the well-formedness of arrows undecidable. Specifying pairing as a partial function can be accomplished in MEL along the same lines as in module CATEGORY. First, pairing is introduced with codomain Arrow?

```
op <_,_>                : Arrow Arrow -> Arrow? .
```

then a conditional membership specifies which arrows are well-formed

```
cmb   < F,G > : Arrow               if  dom(F) == dom(G) .
```

and then all equations are turned into conditional equations to be applicable only in the case of well-formedness. For example (ExP1) becomes

```
ceq   < F,G > ; proj1(A,B) = F   if  dom(F) == dom(G) and
                                     cod(F) == A and
                                     cod(G) == B .
```

4.3 Strict Associativity

In a category with binary products there are two ways to build a ternary product: $A \times (B \times C)$ and $(A \times B) \times C$. Categorically, there is no reason to distinguish them since they are canonically isomorphic by

$$< \mathit{proj}_1(A \times B, C) \; ; \; \mathit{proj}_1(A, B), \\ < \mathit{proj}_1(A \times B, C) \; ; \; \mathit{proj}_2(A, B), \; \mathit{proj}_2(A \times B, C) >> \qquad (\mathit{assocLeft})$$

$$<< \mathit{proj}_1(A, B \times C), \; \mathit{proj}_2(A, B \times C) \; ; \; \mathit{proj}_1(B, C) >, \\ \mathit{proj}_2(A, B \times C) \; ; \; \mathit{proj}_2(B, C) > \qquad (\mathit{assocRight})$$

But when modelling free categories as initial models of some equational theory, two different but isomorphic objects are distinguished since they are two different terms. So if two functions f and g have to be composed, the codomain of the first being $A \times (B \times C)$ and the domain of the second being $(A \times B) \times C$, we have to explicitly put in the appropriate isomorphism, which does nothing else than shift parentheses and bloat definitions.

For that reason, defining morphisms by composition in the free category with binary products is a bit cumbersome. Instead of just writing

```
eq  h = f ; g .
```

we have to write

```
eq  h = f ; < < proj1(A, B x C) , proj2(A, B x C) ; proj1(B, C) > ,
              proj2(A, B x C) ; proj2(B, C)
            > ; g .
```

To avoid that, the reasoning is better done in a different but equivalent category, the free category with strictly associative products:

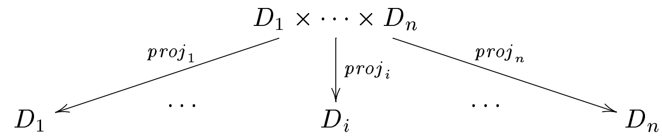
Definition 4.7 (Strictly Associative Product). A category is said to have strictly associative products iff it has products such that (1) any two objects $A \times (B \times C)$ and $(A \times B) \times C$ are equal and (2) the corresponding isomorphisms coincide with the identity on that object.

Interpretation in Set What I want to reason about are equations of functions, i.e. equations of morphisms in **Set**. What I am reasoning about is something else, namely ground equations of morphisms in a free category with structure. This is not a problem because such ground equations hold in all categories with that structure (not just the free one) and most of the structure I will consider (finite products, binary sums, distributivity, final coalgebras of some functors) is available in **Set**. Thus, ground equations of morphisms, that hold in the free category with that structure, hold in **Set** as well. I do not know, however, whether **Set** has binary products that are strictly associative. To show that all my reasoning in the free category with strictly associative products works in **Set** as well, it suffices to construct a product preserving functor from the free category with strictly associative products into **Set**.

The free category with strictly associative products generated by a base category **B** is denoted **Th**. The category in which we want to interpret associative products, called domain, will be denoted **D** and is required to have binary products. In particular, **Set** qualifies as domain. I now extend some functor $M_{\mathbf{B}}\colon \mathbf{B} \to \mathbf{D}$ to a functor $M\colon \mathbf{Th} \to \mathbf{D}$. To do so, I could map a nested binary product in **Th** onto a nested binary product in **D** by choosing a canonical nesting such as shifting parentheses to the right:

$$M\colon A_1 \times \cdots \times A_m \mapsto A_1 \times (A_2 \times \cdots \times (A_{m-1} \times A_m) \cdots)$$

Then M has to be defined inductively, and the proof that it is well-defined and product-preserving would also have to involve inductions. Since **D** has binary products, it also has finite n-ary products with $n \geq 1$. To dispense with induction in the following definition and proof, a nested strictly associative product of n objects from **B** in **Th** is interpreted as an n-ary product in **D**. A product of n objects $D_1 \dots D_n$ in **D** with its projections will be denoted



Definition 4.8. Let $A = A_1 \times \cdots \times A_m$, $B = B_1 \times \cdots \times B_n$ where $A_1 \dots A_m B_1 \dots B_n$ are objects in **B**, $m, n \geq 1$. Define $M\colon \mathbf{Th} \to \mathbf{D}$ as

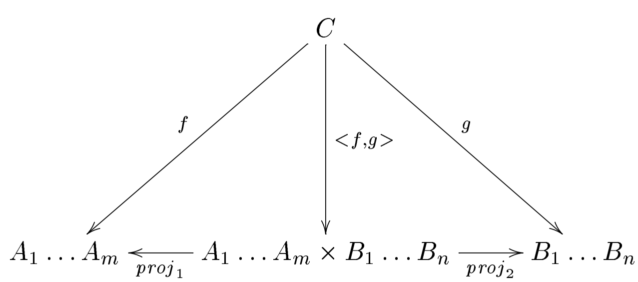
$$\begin{array}{lcl}
M(A \times B) & = & M(A_1) \times \cdots \times M(A_m) \times M(B_1) \times \cdots \times M_{\mathbf{B}}(B_n) \\
M(proj_1(A,B)) & = & < \; proj_1(M(A_1 \times \cdots \times A_m \times B_1 \times \cdots \times B_n)), \dots, \\
 & & \quad proj_m(M(A_1 \times \cdots \times A_m \times B_1 \times \cdots \times B_n)) > \\
M(proj_2(A,B)) & = & < \; proj_{m+1}(M(A_1 \times \cdots \times A_m \times B_1 \times \cdots \times B_n)), \dots, \\
 & & \quad proj_{m+n}(M(A_1 \times \cdots \times A_m \times B_1 \times \cdots \times B_n)) > \\
M(< f, g >) & = & < \; M(f) \; ; \; proj_1(A_1 \dots A_m), \dots, M(f) \; ; \; proj_m(A_1 \dots A_m), \\
 & & \quad M(g) \; ; \; proj_1(B_1 \dots B_n), \dots, M(g) \; ; \; proj_n(B_1 \dots B_n) > \\
 & & \text{where } f \text{ and } g \text{ are morphisms in } \mathbf{Th} \text{ with } dom(f) = dom(g) \\
M(f) & = & M_{\mathbf{B}}(f) \\
M(C) & = & M_{\mathbf{B}}(C) \\
 & & \text{where } f \text{ is a morphism and } C \text{ is an object in } \mathbf{B} \\
M(f \; ; \; g) & = & M(f) \; ; \; M(g) \\
M(id_C) & = & id_{M(C)} \\
 & & \text{where } f \text{ and } g \text{ are morphisms and } C \text{ is an object in } \mathbf{Th} \text{ with } cod(f) = dom(g)
\end{array}$$

Note that, by definition, $M((A \times B) \times C) = M(A \times (B \times C))$.

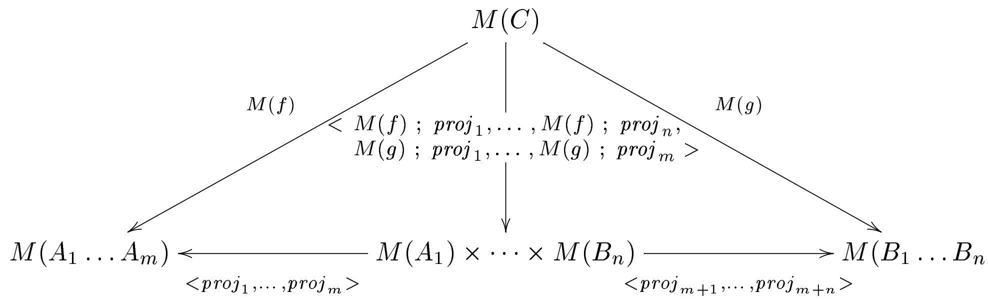
Proposition 1. M is a product-preserving functor.

Proof. To show that M is a functor, we just have to show that it is well-defined, i.e. that diagrams that are defined to commute in **Th** are taken to commuting diagrams in **D**.

Consider the following commuting diagram in **Th**



on which M yields the following diagram in $\mathbf{D}$



It is easy to see that the last diagram commutes. Further, it has to be shown in $\mathbf{D}$ that for any objects A, B, C in $\mathbf{Th}$

$$M(\mathit{assocLeft}) = M(\mathit{assocRight}) = id_{M(A)\times M(B)\times M(C)}$$

where $\mathit{assocLeft}\colon (A \times B) \times C \to A \times (B \times C)$ and $\mathit{assocRight}\colon A \times (B \times C) \to (A \times B) \times C$ are the canonical isomorphisms given on page 11. That can be done by applying the inductive definition of M as much as possible, and then simplifying the result using the identities of the ternary product in $\mathbf{D}$.

It remains to be shown that M preserves products. To that end, consider some $h\colon M(C) \to M(A_1) \times \cdots \times M(B_n)$ such that

$$\begin{aligned} h \;;\; <\mathit{proj}_1, \ldots, \mathit{proj}_n> &= M(f) \qquad \text{and} \\ h \;;\; <\mathit{proj}_{m+1}, \ldots, \mathit{proj}_{m+n}> &= M(g) \end{aligned}$$

Now $< M(f) \;;\; \mathit{proj}_1, \ldots, M(f) \;;\; \mathit{proj}_m, M(g) \;;\; \mathit{proj}_1, \ldots, M(g) \;;\; \mathit{proj}_n >$ can be shown equal to h by substituting $M(f)$ and $M(g)$ from the last two equations and simplifying the result using the identities of the three products involved: the m-ary, the n-ary and the $(m+n)$-ary. □

To specify strict associativity equationally and to execute it with Maude, module `PRODUCT` is changed by making associativity a structural property of constructor `_x_` on objects and adding equations that express property (2) of Definition 4.7.

$$\begin{aligned} \mathit{proj}_1(A \times B, C) \;;\; \mathit{proj}_1(A, B) &= \mathit{proj}_1(A, B \times C) \\ \mathit{proj}_2(A, B \times C) \;;\; \mathit{proj}_2(B, C) &= \mathit{proj}_2(A \times B, C) \\ < \mathit{proj}_1(A \times B, C) \;;\; \mathit{proj}_2(A, B) \;,\; \mathit{proj}_2(A \times B, C) > &= \mathit{proj}_2(A, B \times C) \qquad \text{(assocP1-4)} \\ < \mathit{proj}_1(A, B \times C) \;,\; \mathit{proj}_2(A, B \times C) \;;\; \mathit{proj}_1(B, C) > &= \mathit{proj}_1(A \times B, C) \end{aligned}$$

Proposition 2. In a category with binary products (assocP1-4) are equivalent to property (2) of Definition 4.7.

Proof.

"⇐":

$$
\begin{aligned}
& < proj_1(A \times B, C) \; ; \; proj_1(A, B), \\
& < proj_1(A \times B, C) \; ; \; proj_2(A, B) \; , \; proj_2(A \times B, C) >> \\
= \quad & < proj_1(A, B \times C) \; , \; < proj_1(A \times B, C) \; ; \; proj_2(A, B) \; , \; proj_2(A \times B, C) >> \\
& \hfill \text{(by assocP1)} \\
= \quad & < proj_1(A, B \times C) \; , \; proj_2(A, B \times C) > \\
& \hfill \text{(by assocP3)} \\
= \quad & id_{A \times B \times C} \\
& \hfill \text{(by idP)}
\end{aligned}
$$

The same can be shown for the inverse isomorphism by (assocP2, assocP4 and idP).

"⇒":

$$
\begin{aligned}
& proj_1(A \times B, C) \; ; \; proj_1(A, B) \\
= \quad & id_{A \times B \times C} \; ; \; proj_1(A \times B, C) \; ; \; proj_1(A, B) \\
& \hfill \text{(by left id)} \\
= \quad & assocRight \; ; \; proj_1(A \times B, C) \; ; \; proj_1(A, B) \\
& \hfill \text{(by Definition 4.7)} \\
= \quad & < proj_1(A, B \times C) \; , \; proj_2(A, B \times C) \; ; \; proj_1(B, C) > \; ; \; proj_1(A, B) \\
& \hfill \text{(by ExP1)} \\
= \quad & proj_1(A, B \times C) \\
& \hfill \text{(by ExP1)}
\end{aligned}
$$

(assocP2) can be shown by left id, Definition 4.7 choosing *assocRight*, and twice (ExP2).

$$
\begin{aligned}
& < proj_1(A \times B, C) \; ; \; proj_2(A, B) \; , \; proj_2(A \times B, C) > \\
= \quad & assocLeft \; ; \; proj_2(A, B \times C) \\
& \hfill \text{(by ExP2)} \\
= \quad & id_{A \times B \times C} \; ; \; proj_2(A, B \times C) \\
& \hfill \text{(by Definition 4.7)} \\
= \quad & proj_2(A, B \times C) \\
& \hfill \text{(by left id)}
\end{aligned}
$$

(assocP4) can be shown by (ExP1), Definition 4.7 choosing *assocRight*, and left id.

□

Proposition 3. In a category with strictly associative products the pairing operator is associative.

Proof.

Let f, g, h be arrows in a category with strictly associative products such that their domains coincide.

$$
\begin{aligned}
& < f, < g, h >> \\
= \quad & < f, < g, h >> \; ; \; \mathit{assocRight} \\
& \hfill \text{(by right id and Definition 4.7)} \\
= \quad & << f, < g, h >> \; ; \; < \mathit{proj}_1(A, B \times C) \; , \; \mathit{proj}_2(A, B \times C) \; ; \; \mathit{proj}_1(B, C) >, \\
& < f, < g, h >> \; ; \; \mathit{proj}_2(A, B \times C) \; ; \; \mathit{proj}_2(B, C) > \\
& \hfill \text{(by CompP)} \\
= \quad & <<< f, < g, h >> \; ; \; \mathit{proj}_1(A, B \times C), \\
& < f, < g, h >> \; ; \; \mathit{proj}_2(A, B \times C) \; ; \; \mathit{proj}_1(B, C) >, h > \\
& \hfill \text{(by CompP and twice ExP2)} \\
= \quad & << f, g >, h > \\
& \hfill \text{(by ExP1, ExP2 and ExP1)}
\end{aligned}
$$

□

Thus, associativity can be a structural axiom for the pairing constructor as in module PRODUCT-ASSOC (page 37). The more properties can be made structural axioms modulo which inferences take place, the easier the inferences.

4.4 Booleans and Distributivity

Given a terminal object, denoted 1, as given in module CARTESIAN, the simplest interesting data type, the booleans, can be exhibited as a coproduct. Module BOOLEAN (page 41) introduces some "macros". Constants f and t represent false and true, respectively.

```
op bool           : -> Object .
ops f t           : -> Arrow .
ops not and       : -> Arrow .


eq  bool  =  1 + 1 .
eq  f     =  inc1(1,1) .
eq  t     =  inc2(1,1) .
```

Example 4.9. The definition of negation as

```
eq  not   =  [ t,f ] .
```

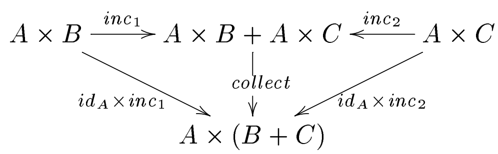
exemplifies the use of copairing to make a case analysis:

1. if the morphism that we compose to the left of `not` goes into the first component of the sum (false), yield true,
2. if it goes into the second component (true), yield false.

The simple statement $\forall A. \neg\neg A = A$ now becomes provable as `not ; not = id` simply by rewriting. Note that, since we are talking about equalities of morphisms (or functions), no variables and no universal quantification are required to express this statement.

Things get a bit more interesting when we define binary functions by case analysis. We cannot just use copairing since the domain of a copairing is a sum, while the domain of the function we want to define, is a product (of sums). It turns out that in a category with products and sums in general it is not possible to express a case analysis on products that considers both arguments. We need the following property:

Definition 4.10 (Distributive Category). A category is called distributive if it has binary sums and finite products such that for all objects A,B and C, the unique arrow *collect* defined by the following diagram is an isomorphism.



This differs from the definition given in [BW99] by not requiring an initial object. As mentioned before, I have no need for it.

The morphism *collect* required in Definition 4.10 already exists and is unique, it is

$$\begin{aligned}[\quad & < proj_1(A,B)\ ,\ proj_2(A,B)\ ;\ inc_1(B,C) >, \\ & < proj_1(A,C)\ ,\ proj_2(A,C)\ ;\ inc_2(B,C) > \quad]\end{aligned} \tag{collect}$$

It is turned into an isomorphism by introducing another arrow

```
op dist :  Object Object Object -> Arrow .
```

and adding the equations requiring that this is the inverse of *collect*:

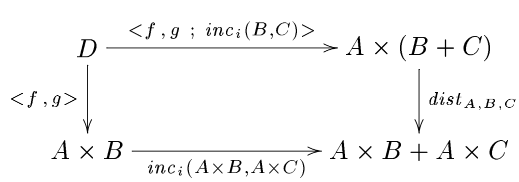
$$\begin{aligned} dist_{A,B,C}\ ;\ [\ & < proj_1(A,B)\ ,\ proj_2(A,B)\ ;\ inc_1(B,C) >, \\ & < proj_1(A,C)\ ,\ proj_2(A,C)\ ;\ inc_2(B,C) >] = id_{A\times(B+C)} \end{aligned} \tag{Dist1}$$

$$\begin{aligned} [\ & < proj_1(A,B)\ ,\ proj_2(A,B)\ ;\ inc_1(B,C) >, \\ & < proj_1(A,C)\ ,\ proj_2(A,C)\ ;\ inc_2(B,C) >]\ ;\ dist_{A,B,C} = id_{A\times B+A\times C} \end{aligned} \tag{Dist2}$$

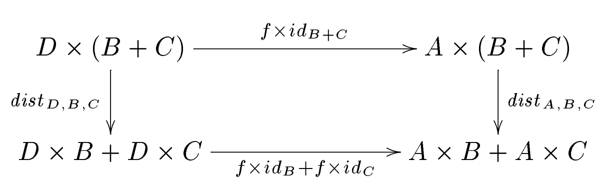
Remark 4.11. $dist_{A,B,C} : A \times (B+C) \to A \times B + A \times C$ provides left distributivity. Right distributivity follows by the commutativity isomorphism of the product and functoriality of the sum:

$$\begin{aligned} & < proj_2(B+C, A), proj_1(B+C, A) >\ ;\ dist_{A,B,C}\ ; \\ (& < proj_2(A,B)\ ,\ proj_1(A,C) > + < proj_2(A,B)\ ,\ proj_1(A,C) >) \end{aligned}$$

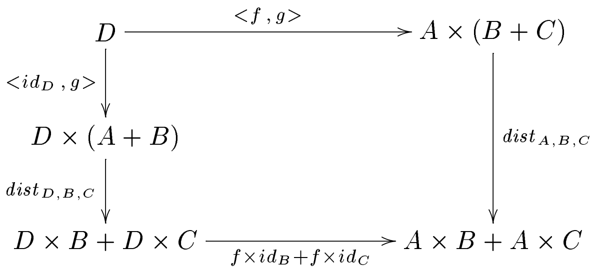
Lemma 4.12. Given a morphisms $f : D \to A$ in a distributive category, the following diagrams commute:



where $i \in \{1, 2\}$ and $g : D \to B$, $g : D \to C$, respectively (Dist3-4)



(Dist5)



where $g\colon D \to B + C$ (Dist6)

Proof.

$$
\begin{array}{ll}
 & < f \ , \ g \ ; \ inc_1(B,C) > \ ; \ dist_{A,B,C} \\
= & < f,g > \ ; \ < proj_1(A,B) \ , \ proj_2(A,B) \ ; \ inc_1(B,C) > \ ; \ dist_{A,B,C} \qquad \text{(by ExP1-2, CompP)} \\
= & < f,g > \ ; \ inc_1(A \times B, A \times C) \ ; \\
 & [< proj_1(A,B) \ , \ proj_2(A,B) \ ; \ inc_1(B,C) >, \\
 & \ < proj_1(A,C) \ , \ proj_2(A,C) \ ; \ inc_2(B,C) >] \ ; \ dist_{A,B,C} \qquad \text{(by ExS1)} \\
= & < f,g > \ ; \ inc_1(A \times B, A \times C) \qquad \text{(by Def. 4.10)}
\end{array}
$$

The proof of (Dist4) is analogous.

$$
\begin{array}{ll}
 & f \times id_{B+C} \ ; \ dist_{A,B,C} \\
= & dist_{D,B,C} \ ; \\
 & [< proj_1(D,B) \ , \ proj_2(D,B) \ ; \ inc_1(B,C) > , \\
 & \ < proj_1(D,C) \ , \ proj_2(D,C) \ ; \ inc_2(B,C) >] \ ; \\
 & f \times id_{B+C} \ ; \ dist_{A,B,C} \qquad \text{(by Def. 4.10)} \\
= & dist_{D,B,C}; \ [< proj_1(D,B) \ ; \ f \ , \ proj_2(D,B) \ ; \ inc_1(B,C) > , \\
 & \qquad < proj_1(D,C) \ ; \ f \ , \ proj_2(D,C) \ ; \ inc_2(B,C) >] \ ; \ dist_{A,B,C} \\
 & \\
= & dist_{D,B,C} \ ; \ [< proj_1(D,B) \ ; \ f \ , \ proj_2(D,B) > \ ; \ inc_1(A \times B, A \times C) \ , \\
 & \qquad < proj_1(D,C) \ ; \ f \ , \ proj_2(D,C) > \ ; \ inc_2(A \times B, A \times C) \ ] \qquad \text{(by Dist3 and Dist4)} \\
= & dist_{D,B,C} \ ; \ (f \times id_B + f \times id_C)
\end{array}
$$

$$
\begin{array}{ll}
 & < f,g > \ ; \ dist_{A,B,C} \\
= & < id_D, g > \ ; \ (f \times id_{B+C}) \ ; \ dist(A,B,C) \\
= & < id_D, g > \ ; \ dist_{D,B,C} \ ; \ (f \times id_B + f \times id_C) \\
 & \qquad \text{(by Dist5)}
\end{array}
$$

□

Those equations are useful in reasoning by case analysis and more complete (not complete – even just the rules for sum and product are not confluent) when oriented into rewrite rules. Their implementation in Maude looks almost the same:

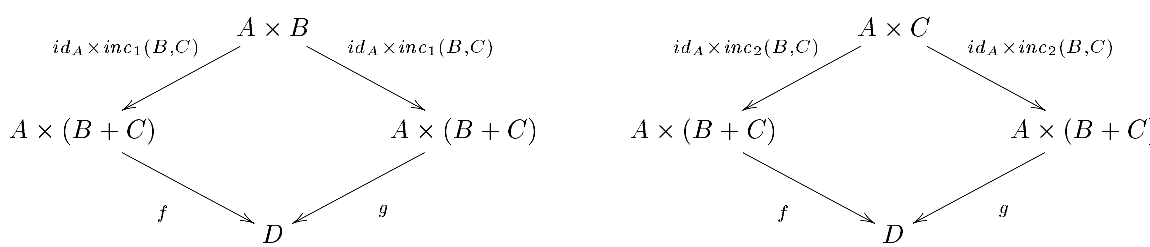
```
eq < F, G ; inc1(B,C) > ; dist(A,B,C)
=  < F,G > ; inc1(A x B, A x C)

eq < F, G ; inc2(B,C) > ; dist(A,B,C)
=  < F,G > ; inc2(A x B, A x C)

eq F x id(B + C) ; dist(A,B,C)
=  dist(D,B,C) ; ( (F x id(B)) + (F x id(C)) )

eq < F,G > ; dist(A,B,C)
=  < id(D), G > ; dist(D,B,C) ;
   ( ( F x id(B) ) + ( F x id(C) ) )
```

To conveniently prove equalities of binary functions by case analysis, we have to introduce some more machinery.

Lemma 4.13. Given two morphisms in a distributive category as in

$$A \times (B + C) \underset{g}{\overset{f}{\rightrightarrows}} D$$

f is equal to g if the following two diagrams commute:



Proof.

$$f = g$$

$\Leftarrow$ $\quad collect\ ;\ f = collect\ ;\ g$ $\qquad$ ($collect$ is an isomorphism)

$\Leftarrow$

$$inc_1(A \times B, A \times C)\ ;\ collect\ ;\ f$$
$$= \quad inc_1(A \times B, A \times C)\ ;\ collect\ ;\ g$$

and

$$inc_2(A \times B, A \times C)\ ;\ collect\ ;\ f$$
$$= \quad inc_2(A \times B, A \times C)\ ;\ collect\ ;\ g$$

(Uniqueness of Copairing)

$\Leftarrow$

$$(id_A \times inc_1(B,C))\ ;\ f$$
$$= \quad (id_A \times inc_1(B,C))\ ;\ g$$

and

$$(id_A \times inc_1(B,C))\ ;\ f$$
$$= \quad (id_A \times inc_1(B,C))\ ;\ g$$

(ExS1 applied on both sides of both equations)

□

Lemma 4.14. given two morphisms in a distributive category as in

$$(A + B) \times (C + D) \underset{g}{\overset{f}{\rightrightarrows}} E$$

then f is equal to g if the following holds:

$$\begin{aligned}
(inc_1(A,B) \times inc_1(C,D)) \; ; \; f &= (inc_1(A,B) \times inc_1(C,D)) \; ; \; g \quad \text{and} \\
(inc_1(A,B) \times inc_2(C,D)) \; ; \; f &= (inc_1(A,B) \times inc_2(C,D)) \; ; \; g \quad \text{and} \\
(inc_2(A,B) \times inc_1(C,D)) \; ; \; f &= (inc_2(A,B) \times inc_1(C,D)) \; ; \; g \quad \text{and} \\
(inc_2(A,B) \times inc_2(C,D)) \; ; \; f &= (inc_2(A,B) \times inc_2(C,D)) \; ; \; g
\end{aligned}$$

Proof. An analogon to Lemma 4.13 can be proven for the case that the sum is the first component of the product. Using it together with Lemma 4.13 yields the desired result. □

Example 4.15. We can define conjunction using the distributivity isomorphism as

```
eq and = dist(bool,1,1) ;
         ( proj1(bool,1) + proj1(bool,1) ) ;
         [ [ f,f ],
           id(bool)
         ] .
```

Here we make a case analysis on the second argument:

1. if it is false we return false no matter what the first argument,
2. if it is true, we return the first argument.

Commutativity of conjunction can now be proven as

```
< proj2(bool,bool), proj1(bool,bool) > ; and = and
```

using Lemma 4.14. The four resulting proof obligations are reduced to `true`, which is hardly surprising since this corresponds to a proof by truth tabling.

5 Final Coalgebras

To represent coalgebras of an endofunctor, I first define the endofunctor as a function on arrows and objects. The functor $T(X) = A \times X$ for streams over A is function `strfunc` and looks as follows:

```
var A : Object .
var F : Arrow .

op  strfunc : Object -> Object .
op  strfunc : Arrow -> Arrow .

eq  strfunc( F ) = id(setA) x F .
eq  strfunc( A ) = setA x A .
```

The constant `setA` is of type `Object` and represents the set A. The constructor `_x_` defined both on objects and arrows is defined in module `PRODUCT`. That `strfunc` really is a functor follows from an easy computation that can even be done by Maude. To that end, skolem constants

```
ops  m n : -> Arrow .
ops  o o1 o2 : -> Object .

eq dom(m) = o1 .
eq cod(m) = o .
eq dom(n) = o .
eq cod(n) = o2 .
```

have to be introduced, and the following reductions are carried out by Maude:

```
Maude> red strfunc( m ; n ) == strfunc(m) ; strfunc(n) .
reduce in STREAMS : strfunc(m ; n) == strfunc(m) ; strfunc(n) .
rewrites: 103
result Bool: true
Maude> red strfunc(id(o)) == id(strfunc(o)) .
rewrites: 13
result Bool: true
```

Now the final coalgebra, for example the set of streams over A can be introduced as an object `streams` and its coalgebra structure as an arrow, which is denoted `next` acording to the intuition that coalgebras represent state spaces and their structure the transition to the next state.

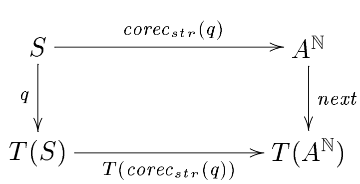
```
  op  next : -> Arrow .
  op  streams  : -> Object .

  eq  dom(next) = streams .
  eq  cod(next) = strfunc(streams) .
```

Finality of the coalgebra can be expressed along the same lines that the universal mapping property of the product was expressed:

First, corecursion on streams is introduced as a constructor on morphisms

```
  var Q : Arrow .

  op corec-str : Arrow -> Arrow? .

  eq dom(corec-str(Q)) = dom(Q) .
  eq cod(corec-str(Q)) = streams .

  cmb corec-str(Q) : Arrow  if  cod(Q) = strfunc(dom(Q)) .
```

and its definition principle (existence) and proof principle (uniqueness) as conditional equations, that express that, given some arrow q there is exactly one arrow, denoted $corec_{str}(q)$, that makes the following diagram commute (streams over A denoted as $A^{\mathbb{N}}$):



Since variables are capital letters in Maude by convention, the `Q` corresponds to the arrow q in the diagram above. The first conditional equation says that `corec-str(Q)` (if defined) is a coalgebra homomorphism, while the second says that any arrow that is a coalgebra homomorphism from the coalgebra with structure `Q` to the final coalgebra (`streams`) is equal to `corec-str(Q)`.

```
var  Q G : Arrow .

ceq  corec-str(Q) ; next = Q ; strfunc(corec-str(Q))
if corec-str(Q) : Arrow .

ceq G = corec-str(Q)  if  G ; next == Q ; strfunc(G) and
                      dom(G) == dom(Q) and
                      cod(G) == streams .
```

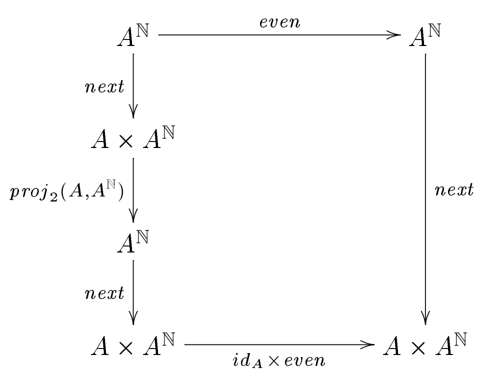
Some function $f: S \to A^{\mathbb{N}}$ that is defined corecusively by a coalgebra structure $q: S \to T(S)$ is expressed as follows:

```
op  f : -> Arrow .

eq  f  =  corec( q ) .
```

Consider the example of the function $even: A^{\mathbb{N}} \to A^{\mathbb{N}}$ from the introduction. Its coinductive definition (a coalgebra structure) $q: S \to T(S)$ is

$$next \ ; \ proj_2(A, A^{\mathbb{N}}) \ ; \ next$$

as you can see by checking the commutativity of the following diagram:



The definition in membership equational logic thus looks as follows:

```
op  even : -> Arrow .

eq  even  =  corec-str( next ; proj2(setA,streams) ; next ) .
```

Using the approach outlined above, it is possible to state coinductive definitions in membership equational logic and to prove coinductive properties using the conditional equation expressing uniqueness of a homomorphism into the final coalgebra (examples will be shown throughout the rest of this section). However, I do not have a full-fledged theorem prover for membership equational logic. Maude can mechanize reasoning in MEL theories by rewriting, but for that, the theories have to be oriented into confluent and terminating conditional rewrite rules. The uniqueness condition can not be executed as a conditional rewrite rule, since there is a variable in the right hand side that does not occur in the left hand side. Unlike the case of the product and the sum, there now is no easy way to remedy that.

Given two morphisms from some domain into the final coalgebra, showing that they are equal requires exhibiting some coalgebra structure on the domain that turns both into coalgebra homomorphisms. Except for simple cases, e.g. where one morphism is the identity or is defined corecursively, this is a problem that can not be solved automatically and will require user interaction.

However, that does not mean that Maude cannot help in proving some theorems coinductively. It just means that the coinduction principle cannot be applied automatically. I will use Maude to prove equations by rewriting in the theory without uniqueness, that is, I *underspecify* the final coalgebra. To prove an equation $f = g$ by coinduction I come up with some coalgebra structure q and check the commutativity of diagrams

$$\begin{array}{ccc} S & \xrightarrow{f} & A^{\mathbb{N}} \\ {\scriptstyle q}\downarrow & & \downarrow{\scriptstyle next} \\ T(S) & \xrightarrow[T(f)]{} & T(A^{\mathbb{N}}) \end{array} \quad \text{and} \quad \begin{array}{ccc} S & \xrightarrow{g} & A^{\mathbb{N}} \\ {\scriptstyle q}\downarrow & & \downarrow{\scriptstyle next} \\ T(S) & \xrightarrow[T(g)]{} & T(A^{\mathbb{N}}) \end{array}$$

by reducing the following in Maude:

```
Maude> reduce  f ; next == q ; strfunc(f) .
```

and

```
Maude> reduce  g ; next == q ; strfunc(g) .
```

If both of those equations are reduced to `true`, I conclude that $f = g$ holds by uniqueness. Since Maude does not know about uniqueness, of course, the commutativity of the two diagrams above can only be shown if it does not depend on uniqueness. In the upcoming subsection about natural numbers there will be a proof that requires two applications of the uniqueness principle.

In general, such proofs by reduction fail. Be it that the rewriting system is not confluent, that the chosen coalgebra structure is wrong, that more than one application of the uniqueness principle is necessary – or maybe the statement to be proved is wrong in the first place. Whatever the reason, we would like to know why.

A way to find out is to inspect normal forms. If the commutativity of the first diagram can not be proved, the normal forms of its two paths can be obtained by

```
Maude> reduce  f ; next .
Maude> reduce  q ; strfunc(f) .
```

Maybe they mostly are equal and it is more promising to use congruence and to try to prove the equalities of the terms at the differing positions. Otherwise one has to revise q.

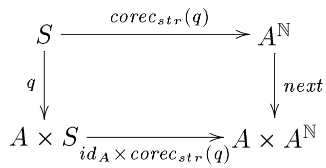
Since this methodology relies on a human inspecting normal forms to guess what lemmas could be useful, introducing `corec-str` as a constructor and defining a function `f` by identifying it with (i.e. rewriting it to) `corec-str(q)` is a bad idea. While for `f` some short and descriptive name can be chosen, `q` usually is a rather large term that does not surrender upon first sight the function it defines. So, instead of `corec( next ; proj2(setA,streams) ; next )` we would like to see `even` in the normal forms. This is achieved by (under)specifying `even` directly:

```
  op  even : -> Arrow .

  eq  dom(even)  =  streams .
  eq  cod(even)  =  streams .

  eq  even ; next  = (next ; proj2(setA,streams) ; next) ; strfunc(even) .
```

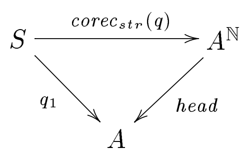
5.1 Streams

Given some fixed set A, the final coalgebra of the functor $T(X) = A \times X$ turns out to be the set of infinite lists (called streams) over A, denoted $A^{\mathbb{N}}$. Its coalgebra structure is the function $\mathit{next}\colon A^{\mathbb{N}} \to A \times A^N$

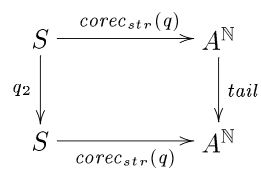
that, given a stream, returns a tuple with the first element and the tail of the stream. It is final since, given some set S and some function $q\colon S \to A \times S$, we can define a function $corec_{str}(q)$ that sends an element s of S to the stream that arises by iterative application of q, starting with s. This function is a coalgebra homomorphism because it makes the following diagram commute:



Since q and $next$ have products as their codomain, there have to be pairings $< q_1, q_2 >= q$ and $< head, tail >= next$. The last diagram can thus be decomposed into



and



That $corec_{str}(q)$ is the only such homomorphism now follows (1) from the fact that two streams are equal if at each position, their elements are equal, (2) the element at position n can be obtained by applying *tail* $n-1$ times, the *head*, and (3) an induction over how many times *tail* is applied. The two last diagrams give induction base and induction step, respectively.

Using the finality of $A^{\mathbb{N}}$ with $< head, tail >$, we corecursively define morphisms $merge\colon A^{\mathbb{N}} \times A^{\mathbb{N}} \to A^{\mathbb{N}}$, $odd, even\colon A^{\mathbb{N}} \to A^{\mathbb{N}}$ (from [JR97])

```
eq  merge ; head  =  proj1(streams, streams) ; head .
eq  merge ; tail  =  < proj2(streams, streams) ,
                       proj1(streams, streams) ; tail > ;
                     merge .

eq  odd ; head  =  head .
eq  odd ; tail  =  tail ; tail ; odd .

eq  even  =  tail ; odd .
```

Merging two streams, the head of the result is the head of the first stream, and the tail of the result is the same as merging the second with the tail of the first. The head of the stream of all elements at odd positions of a stream is the head of the original stream and the tail can be obtained by removing the two first elements from the original stream and the taking the stream of all elements at odd positions.

The only difference from the definitions in [JR97] is that they use variables and lambda abstraction to define these functions elementwise while I use categorical combinators.

Now we can mechanize the proof from [JR97] that merging the two streams that arise by taking all the elements at odd and even positions respectively, yields the original stream.

$$< odd, even >; merge = id_{A^{\mathbb{N}}}$$

As mentioned earlier, this equation can not be proven automatically by reduction in Maude, since we have not formalized the uniqueness property. The user has to make the decision to apply coinduction, i.e. try to prove that

$$< odd, even >; merge; < head, tail > \; = \; < head, tail >; (id_A \times < odd, even >; merge)$$

and

$$id_{A^{\mathbb{N}}} ; < head, tail > = < head, tail > ; (id_A \times id_{A^{\mathbb{N}}})$$

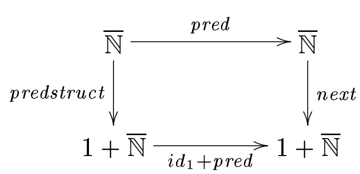
both of which can be reduced to `true` in Maude.

```
Maude> reduce in STREAMS : < odd,even > ; merge ; < head,tail > ==
                           < head,tail > ; strfunc(< odd,even > ; merge) .
rewrites: 289
result Bool: true

Maude> reduce in STREAMS : id(streams) ; < head,tail > ==
                           < head,tail > ; strfunc(id(streams)) .
rewrites: 24
result Bool: true
```

5.2 Natural Numbers

Natural numbers or more correctly the *completed* natural numbers, denoted $\overline{\mathbb{N}}$, i.e. the set of all naturals and one more element, called infinity and denoted ∞, is the final coalgebra of the functor $T(X) = 1 + X$.

The intuition is the following: Given elements from some set S, and some function $next \colon S \to 1 + S$, we can apply *next* to receive either some new element of S or the only element of the singleton. Let's say we do not know anything about those elements. So the only way we can distinguish them is by how often we can apply *next* successively. If this is once, we call the element 0, twice, we call it 1 and so on. Of course, it may also happen that applying q indefinitely often will always result in some new element of S. In that case we call it ∞. Identifying all elements in S that are not distinguishable, i.e. taking the final coalgebra of endofunctor T on sets, gives us the completed naturals. The coalgebra structure *next* is essentially the predecessor, except that its codomain are not the naturals since at zero it yields the element of the singleton. A predecessor $pred \colon \overline{\mathbb{N}} \to \overline{\mathbb{N}}$ (that loops at zero) can be defined as the unique function that makes the following diagram commute.



What should *predstruct* look like? Knowing the intended meaning of *pred* and *next*, we know that *predstruct* ; $(id_1 + pred)$ should be the following:

$$x \mapsto \begin{cases} * & x < 2 \\ x - 2 & 1 < x < \infty \\ \infty & x = \infty \end{cases}$$

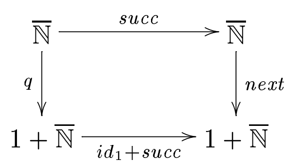
Thus, *predstruct* should be defined as

$$x \mapsto \begin{cases} * & x < 2 \\ x - 1 & 1 < x < \infty \\ \infty & x = \infty \end{cases}$$

This function can be represented as a morphism in a category with sums, terminal object and final coalgebra object:

```
eq  predstruct  =  next ; [ inc1(1,nat),
                            next ; ( id(1) + succ )
                          ] .
```

The successor can not be defined corecursively. The problem is, that a function $q\colon \overline{\mathbb{N}} \to 1 + \overline{\mathbb{N}}$ to make this diagram commute



would have to send 0 to some $x \in \overline{\mathbb{N}}$ such that $succ(x) = 0$. There are no such elements and therefore there is no such function q.

We can, however, define the successor by composition: $succ = inc_2(1, \overline{\mathbb{N}})\ ;\ next^{-1}$. The inverse of $next$ can be defined corecursively as $corec(id_1 + next)$. It will be denoted `nextinv`. Functions $corec(id_1 + next)$ and $next^{-1}$ are indeed equal, since $next^{-1}$ makes the corresponding finality diagram commute and there can only be one such function.

To prove $pred(succ(x)) = x$ we now try to prove that $pred\ ;\ succ$ is a endomorphism on the final coalgebra and thus equal to $id_{\overline{\mathbb{N}}}$.

```
Maude> reduce succ ; pred ; next == next ; natfunc( succ ; pred ) .
```

which is `false`. Inspecting the normal forms by

```
reduce succ ; pred ; next .
rewrites: 758
result Arrow: next ; [inc1(1, nat),next ; nextinv ; inc2(1, nat) ; nextinv ;
    pred ; inc2(1, nat)]

reduce next ; natfunc(succ ; pred) .
rewrites: 227
result Arrow: next ; [inc1(1, nat),inc2(1, nat) ; nextinv ; pred ; inc2(1,
    nat)]
```

reveals that they differ only in a "`next ; nextinv`" occuring in the first normal form that does not appear in the second. This can be dropped since $next\ ;\ next^{-1}$ is a homomorphism on the final coalgebra and thus equal to $id_{\overline{\mathbb{N}}}$.

That can be shown by reducing

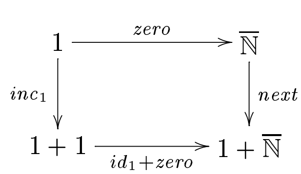
```
next ; nextinv ; next == next ; natfunc(next ; nextinv) .
```

to `true`. We can now add

```
eq  next ; nextinv = id(nat) .
```

as a lemma to our rewrite system to prove the original equation automatically.

The constant 0 can easily be defined corecursively as



Virtue of the coalgebraic (as opposed to the algebraic) definition of the naturals is the ease in defining binary functions. While one has to resort to function types and currying to inductively define addition of naturals, defining it coinductively is straightforward. We just have to come up with a function *addstruct* such that commutativity of the following diagram forces *add* to be addition:

$$\begin{array}{ccc} \overline{\mathbb{N}} \times \overline{\mathbb{N}} & \xrightarrow{add} & \overline{\mathbb{N}} \\ {\scriptstyle addstruct} \downarrow & & \downarrow {\scriptstyle next} \\ 1 + \overline{\mathbb{N}} \times \overline{\mathbb{N}} & \xrightarrow[id_1 + add]{} & 1 + \overline{\mathbb{N}} \end{array}$$

Thus, *addstruct* should be defined as

$$(n, n') \mapsto \begin{cases} * & n = n' = 0 \\ (n, n' - 1) & n = 0 \quad n' > 0 \\ (n', n - 1) & n > 0 \end{cases}$$

This is now represented using categorical combinators. To acquire information about a natural number (i.e. to know whether it is 0 or not) *next* is applied. "Destructor" seems quite an apt name for it, which is why the first argument is copied before applying *next*. The distributivity isomorphism is put in explicitly, the "-1" is now implicit in the *next*. The case analysis is now structured into two nested copairings. Note that,

1. there is a reason I chose $(n', n-1)$ in the last case instead of $(n-1, n')$ which would have been correct, too. Opting for the second, I would have had to insert the commutativity isomorphism (`< proj2(nat, nat), proj1(nat, nat) >`) before the last inclusion in the definition below, and

2. that I use strict associativity: the second argument of the outer copairing should have domain $(\overline{\mathbb{N}} \times \overline{\mathbb{N}}) \times \overline{\mathbb{N}}$ but the domain of `proj2(nat, nat x nat)` is $\overline{\mathbb{N}} \times (\overline{\mathbb{N}} \times \overline{\mathbb{N}})$

```
eq  addstruct  =
    < id(nat x nat), proj1(nat, nat) ; next > ;
    dist(nat x nat, 1, nat) ;
    [
      proj1(nat x nat, 1) ;
      ( id(nat) x next ) ;
      dist(nat,1,nat) ;
      [
        proj2(nat,1) ;
        inc1(1, nat x nat),

        inc2(1, nat x nat)
      ],

      proj2(nat, nat x nat) ;
      inc2(1, nat x nat)
    ] .
```

The only statement about *add* that I was able to prove automatically with Maude was

$$< zero \; , \; zero > \; ; \; add = zero$$

(again by manually applying the uniqueness principle). I did not spend much time on more useful properties, such as commutativity. Proving automatically that this corecursive *add* does indeed correspond to our notion of addition of naturals, i.e. that

$$add(zero, x) = x \quad \wedge \quad add(succ(x), y) = succ(add(x, y))$$

did not succeed by just applying uniqueness and then automatically rewriting. This is of course just a problem of which tool to choose for reasoning in those theories. The propositions are true, they are proven (using bisimulation) in [R96]. Surely replacing Maudes rewrite engine by some interactive equational theorem prover would help.

5.3 Sequences

The union of the set of streams and the set of (finite) lists over some fixed set A is called the set of sequences over A and denoted A^∞. Together with a function $next\colon A^\infty \to 1 + A \times A^\infty$ it is the final coalgebra of the functor $T(X) = 1 + A \times X$. Intuitively, a coalgebra of this functor is a deterministic automaton or a process and its coalgebra structure the transition to the next state. There is no input and the output alphabet of the automaton is A. Given some state of the automaton, applying *next* either yields an element of A together with the next state or it yields $*$, meaning that now the automaton has reached a final state and terminated. In the final coalgebra all states that are bisimilar, i.e. not distinguishable using *next*, are identified. Its elements thus correspond to the behaviours that the states of an automaton exhibit. And those behaviours correspond to sequences over A. Seen in this light, streams are behaviours of deterministic automata that cannot terminate, i.e. where each state has exactly one transition to a next state.

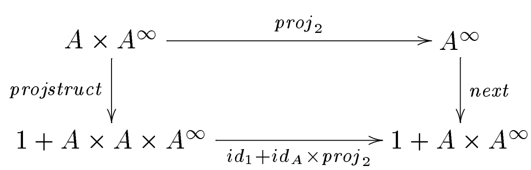
We now define the empty sequence $nil\colon 1 \to A^\infty$, the sequence constructor $cons\colon A \times A^\infty \to A^\infty$, the tail $tail\colon A^\infty \to A^\infty$ and concatenation $conc\colon A^\infty \times A^\infty \to A^\infty$ on sequences, by exhibiting the appropriate coalgebra structures:

```
eq  nilstruct  =  inc1(1, setA x 1) .

eq  nextinvstruct  =  id(1) + (id(setA) x next) .

eq  cons  =  inc2(1, setA x seq) ; nextinv .

eq  tailstruct  =  next ; [
                              inc1(1, setA x seq) ,

                              proj2(setA, seq) ; next ;
                              (id(1) + < proj1(setA, seq)  , cons >)
                            ] .


eq  concstruct  =  < id(seq x seq) ,
                     proj1(seq, seq) ; next
                   > ;
                   dist(seq x seq, 1, setA x seq) ;
                   [
                     proj1(seq x seq, 1) ;
                     ( id(seq) x next ) ;
                     dist(seq, 1, setA x seq);
                     [
                       proj2(seq, 1) ;
                       inc1(1, setA x seq x seq)
                       ,
                       < proj2(seq, setA x seq) ,
                         proj1(seq, setA x seq) > ;
                       inc2(1, setA x seq x seq)
                     ]
                     ,
```

```
    ( proj2(seq, seq) x id(setA x seq) ) ;
    < proj2(seq, setA x seq) ,
      proj1(seq, setA x seq) > ;
    inc2(1, setA x seq x seq)
  ] .
```

Since the completed naturals can be seen as a special case of sequences where $A = 1$, the functions defined above are generalizing *zero*, *succ*, *pred* and *add*, respectively, and so are their defining coalgebra structures. The only difference lies in *concstruct*: the commutativity isomorphism is used twice, but does not occur in *addstruct*. As noted before, we could introduce it in *addstruct* as well, it would not make a difference since *add* is commutative. But *conc* is not, therefore the isomorphism in *concstruct* is crucial.

Proving $cons\ ;\ tail = proj_2$ now is a bit more complex than proving $pred\ ;\ succ = id_{\mathbb{N}}$. To prove the second equation coinductively, coming up with the appropriate defining coalgebra structure is immediate, because $id_{\mathbb{N}}$ is a coalgebra homomorphism on the final coalgebra. But of course $proj_2$ is not. Here we have to find a function *projstruct* that makes the following diagram commute:



That is easy in this case since all that has to be done is applying *next* and making sure that the result appears in the appropriate places in the ternary product. The following definition does the job.

```
eq projstruct  =  proj2(setA, seq) ; next ;
                  ( id(1)
                    +
                    < proj1(setA, seq) , id(setA x seq) >
                  ) .
```

We can check whether this coalgebra structure does indeed define $proj_2$ by executing

```
Maude> reduce  proj2(setA, seq) ; next ==  projstruct ; seqfunc(proj2(setA, seq)) .
```

where `seqfunc` is the functor for sequences $T(X) = 1 + A \times X$. The equation reduces to `true`.

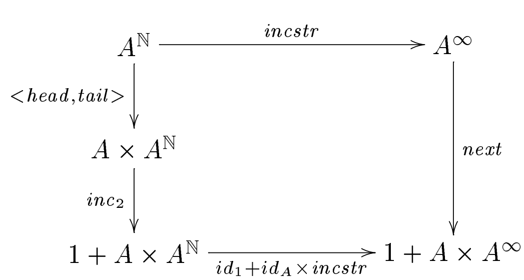
Knowing that, we can prove $cons\ ;\ tail$ equal to $proj_2$ by showing that it too makes the above diagram commute.

```
Maude> reduce  cons ; tail ; next  ==  projstruct ; seqfunc(cons ; tail) .
```

also yields `true`.

Not surprisingly, $< nil\ ,\ nil >\ ;\ conc = nil$ can also be proven.

Streams over A are a subset of the sequences over A. The corresponding inclusion can be defined corecursively:

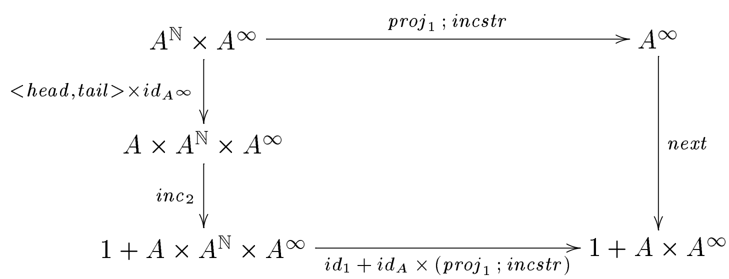


Now we want to prove that $conc(x,y)$ is not distinguishable from x if x is an infinite sequence (i.e. a stream). We thus prove that in the final coalgebra the two are equal. Rephrasing this in element-free language yields $incstr \times id_{A^\infty}$; $conc = proj_1(A^{\mathbb{N}}, A^\infty)$; $incstr$.

Here again we have to come up with the appropriate coalgebra structure that allows to show equality by uniqueness. We find that

```
eq projincstrstruct =  ( < head,tail > x id(seq) ) ;
                      inc2(1,setA x streams x seq) .
```

does the job since we can prove commutativity of the following diagram by reduction:



Since replacing $proj_1(A^{\mathbb{N}}, A^\infty)$; $incstr$ by $incstr \times id_{A^\infty}$; $conc$ also makes the above diagram commute, which also can be proven by reduction, we conclude by uniqueness that they are equal.

6 Coinduction Strategies

Strategies are procedures that inspect a proposition that has to be proved and determine what steps should be taken to prove it. In our categorical framework the difficult, creative part in a coinductive proof of $h_1 = h_2$ with $h_1, h_2 \colon S \to F$, F being the final coalgebra, is coming up with the appropriate coalgebra structure $q \colon S \to T(S)$ that turns both h_1 and h_2 into coalgebra homomorphisms. An essential building block for coinduction strategies would thus be a function, say

```
op  codef  :  Arrow -> Arrow? .
```

that, given some morphism h, yields a coinductive definition of h, that is, some q such that $h = corec(q)$. Such a *codef* could only be defined on morphisms into the final coalgebra, but would necessarily be partial even on those since not all such morphisms are coinductively defineable, see e.g. *succ*. It would also have to make some choice since two different $q_1, q_2 \colon S \to T(S)$ might coinductively define the same function, see e.g. *add*.

A strategy that has to prove $h_1 = h_2$ could check whether *codef* is defined on one side of the equation, and if it is, say on h_1, return the resulting proof obligation:

$$h_2 \ ; \ next = codef(h_1) \ ; \ T(h_2)$$

Since most proofs in section 5 involved proving equal a function to either the identity or to some corecursively defined function, I already used the definition of *codef* as

$$codef(id(F)) = next codef(corec(q)) = q.$$

On isomorphisms, the definition of *codef* is straightforward, too. That is useful, remember that the structures of final coalgebras are isomorphisms. For a copairing of two coinductively defined morphisms, its coinductive definition can be constructed from the coinductive definitions of its arguments.

On other types of morphisms *codef* can be defined for a specific endofunctor. That happened for functor $T(X) = 1 + A \times X$ and morphism $proj_2$ (see page 28).

The following table gives a – still rather incomplete – picture of what such a function *codef* should look like. $(F, next)$ denotes the final coalgebra, h is some morphism with $cod(h) = F$ and i and j are isomorphisms.

h	$codef(h)$
id_F	$next$
$corec(q)$	q
i	$i \; ; \; next \; ; \; T(i^{-1})$
$j \; ; \; q$	$j \; ; \; codef(q) \; ; \; T(j^{-1})$
$[q_1, q_2]$	$[codef(q_1) \; ; \; T(inc_1(A, B)),$ $codef(q_2) \; ; \; T(inc_2(A, B))]$

7 Conclusions

An approach to coinductive theorem proving has been proposed that is based on the finality of coalgebras expressed in terms of equalities of morphisms in a category with structure. Such categories have been specified in membership equational logic, which seems the natural candidate for that purpose because it supports convenient treatment of partiality. Then the Maude system has been used to prove some statements about streams, natural numbers and sequences.

This approach allows to prove some inductive and coinductive properties entirely by equational reasoning because universally quantified first order propositions translate into ground equations of morphisms. However, complexity does not just vanish. Coinductive function definitions have to be given in categorical combinator language, i.e. by composing projections, inclusions, pairings, copairings etc. Certainly the definition of *addstruct* using elements (page 26) is easier to understand than its transliteration into combinator language. When defining a binary function by case analysis, nobody wants to worry about where to insert the distributivity isomorphisms. It has to be seen how this language can be made more intuitive. Also, just because inductive and coinductive proofs can be reduced to equational proofs, that by itself does not mean that they become simpler. Their complexity depends on two things:

1. The structure of the underlying category. In the case of streams, this structure consists of the product only. Since the equality of morphisms in such a categories can be decided by e.g. module `CARTESIAN`, proofs are easy to the extent that they can be done automatically by Maude. Sequences, in contrast, live in a category with product and sum and distributivity of the first over the second. Equality of morphisms in such categories is widely thought to be decidable, but to my knowledge, there is no proof of that.
2. The difficulty of finding the right coalgebra structure for applying the coinduction principle. For all propositions proved in section 5 this structure was found in a straightforward fashion, that could be mechanized as outlined in section 6. That is not true in general, consider for example

$$add(succ(x), y) = succ(add(x, y)).$$

Coinductive proofs have not fully been mechanized inside Maude, because the coinduction principle is just the inference rule that has *not* been mechanized. Actually, the user decides to apply coinduction on a proposition, comes up with the coalgebra structure, determines the resulting proof obligations, uses Maude to prove them and then concludes that the original proposition must be true. This is necessitated by the fact that Maude is a rewrite engine, not a theorem prover. In a full-fledged MEL theorem prover, the coinduction principle can be mechanized as outlined in section 5. Of course Maude/Rewriting Logic could be used to *implement* an inductive MEL theorem prover as demonstrated in [CDEM99], or the existing theorem prover could be extended. The prover there “as is” is not suited for reasoning in structured categories, because it can prove equations only by rewriting and does not support interactive application of equations. Basing an interactive MEL theorem prover on Maude has the obvious advantages of

1. a straightforward representation of the inference rules as conditional rewrite rules and
2. inheriting Maude's fast rewriting and matching-modulo algorithms.

But even so, it still is a new implementation of an interactive theorem prover and since my interest is not in writing yet another theorem prover, but in ways to prove coinductive properties equationally, that should be the last option if none of the existing ones can be adapted to suit this task.

Future Work The foremost task is to find the right ground on which to base tool support for reasoning in structured categories. While Maude has its merits, "off-the-shelf" provers like Coq, Isabelle and PVS should be considered. Another question is what other properties besides strict associativity should be required of the structured category in which is reasoned. An interesting candidate is strict distributivity. What is its interpretation in **Set**? Does it simplify definitions and proofs? To aid mechanization, a decision procedure for equality of morphisms in the free distributive category generated by a graph would be helpful. Finally, the coinduction strategies outlined in section 6 should be elaborated and extended a lot.

Acknowledgements I would like to thank my supervisor Prof. Horst Reichel for having provided an interesting problem and numerous advice on how to solve it. I am also very thankful to José Meseguer for his encouragement and for making possible my half-year stay at SRI during which most of this work was done. For his interest in my work and his helpful comments, I would like to thank Mark-Oliver Stehr. Christiano Braga gladly answered my questions about Maude. Hendrik Tews and Dmitri Schamschurko read preliminary versions of this work and prompted improvements.

References

[BW99] Michael Barr and Charles Wells. Category Theory for Computing Science. 3rd Ed. Les Publications CRM, Montréal 1999.

[JR97] Bart Jacobs and Jan Rutten. A Tutorial on (Co)Algebras and (Co)Induction. Bulletin of the EATCS,(62):222-259, June 1997.

[R96] J.J.M.M. Rutten. Universal Coalgebra: a theory of systems. CWI Report CS-R9652, 1996.

[M98] José Meseguer. Membership Algebra as a semantic framework for equational specification. In F. Parisi-Presicce, ed., Proc. WADT'97,18-61, Springer LNCS 1376, 1998.

[Maude99] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: Specification and Programming in Rewriting Logic. Manuscript, Computer Science Laboratory SRI International, March 8, 1999.

[CDEM99] Manuel Clavel, Francisco Durán, Steven Eker, José Meseguer. Building equational proving tools by reflection in rewriting logic. In Proc. of the CafeOBJ Symposium '98, Numazu, Japan. CafeOBJ Project, April 1998.

[CS00] J.R.B. Cockett and R.A.G. Seely. Finite sum-product logic. Manuscript, April 2000. To be published.

[P97] L.C. Paulson. Mechanizing coinduction and corecursion in higher order logic. Journ. of Logic and Computation, 7:175-204, 1997.

[HJ97] U. Hensel and B. Jacobs. Proof principles for datatypes with iterated recursion. Technical Report CSI-R9703, Computing Science Institute, University of Nijmegen, 1997.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, volume B, pages 243-320. Elsevier, 1990.

A Statement of Academical Honesty

(Selbständigkeitserklärung)

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbst angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dresden, den 20. Mai 2000

Kai Brünnler

B Maude Functional Modules

List of Modules

```
fmod CARTESIAN is

  sorts   Object Arrow .

  op id           :  Object -> Arrow .
  op _;_          :  Arrow Arrow -> Arrow .
  op _x_          : Object Object -> Object .
  ops proj1 proj2 : Object Object -> Arrow .
  op <_,_>        : Arrow Arrow -> Arrow .
  op 1  : -> Object .
  op !  : Object -> Arrow .

  vars F G H      : Arrow .
  vars A B        : Object .

  eq  dom( proj1(A,B) ) = A x B .
  eq  dom( proj2(A,B) ) = A x B .
  eq  cod( proj1(A,B) ) = A .
  eq  cod( proj2(A,B) ) = B .
  eq  dom( < F,G > ) = dom(F) .
  eq  cod( < F,G > ) = cod(F) x cod(G) .
  eq  dom(!(A)) = A .
  eq  cod(!(A)) = 1 .

  eq    F ; ( G ; H ) =  ( F ; G ) ; H .
  eq    id(A) ; F  =  F .
  eq    F ; id(A)  =  F .

  eq    < F,G > ; proj1(A,B) = F .
  eq    < F,G > ; proj2(A,B) = G .
  eq    < F ; proj1(A,B), F ; proj2(A,B) > = F .
  eq    F ; < G,H >  =  < F ; G, F ; H > .
  eq    < proj1(A,B), proj2(A,B) >  =  id(A x B) .

  eq   proj1(1,A)  =  !(1 x A) .
  eq   proj2(A,1)  =  !(A x 1) .
  eq   !(1)  =  id(1) .
  eq   F ; !(A)  =  !(dom(F)) .
  eq  < !(B), F ; proj2(1,A) >  =  F .
  eq  < F ; proj1(A,1), !(B) >  =  F .
  eq   < !(1 x A), proj2(1,A) >  =  id(1 x A) .
  eq   < proj1(A,1), !(A x 1) >  =  id(A x 1) .
  eq  < id(1), F ; proj2(1,A) >  =  F .
  eq  < F ; proj1(A,1), id(1) >  =  F .

endfm
```

Module 2: Decision Procedure for the Word Problem in a Cartesian Category

```
fmod PRODUCT is

  including CATEGORY .

  op _x_                : Object Object -> Object .
  op _x_                : Arrow Arrow -> Arrow .
  ops proj1 proj2       : Object Object -> Arrow .
  op <_,_>              : Arrow Arrow -> Arrow? .

  vars F G H            : Arrow .
  vars A B              : Object .

  eq  dom( proj1(A,B) ) = A x B .
  eq  dom( proj2(A,B) ) = A x B .
  eq  cod( proj1(A,B) ) = A .
  eq  cod( proj2(A,B) ) = B .

  eq  dom( < F,G > ) = dom(F) .
  eq  cod( < F,G > ) = cod(F) x cod(G) .

--- express product as pairing

  eq  F x G = < proj1( dom(F), dom(G) ) ; F ,
              proj2( dom(F), dom(G) ) ; G > .

--- pairing is defined for arrows w/ the same domain only

  cmb   < F,G > : Arrow             if  dom(F) == dom(G) .

--- existence of pairing

  ceq   < F,G > ; proj1(A,B) = F   if  dom(F) == dom(G) and
                                       cod(F) == A and
                                       cod(G) == B .

  ceq   < F,G > ; proj2(A,B) = G   if  dom(F) == dom(G) and
                                       cod(F) == A and
                                       cod(G) == B .

--- uniqueness of pairing

  ceq   < F ; proj1(A,B), F ; proj2(A,B) > = F  if  cod(F) == A x B .

--- and this has to be added to make it confluent

  ceq   F ; < G,H >  =  < F ; G, F ; H >  if  dom(G) == dom(H) and
                                              dom(G) == cod(F) .

  eq    < proj1(A,B), proj2(A,B) >  =  id(A x B) .

endfm
```

Module 3: Binary Products

```
fmod PRODUCT-ASSOC is

  including CATEGORY .

  op _x_                : Object Object -> Object [assoc] .
  op _x_                : Arrow Arrow -> Arrow .
  ops proj1 proj2       : Object Object -> Arrow .
  op <_,_>              : Arrow Arrow -> Arrow? [assoc] .

  vars F G H            : Arrow .
  vars A B C            : Object .

...
(same statements as in PRODUCT)
...


--- strict associativity

  eq  proj1(A x B, C) ; proj1(A,B)  =  proj1(A, B x C) .
  eq  proj2(A, B x C) ; proj2(B,C)  =  proj2(A x B, C) .

  eq  < proj1(A x B, C) ; proj2(A,B), proj2(A x B, C) >  =  proj2(A, B x C) .
  eq  < proj1(A, B x C), proj2(A, B x C) ; proj1(B,C) >  =  proj1(A x B, C) .

  ceq  < F,G > ; proj1(A, B x C)  =  F ; proj1(A,B)
  if  dom(F) == dom(G)  and
      cod(F) == A x B   and
      cod(G) == C .

  ceq  < F,G > ; proj2(A x B, C)  =  G ; proj2(B,C)
  if  dom(F) == dom(G)  and
      cod(F) == A       and
      cod(G) == B x C .

endfm
```

Module 4: Strictly Associative Product

```
fmod TERMINAL is

  including CATEGORY .
  including PRODUCT-ASSOC .

  op 1  : -> Object .
  op !  : Object -> Arrow .

  var A B : Object .
  var F   : Arrow .

  eq  dom(!(A)) = A .
  eq  cod(!(A)) = 1 .


  eq   proj1(1,A)  =  !(1 x A) .
  eq   proj2(A,1)  =  !(A x 1) .
  eq   !(1)  =  id(1) .
  ceq  F ; !(A)  =  !(dom(F))  if  cod(F) == A .

  ceq  < !(B), F ; proj2(1,A) >  =  F  if  cod(F) == 1 x A  and
                                           dom(F) == B .
  ceq  < F ; proj1(A,1), !(B) >  =  F  if  cod(F) == A x 1  and
                                           dom(F) == B .

  eq   < !(1 x A), proj2(1,A) >  =  id(1 x A) .

  eq   < proj1(A,1), !(A x 1) >  =  id(A x 1) .

  ceq  < id(1), F ; proj2(1,A) >  =  F  if  cod(F) == 1 x A  and
                                            dom(F) == 1 .
  ceq  < F ; proj1(A,1), id(1) >  =  F  if  cod(F) == A x 1  and
                                            dom(F) == 1 .

endfm
```

Module 5: Terminal Object

```
fmod SUM is

  including CATEGORY .

  op _+_              : Object Object -> Object .
  op _+_              : Arrow Arrow -> Arrow .
  ops inc1 inc2       : Object Object -> Arrow .
  op [_,_]            : Arrow Arrow -> Arrow? .

  vars F G H          : Arrow .
  vars A B            : Object .

  eq  cod( inc1(A,B) ) = A + B .
  eq  cod( inc2(A,B) ) = A + B .
  eq  dom( inc1(A,B) ) = A .
  eq  dom( inc2(A,B) ) = B .

  eq  dom( [ F,G ] ) = dom(F) + dom(G) .
  eq  cod( [ F,G ] ) = cod(F) .

--- express sum as copairing

  eq  F + G = [ F ; inc1( cod(F), cod(G) ),
              G ; inc2( cod(F), cod(G) ) ] .

--- copairing is defined for arrows w/ the same codomain only

  cmb   [ F,G ] : Arrow            if   cod(F) == cod(G) .

--- existence of copairing

  ceq   inc1(A,B) ; [ F,G ] = F  if  cod(F) == cod(G) and
                                     dom(F) == A and
                                     dom(G) == B .

  ceq   inc2(A,B) ; [ F,G ] = G  if  cod(F) == cod(G) and
                                     dom(F) == A and
                                     dom(G) == B .

--- uniqueness of copairing

  ceq   [ inc1(A,B) ; F, inc2(A,B) ; F ] = F  if  dom(F) == A + B .

--- and this has to be added to make it confluent

  ceq  [ F,G ] ; H  =  [ F ; H, G ; H ]  if  cod(F) == cod(G) and
                                             cod(F) == dom(H) .

  eq  [ inc1(A,B), inc2(A,B) ]  =  id(A + B) .

endfm
```

Module 6: Binary Sums

```
fmod DISTRIBUTIVE is

  including CATEGORY .
  including PRODUCT-ASSOC .
  including SUM .

  op  dist : Object Object Object -> Arrow .

  vars  A B C D  : Object .
  vars  F G      : Arrow .

  eq  dom( dist(A,B,C) )  =  A x (B + C) .
  eq  cod( dist(A,B,C) )  =  (A x B) + (A x C) .

  eq  dist(A,B,C) ;
      [ < proj1(A,B), proj2(A,B) ; inc1(B,C) > ,
        < proj1(A,C), proj2(A,C) ; inc2(B,C) > ]  =  id(A x (B + C)) .

  eq  < proj1(A,B), proj2(A,B) ; inc1(B,C) > ; dist(A,B,C)
      =  inc1(A x B, A x C) .

  eq  < proj1(A,C), proj2(A,C) ; inc2(B,C) > ; dist(A,B,C)
      =  inc2(A x B, A x C) .

  ceq  < F, G ; inc1(B,C) > ; dist(A,B,C)
       =  < F,G > ; inc1(A x B, A x C)
       if  dom(F) == dom(G)  and  cod(F) == A  and  cod(G) == B .

  ceq  < F, G ; inc2(B,C) > ; dist(A,B,C)
       =  < F,G > ; inc2(A x B, A x C)
       if  dom(F) == dom(G)  and  cod(F) == A  and  cod(G) == C .

  ceq  < F, inc1(B,C) > ; dist(A,B,C)
       =  < F,id(B) > ; inc1(A x B, A x C)
       if  dom(F) == B  and  cod(F) == A .

  ceq  < F, inc2(B,C) > ; dist(A,B,C)
       =  < F,id(C) > ; inc2(A x B, A x C)
       if  dom(F) == B  and  cod(F) == A .

  ceq  < proj1(A, B + C) ; F, proj2(A, B + C) > ; dist(D,B,C)
      =  dist(A,B,C) ; ( ( F x id(B) ) + ( F x id(C) ) )
      if  dom(F) == A  and  cod(F) == D .

  ceq  < F,G > ; dist(A,B,C)
      =  < id(dom(F)), G > ; dist(dom(F),B,C) ;
         ( ( F x id(B) ) + ( F x id(C) ) )
      if  dom(F) == dom(G)  and cod(F) == A  and
          cod(G) == B + C and not isId(F) .

  op   isId : Arrow -> Bool .
  eq   isId(id(A)) = true .

endfm
```

Module 7: Distributivity

```
fmod BOOLEAN is

  including CATEGORY .
  including SUM .
  including PRODUCT-ASSOC .
  including TERMINAL .
  including DISTRIBUTIVE .


  op bool            : -> Object .
  ops f t            : -> Arrow .
  ops not and        : -> Arrow .


  eq  bool  =  1 + 1 .
  eq  f     =  inc1(1,1) .
  eq  t     =  inc2(1,1) .

  eq  not   =  [inc2(1,1), inc1(1,1)] .

  eq  and  =  dist(bool,1,1) ;
              ( proj1(bool,1) + proj1(bool,1) ) ;
              [ [ f,f ],
                id(bool)
              ] .

endfm
```

Module 8: Booleans

```
fmod PARAMETER is

  including CATEGORY .

  op setA : -> Object .

endfm
```

Module 9: Parameter Object

```
fmod STREAM-FUNCTOR is

  including PRODUCT-ASSOC .
  including PARAMETER .

  var A : Object .
  var F : Arrow .

  op  strfunc : Object -> Object .
  op  strfunc : Arrow -> Arrow .

  eq    strfunc( F ) = id(setA) x F .
  eq    strfunc( A ) = setA x A .

endfm
```

Module 10: Functor $T(X) = A \times X$

```
fmod NATURALS-FUNCTOR is

  including SUM .
  including TERMINAL .

  op  natfunc : Object -> Object .
  op  natfunc : Arrow -> Arrow .

  var A : Object .
  var F : Arrow .

  eq    natfunc( F ) = id(1) + F  .
  eq    natfunc( A ) = 1 + A .

endfm
```

Module 11: Functor $T(X) = 1 + X$

```
fmod SEQUENCE-FUNCTOR is

  including PRODUCT-ASSOC .
  including SUM .
  including TERMINAL .
  including PARAMETER .

  var S : Object .
  var F : Arrow .

  op  seqfunc : Object -> Object .
  op  seqfunc : Arrow -> Arrow .

  eq    seqfunc( F ) = id(1) + ( id(setA) x F ) .
  eq    seqfunc( S ) = 1 + ( setA x S ) .

endfm
```

Module 12: Functor $T(X) = 1 + A \times X$

```
fmod STREAMS is

  including STREAM-FUNCTOR .

  ops head tail : -> Arrow .
  ops merge odd even : -> Arrow .
  op  streams : -> Object .

  eq dom(head) = streams .
  eq cod(head) = setA .
  eq dom(tail) = streams .
  eq cod(tail) = streams .

  eq dom(merge) = streams x streams .
  eq cod(merge) = streams .
  eq dom(odd) = streams .
  eq cod(odd) = streams .
  eq dom(even) = streams .
  eq cod(even) = streams .


  eq  merge ; head  =  proj1(streams, streams) ; head .
  eq  merge ; tail  =  < proj2(streams, streams) ,
                         proj1(streams, streams) ; tail > ; merge .

  eq  odd ; head  =  head .
  eq  odd ; tail  =  tail ; tail ; odd .

  eq  even  =  tail ; odd .

endfm
```

Module 13: Streams

```
fmod NATURALS is

  including DISTRIBUTIVE .
  including NATURALS-FUNCTOR .

  op  next : -> Arrow .
  op  nat  : -> Object .
  eq  dom(next) = nat .
  eq  cod(next) = natfunc(nat) .

  ops nextinv succ : -> Arrow .
  eq  dom(nextinv) = natfunc(nat) .
  eq  cod(nextinv) = nat .

  eq  nextinv ; next  =  natfunc(next) ; natfunc(nextinv) .
  eq  succ  =  inc2(1,nat) ; nextinv .

  ops pred predstruct : -> Arrow .
  eq  dom(pred) = nat .
  eq  cod(pred) = nat .

  eq  predstruct  =  next ; [ inc1(1,nat),
                              next ; ( id(1) + succ )
                            ] .
  eq  pred ; next  =  predstruct ; natfunc(pred) .


  ops  zero zerostruct : -> Arrow .
  eq  zerostruct  =  inc1(1,1) .
  eq  dom(zero)  =  1 .
  eq  cod(zero)  = nat .

  eq  zero ; next  =  zerostruct ; natfunc(zero) .

endfm
```

Module 14: Completed Natural Numbers

```
fmod ADD is

  including NATURALS .

  ops  addstruct add : -> Arrow .
  eq  dom(addstruct)  =  nat x nat .
  eq  cod(addstruct)  =  1 + (nat x nat) .
  eq  dom(add)  =  nat x nat .
  eq  cod(add)  =  nat .

  eq  add ; next = addstruct ; natfunc(add) .

  eq  addstruct  =
      < id(nat x nat), proj1(nat, nat) ; next > ;
      dist(nat x nat, 1, nat) ;
      [
        proj1(nat x nat, 1) ;
        ( id(nat) x next ) ;
        dist(nat,1,nat) ;
        [
          proj2(nat,1) ;
          inc1(1, nat x nat),
          inc2(1, nat x nat)
        ],
        proj2(nat, nat x nat) ;
        inc2(1, nat x nat)
      ] .

endfm
```

Module 15: Addition of Completed Natural Numbers

```
fmod SEQUENCES is

  including SEQUENCE-FUNCTOR .
  including DISTRIBUTIVE .

  op  next : -> Arrow .
  op  seq : -> Object .

  ops nextinv stail cons : -> Arrow .
  ops proj2struct stailstruct : -> Arrow .

  eq  dom(next)  =  seq .
  eq  cod(next)  =  seqfunc(seq) .

  eq  dom(nextinv)  =  seqfunc(seq) .
  eq  cod(nextinv)  =  seq .
  eq  dom(stail)  =  seq .
  eq  cod(stail)  =  seq .
  eq  dom(cons)  =  setA x seq .
  eq  cod(cons)  =  seq .

  eq  nextinv ; next  =  seqfunc(next) ; seqfunc(nextinv) .

  eq  cons  =  inc2(1, setA x seq) ; nextinv .

  eq  stail ; next = stailstruct ; seqfunc(stail) .

  eq  stailstruct  =  next ; [
                              inc1(1, setA x seq) ,

                              proj2(setA, seq) ; next ;
                              (id(1) + < proj1(setA, seq) , cons >)
                            ] .

  eq proj2struct  =  proj2(setA, seq) ; next ;
                     [ inc1(1, setA x setA x seq) ,
                       < proj1(setA, seq) , id(setA x seq) > ;
                       inc2(1, setA x (setA x seq)) ] .

  eq  next ; nextinv  =  id(seq) .

  ops nil nilstruct : -> Arrow .
  eq  dom(nil)  =  1 .
  eq  cod(nil)  =  seq .
  eq  nilstruct  =  inc1(1, setA x 1) .
  eq  nil ; next  =  nilstruct ; seqfunc(nil) .

endfm
```

Module 16: Sequences

```
fmod CONCATENATE is

  including SEQUENCES .

  ops conc concstruct : -> Arrow .
  eq  dom(conc)  =  seq x seq .
  eq  cod(conc)  =  seq .
  eq  concstruct  =  < id(seq x seq) ,
                       proj1(seq, seq) ; next
                     > ;
                     dist(seq x seq, 1, setA x seq) ;
                     [
                       proj1(seq x seq, 1) ;
                       ( id(seq) x next ) ;
                       dist(seq, 1, setA x seq);
                       [
                         proj2(seq, 1) ;
                         inc1(1, setA x (seq x seq))
                         ,
                         < proj2(seq, (setA x seq)) ,
                           proj1(seq, (setA x seq)) > ;
                         inc2(1, setA x seq x seq)
                       ]
                       ,
                       ( proj2(seq, seq) x id(setA x seq) ) ;
                       < proj2(seq, setA x seq) ,
                         proj1(seq, setA x seq) > ;
                       inc2(1, setA x seq x seq)
                     ] .

  eq  conc ; next  =  concstruct ; seqfunc(conc) .

endfm
```

Module 17: Concatenation of Sequences

```
fmod INCSTR is

  including SEQUENCES .
  including STREAMS .

  op incstr : -> Arrow .

  eq  dom(incstr)  =  streams .
  eq  cod(incstr)  =  seq .

  eq  incstr ; next  = < head, tail > ; inc2(1,setA x streams) ;
                       seqfunc(incstr) .
endfm
```

Module 18: Inclusion of Streams into Sequences