DIAGRAMMES

JOHN W. GRAY

Executable specifications for data-type constructors

Diagrammes, tome 24 (1990), p. 7-31

<http://www.numdam.org/item?id=DIA_1990__24__7_0>

© Université Paris 7, UER math., 1990, tous droits réservés.

L'accès aux archives de la revue « Diagrammes » implique l'accord avec les conditions générales d'utilisation (http://www.numdam.org/legal.php). Toute utilisation commerciale ou impression systématique est constitutive d'une infraction pénale. Toute copie ou impression de ce fichier doit contenir la présente mention de copyright.

NUMDAM

Article numérisé dans le cadre du programme
Numérisation de documents anciens mathématiques
http://www.numdam.org/

DIAGRAMMES VOLUME 24 . 1990

ACTES DES JOURNEES E.L.I.T.

(UNIV. PARIS 7, 27 JUIN-2 JUILLET 1988)

Executable Specifications for Data-Type Constructors

by John W. Gray
University of Illinois at Urbana-Champaign

Abstract:
The theory of sketches is discussed briefly and the example of natural numbers is treated in detail, both in a simple and a more complicated version. Then the functor Setof(-) is constructed by defining functions which return lists of rules depending on the parameter sketch that is substituted in the slot position. The functor Setof(-) is applied to the sketch NAT, yielding a large sketch with many properties. Finally, the functor Setof(-) is applied twice to NAT, resulting in a very large sketch. These constructions are implemented in the symbolic program *Mathematica*. Appendix A shows a number of examples of the code and Appendix B is a complete listing of the program.

1. Introduction.

A *sketch* is a directed graph in which certain nodes (or objects) are declared to be formal limits (e.g., products, equalizers, and pullbacks) of other objects, and certain diagrams made up of edges (or arrows) of the graph are declared to be formally commutative. See [1], [2], [5], [13], and [18] for detailed descriptions of the theory of sketches. Sketches are alternatives to algebraic specifications as presentations of algebraic theories and data types. The objects of the graph correspond to sorts and products of sorts in a specification and the arrows correspond to operations and compositions of operations. The formal commutative diagrams correspond to equations. A model of a sketch is a function ("functor") M that assigns a set to each object and a function to each arrow in such a way that formal limits are taken to actual limits and formal commutative diagrams are taken to actual commutative diagrams.

Sketches are much more tractable from a categorical point of view than specifications. In [13], I discussed a functorial construction A†of B where A† is a sketch with additional structure and B is an ordinary sketch. StacksOfChar and SetsOfBoole are examples. The purpose of this paper is to show that this construction is computationally feasible. This means first of all that a direct description of a sketch can be given in *Mathematica* in such a way that *Mathematica* will then carry out computations in the initial algebra for the sketch; i.e., it will reduce terms in the initial algebra to normal form. Furthermore, endofunctors on sketches of the form A†of(-) can be constructed which accept *Mathematica* sketches as input and produce *Mathematica* sketches as output.

reçu le 09/07/1990

The construction AtofB is considerably more powerful that the parametric specifications found in [6] or [7] which are simple pushouts of pairs of sketches along common sub-sketches. If one thinks of the two sketches in such a pushout as coordinate axes, then the construction AtofB fills in a whole two dimensional rectangle. Every operation of one sketch operates on every sort of the other sketch. The effect is that very large, correct sketches can be constructed from small inputs; i.e., large data types can be constructed from small components.

2. Note about *Mathematica*.

This document was produced in ***Mathematica*** and is a printout of a Macintosh notebook in that language. From our point of view, *Mathematica* is a rewrite-rule language and we make extensive use of these facilities. Bold face Courier (fixed width) font is used for inputs and the outputs from the program are in plain face Courier font. Capitalized terms in input expressions are built-in functions. All other expressions are defined in this notebook. The square boxes in the left margin indicate levels of outlining. Normally, the document is closed up into the top level except for the cell being examined. Here, of course, it is entirely printed out.

2.1. Three kinds of rewrite rules.

Rules of the form expr1 = expr2 evaluate the right hand side immediately, and then are automatically applied whenever possible to rewrite the left hand side as the right hand side. Rules of the form expr1 := expr2 evaluate the right hand side only when they are applied. They are always used here for "function" definition, in the form f[x_] := expr. Here x_ means a slot named x to be matched. Whatever matches x is then substituted for all occurrences of x in the right hand side. Rules of the form

```
expr1 := expr2 /; expr3
```

are conditional rewrite rules which are only applied when expr3 evaluates to True. All user defined rules given by = rules or := rules are added to the global list of rewrite rules maintained by *Mathematica* .

2.2 Pattern matching

The pattern matching in expressions like f[x_] can be controlled by adding expressions after the _. There are three forms for this. In an expression like

```
nat[n_Integer]:= True,
```

"Integer" is a built-in expression head. Any built-in or user-defined head can occur in this position and then n_ will match only expressions with this head. In an expression like

```
nat[s[n_?nat]]:= True,
```

nat is a user-defined predicate. Any predicate can occur in this position after "?" and then n_ will match only those expressions for which the predicate evaluates to true. Finally, the two forms can be combined in the form f[x_head?predicate]. This form is not used here.

Rules of the form:

```
setinsertrules[obj_]  :=  (
   insert[obj][
       r[n_?obj,insert[obj][r[m_?obj  ,p_?(P[obj])]]]
                 ] :=
        insert[obj][r[m,insert[obj][r[n, p]]]]/;
                                          !OrderedQ[{n,m}],

   insert[obj][
       r[n_?obj,insert[obj][r[n_?obj,p_?(P[obj])]]]
                  ] := insert[obj][r[n,p]]
                                )
```

play an important role in this program. When this expression is evaluated with some argument for obj, then as a side effect, it adds the rules in the list (indicated by brackets {, }) on the right hand side to the global list of rewrite rules. There are two rules in this list, parameterized by obj, defining the function "insert[obj]"; i.e., insert is a function whose values are again functions. For example, when obj ranges over the objects of Setof(nat), then the program produces a total of 32 rules for "insert" .

■ 2.3 Substitution

Substitution in an expression is indicated by "/." and ->. E. g.

```
%  /.  n[1]  ->  s[s[zero]]
```

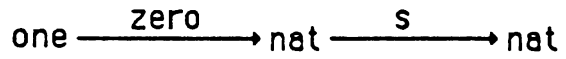
means substitute s[s[zero]] for n[1] in %; i.e., in the previous expression.

■ 3. The Natural Numbers as a Sketch

■ 3.1 The sketch for natural numbers

□ 3.1.1 Dedekind-Lawvere natural numbers

Natural numbers are defined by a sketch. A very simple sketch for the *Dedekind-Lawvere natural numbers* is illustrated as follows:



In this graph, "one" means that the object one has been declared to be the empty formal product; i.e. the symbolic terminal object. Thus "zero" denotes a constant and s denotes an endomorphism of nat. There are no other formal products and no formal commutative diagrams.

A *model* of a sketch assigns a set to each node of the graph and a function with the indicated domain and codomain to each arrow. Formal limits in the graph are taken to actual limits of sets and diagrams declared to be formally commutative are taken to actual commutative diagrams of functions. The *initial object* in the category of all models of the sketch is realized by the *term model* whose value for each node consists of all closed terms with codomain that node. For the Dedekind-Lawvere natural numbers, the terms with codomain nat are "zero" and "s(s(. . . s(zero). . .))" for any positive number of occurrences of "s". Hence the initial model for this sketch is isomorphic to the usual natural numbers.

In general, the term model is constructed by a categorical completion process. (See [1] or [16].) Starting from a directed graph, one constructs the free category on the graph. Its objects are the objects of the graph and its arrows are paths of arrows in the graph. Composition is just juxtaposition of paths. Now, if some object is supposed to be a product of two others, then each pair of morphisms into the factors must determine a (unique) morphism into the product. Hence, more arrows have to be added to the free category. But then there may be more composable paths which also have to be added, so the process has to be iterated, perhaps transfinitely often. If the original graph is finite, with only finite formal limits, then there is a countable construction which is finite at every stage for the free category with finite limits generated by a given graph with formal limits. Finally, the formally commutative diagrams determine a congruence relation on the arrows in this free category with finite limits. The corresponding quotient category is called the *theory* of the sketch, denoted by T(A). For any object a of A, and hence of T(A), the set T(A)(one, a) of morphisms in T(A) from one to a, is the set of closed terms of sort a.

Here is a cell which implements the Dedekind-Lawvere natural numbers. Everything is preceded by a "dl" to distinguish it from the Peano natural numbers implemented below. There isn't much that can be done with this cell, but it can be used in the later parameterized sketches to test out how they work with very simple input.

```
  objects[dlnat]       = {dlone, dlnat}
arrows[dlnat]          = {dlzero, dls}

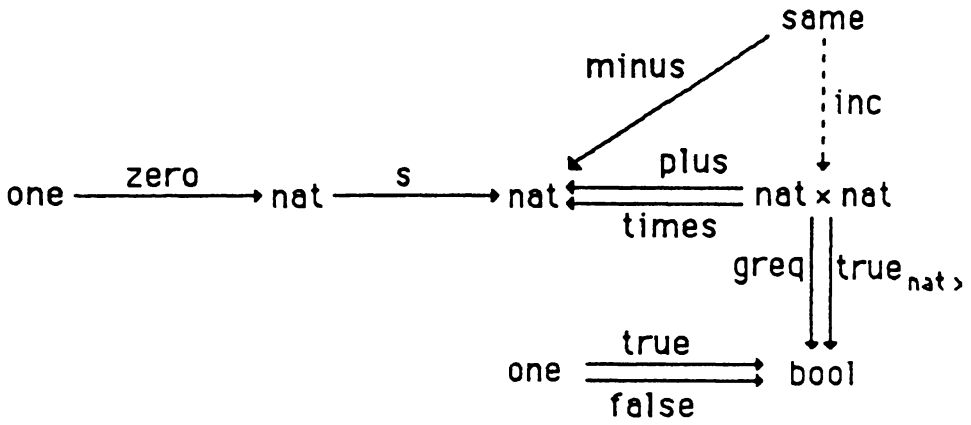
domain[dlzero]         = dlone
domain[dls]            = dlnat
codomain[dlzero]       = dlnat
codomain[dls]          = dlnat

dlone[oo]              = True
 dlone[_]              := False
dlnat[zero]            = True
dlnat[s[n_?nat]]       := True
dlnat[_]               := False
```

The first two lines describe the objects and arrows of the graph and the next four lines the beginning (domain) and end (codomain) of each arrow. The last five lines implement the term model for this sketch. The first two of them say that the only element of one is "oo" and the last three say that the elements of nat are are built up from zero by prepending "s" to already existing elements. (See the next section for a more complete description.)

□ 3.1.2 Peano natural numbers

In this notebook we will implement a more elaborate sketch for the natural numbers, which we will call the *Peano natural numbers*. It's sketch (in principle) looks as follows:



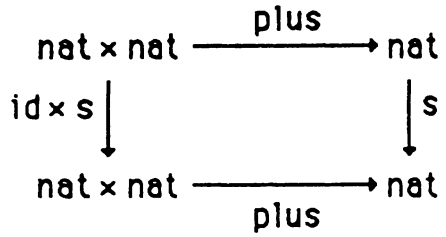
Here the node named "nat x nat" is (by virtue of its name) declared to be the formal product of nat with itself and the node named "same" is declared to be the formal equalizer of the two arrows "greq" and "true". The node "bool" will become the two element boolean algebra of truth values. The node "one" is repeated to make the sketch easier to draw. The arrow "inc" is dashed because it is not implemented here, except as a subtype. The arrows "plus", "times", and "minus" will become the usual functions with those names, except that minus(n, m) will not be reduced if m > n. The arrow "greq" will become the predicate "greater than or equal to".

There are two practical considerations in implementing the term model for this sketch. First of all, the terms will in fact be *Mathematica* expressions, so we have to pick out a subset of such expressions for each object in the graph. Therefore, our sets of terms in the term model should be regarded as Von Neumann sets; i.e., predicates on the universe of all *Mathematica* expressions. For instance, there will be a predicate "nat" as in the previous section which takes the value True for the *Mathematica* expressions "zero" and "s[n]" providing nat[n] = True. This last condition is enforced by giving the rule

```
nat[s[n_?nat]] = True.
```

We think of pattern matching expressions of the form "x_?pred" as having the same force as type assertions "x : type" in other languages. We sometimes even use this terminology when it seems appropriate.

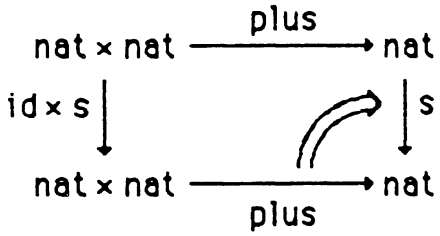
The second, even more significant consideration is that there is no way to implement the equations implied by asserting that certain diagrams commute. For instance, one equation that has to be implemented says that the diagram



commutes; i.e., that the equation plus[n,s[m]] = s[plus[n, m]] holds for all n and m. (Note that not all of the arrows for this diagram have been indicated in the sketch. The actual sketch for the Peano natural numbers is much larger than the part that is drawn here.) We don't know how to impose this equation, since imposing it would amount to forming equivalence classes for the congruence relation it generates on terms. Instead, we implement it also as a rewrite rule in the form:

plus[n_?nat,s[m_?nat]] := s[plus[n, m]]

This has the effect of choosing canonical members of congruence classes. Such a rewrite rule can be represented in the sketch by decorating the diagram with a 2-cell indicating the direction in which this rewrite rule is to be applied.



The actual sketch for the Peano natural numbers contains 14 such 2-cells. These 2-cells are all accounted for in the implementation. The interpretation of the term model is now also somewhat different. What one actually has in the implementation is a "signature", or a "free sketch", by which we mean a directed graph with symbolic limits, but no equations. Instead, it has 2-cells in certain diagrams. Such a structure is called an *order-enriched sketch*. The rewrite rules described by such 2-cells determine rewrite rules for the terms of this signature. The sets of normal forms with respect to these rewrite rules then constitute the initial algebra for our original sketch.

■ 3.2 Description of the implementation of the natural numbers.

See the section entitled "The natural numbers" in Appendix B.

□ 3.2.1 The polymorphic identity and the polymorphic product structure.

A *polymorphic identity operation* is defined by the requirements that for any argument type, the expression "id[x_?arg]" has the same type as x and equals x; i.e.,

```
arg[id[x_?arg] := True
id[x_?arg]     := x
```

To use this for nat, for instance, it is necessary to evaluate the command "setupid[nat]". However, no type arguments are necessary to use "id" once it has been set up.

Similarly, there is a *polymorphic product construction* that constructs a product arg1 × arg 2 for any pair of types arg1 and arg2. Its elements are records of the form r[x,y] where x has type arg1 and y has type arg2; i.e., pairs are represented by the notation r[a,b]. "r" is either the last letter of "pair" or the first letter of "record". First and second projection functions, p1 and p2 are defined and return x and y respectively. In conformance with the philosophy of sketches rather than categories, all possible products are not constructed. Only those particular products that are actually used are constructed. Here, the only such product is "nat × nat". The product constructor is a function X[arg1, arg2] which is frequently written with infix notation (arg1~X~arg2). (The parentheses are necessary.) One way to automatically construct the products of all the basic (i.e., non-product) objects would be to use the command

```
Do[Thread[setupprod, objectlist[[i]], objectlist], {i, 1, Length[objectlist]}]
```

Finally, the corresponding construction for products of arrows is implemented, although it is not used in this sketch.

□ 3.2.2 The underlying graph.

We first construct the underlying directed graph of the sketch for the Peano natural numbers by implementing the following definitions.

```
objects[NAT] = {one, nat, nat~X~nat, bool, same}
arrows[NAT]  = {oo, True, False, zero, s, plus, times, greq, minus}

domain[oo]     = one          codomain[oo]     = one
domain[True]   = one          codomain[True]   = bool
domain[False]  = one          codomain[False]  = bool
domain[zero]   = one          codomain[zero]   = nat
domain[s]      = nat          codomain[s]      = nat
domain[plus]   = nat~X~nat    codomain[plus]   = nat
domain[times]  = nat~X~nat    codomain[times]  = nat
domain[greq]   = nat~X~nat    codomain[greq]   = bool
domain[minus]  = same         codomain[minus]  = nat
```

□ 3.2.3 Implementation of the objects.

The name of the set of terms of a given type is the same as the name of the type. The set denoted by one is the singleton set with one element, the element "oo"; i.e., oo is the only arrow from one to one - the only closed term of type one. The set denoted by nat is defined recursively to contain zero and the successor of any element; i.e., there are rules:

```
nat[zero] = True
nat[s[n_?nat]] := True.
```

Zero, s[zero], s[s[zero]], etc. are all (composed) arrows from one to nat so they are all closed terms of type nat. "Nat" serves both as the name for the domain and codomain of "s", and also as the predicate determining membership in the set of terms of type nat. The set denoted by nat~X~nat is constructed as the product of nat with itself. "Same" is implemented as a subtype of nat~X~nat.

□ 3.2.4 Implementation of the arrows.

The arrows with domains other than one are implemented by restricting their arguments to be of the proper type and asserting that their values satisfy the predicate for the corresponding codomain. Thus, for instance, we require that

```
nat[plus[r[n_?nat, m_?nat]]] = True,
```

except that this is written in the preferable form:

```
nat[plus[t_?(nat~X~nat)]] = True.
```

There are similar equations for times and greq. However, minus[n, m] has type nat; i.e., satisfies the predicate "nat", only if $n \geq m$. For anything else, it is unreduced.

Another view of the matter is that the expression

```
nat[plus[t_?(nat~X~nat)]] = True
```

says that "plus[t_?(nat~X~nat)]" is a term of type nat and hence it is of exactly the same character as the statements about nat in the preceding section. In other words, all the statements about nat are just characterizing the terms of type nat.

In the first view, plus, times and minus are functions whose values are specified by the usual recursion formulas. Thus, for instance, we have the two rules:

```
plus[r[n_?nat, zero]] := n
plus[r[n_?nat, s[m_?nat]]] := s[plus[r[n, m]]].
```

There are similar rules for times, greq, and minus, except that the recursive rule for minus in only applied if $n \geq m$; that is, if r[n,m] belongs to "same".

```
minus[r[s[n_?nat],s[m_?nat]]]:= minus[r[n,m]] /; same[r[n,m]].
```

The corresponding other view of these expressions is that plus, times and minus are term constructors (i.e., combinators) of appropriate types and the recursion formulas are the rewrite rules that determine the normal forms for the terms of various types. Much of the power of *Mathematica* as a language for programming abstract mathematics derives from this dual view of ":= rules" as on the one hand specifying function values and on the other as giving rewrite rules for expressions.

□ 3.2.5 Integer arithmetic.

For convenience in examples and for use in later parameterized sketches, there is an extra cell in Appendix B embedding ordinary arithmetic of integers into the Peano natural numbers.

□ 3.2.6 Variables.

It is possible to have non-closed terms involving variables. The corresponding cell in Appendix B constructs an infinite number of variables, called ni, which have type nat and so can be used in natural number expressions. They behave correctly with respect to substitution since evaluation in *Mathematica* is given by substitution.

■ 3.3 Syntax of nat.

A sketch, or an algebraic specification, can be viewed as a description of a small typed functional programming language in which the sorts and products of sorts are the types and the operations are the terms. Such a language can be described by a formal syntax as usual. The formal syntax of nat is quite simple. There are objects (or types) given by the grammar:

B :: one | nat | bool | same

T :: B | T × T | T → T

There are arrows (or terms) given by type assignments as in [22]. A *type assertion* is a statement of the form x : type. In our formulation, this is equivalent to either of the statements: x ∈ type, or type[x] = True. A *type assignment* Γ is a finite set of type assertions x : t , where x is a variable and t is a type, such that no variable appears twice. Write Γ, x : t for the type assignment with x : t added to Γ, where it is assumed that x does not appear in Γ. Terms are introduced by formulas Γ > m : t, which can be read as "m has type t relative to Γ."

The well-typed terms are generated freely by the following *basic axioms* and *axiom schemes* which are part of the axioms for the grammar of any sketch.

$$\emptyset > c : t \quad \text{where c is a constant of type t.}$$

$$\Gamma, x : t > x : t$$

$$\frac{\Gamma > n : t \to t',\ \Gamma > f : t' \to t''}{\Gamma > f[n] : t \to t''}$$

$$\frac{\Gamma > m : t \to t',\ \Gamma > n : t \to t''}{\Gamma > r[m, n] : t \to t' \times t''}$$

$$\frac{\Gamma > p : t \to t' \times t''}{\Gamma > p1[p] : t \to t'} \qquad \frac{\Gamma > p : t \to t' \times t''}{\Gamma > p2[p] : t \to t''}$$

There are 9 constants in the grammar of the sketch for nat:

oo : one → one, zero : one → nat, True : one → bool, False : one → bool,
s : nat → nat, plus : nat × nat → nat, times : nat × nat → nat,
minus : same → nat, greq : nat × nat → bool.

Nat does not have any additional axioms. Note that functional types are implemented here in the form: p : t → t' if and only if domain[p] = t and codomain[p] = t'.

Equations are written in the form Γ > a == b : t.

$$\frac{\Gamma > p : one \to nat \times nat, \quad \Gamma > greq[pr1[p], pr2[p]] == True : one \to bool}{\Gamma > p : one \to same}$$

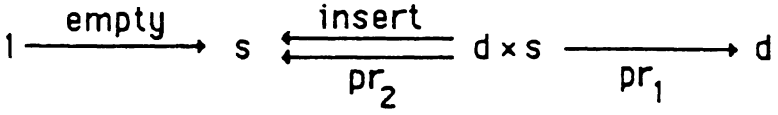
$$\frac{\Gamma > m : one \to nat, \quad \Gamma > n : one \to nat}{\Gamma > plus[m, s[n]] == s[plus[m, n]] : one \to nat}$$

etc.

4. SetOf(-) as a Constructor on Sketches

4.1 A non-parametric sketch for sets of data

Setofdata is a sketch whose underlying graph in a first approximation looks like:



See [13] for a description of the complete sketch. In what follows, "d" will be treated as a variable slot to be filled by other sketches. For the moment, consider a model M of this sketch with M[d] = D, where D is some unspecified set of data. Then M[s] = S is supposed to consist of finite subsets of D; i.e., elements of S are finite subsets of D. Empty is a constant of sort S and represents the empty set of data. Insert is a function from pairs consisting of an element d of S and a subset D' of D. Insert(d, D') then denotes the new subset of D in which d has been added to D'. Two equations should be satisfied by insert:

insert(d, insert(d, D')) = insert(d, D')
insert(d, insert(d', D')) = insert(d', insert(d, D')).

Thus, inserting an element twice is the same as doing it once, and the order in which elements are inserted doesn't matter. Thus, insert is an operation of D on the set S of subsets of D which is idempotent and commutative.

Here is a simple non-parametric version of sets of data in which any kind of data is allowed.

```
objects[setofdata] = {one, set, data}
arrows{setofdata]  = {empty, insert}
domain[empty]  = one; codomain[empty] = set
domain[insert] = data~X~set; codomain[insert]= set
```

```
one[oo] = True; one[_] := False
data[_]    := True
set[empty] = True
set[insert[r[n_?data, p_?set]]] := True

  insert[r[n_?data,  insert[
            r[m_?data,  p_?set]]]]:=
               insert[r[m,insert[r[n,p]]]]/;
                                    !OrderedQ[{n,m}],
  insert[r[n_?data,  insert[
           r[n_?data,  p_?set]]]]:= insert[n,p]
```

Note that the first rule for insert only works because there is a global total order for all *Mathematica* terms. Otherwise, this rule would just cycle infinitely often. The symbol ! is negation. This is required because OrderedQ[n, n] = True, which would prevent the second rule from ever being used.

■ 4.2 Description of the data-type constructor Setof(-)

Setof(-) is a functor from the category of order-enriched sketches to itself. See [13]. Given an ordered-enriched sketch A as input, it produces an order-enriched sketch SETOF(A) as output. Thus we have to define functions such that given A, they first produce the underlying graph of SETOF(A), and then produce the collection of needed rewrite rules for SETOF(A) from the rewrite rules of A.

We first construct the underlying directed graph function of setof(-) by implementing the following definitions. Here "arg_" represents any sketch and SETOF[arg_] represents the sketch to be constructed. Upper case "U" here means "union". From now on "P" is to be thought of as the covariant "finite powerset" functor. Its value on arrows is denoted by "Pm" (m for "morphism"). I.e., in the context of set theory, for any set Z, P(Z) is the set of finite subsets of Z, and for any function f, Pm(f) is the induced function (morphism) on subsets taking a subset D' to the set

$$Pm(f)(D') = \{f(d) \mid d \in D'\}.$$

See, for instance, the example under the Setof[NAT] heading in Appendix A.

objects[SETOF[arg_]] = objects[arg] ∪ P[objects[arg]] ∪ {arg x P[arg]}
arrows[SETOF[arg_]] = arrows[arg] ∪ P[arrows[arg]] ∪ {arrow x P[arrow]}
∪ empty[objects[arg]] ∪ insert[objects[arg]]

domain[empty[obj_]] = one
domain[insert[obj_]] = obj × P[obj]
domain[Pm[arr_]] = Pm[domain[arr]]
codomain[empty[obj_]] = P[obj]
codomain[insert[obj_]] = P[obj]
codomain[Pm[arr_]] = Pm[codomain[arr]]

We need predicates for each object of the argument saying that empty[obj] and insert[obj] have that type. In the alternative view, empty[obj] and insert[obj] are terms of type "P[obj]". Note that the only way to get an element of P[obj] is to take either empty[obj] or insert an element of the given type into a set of the given type. Empty[obj] is the only basic element. We define a function of one parameter whose value for each object is the list of these two rules.

```
predsetof[obj_] = {P[obj][empty[obj]] = True,
                   P[obj][insert[obj][n_?obj, p_?(P[obj])]] = True}
```

In the implementation in Appendix B, "predsetof" is given the attribute "Listable" so we can apply it to the list of objects of the argument sketch.

Next, the equations for insert have to be implemented as rewrite rules. Again this is done by a "Listable" function of one parameter whose value for each object is the list of the two rules for insert.

```
setinsertrules[obj_] :=
    {insert[obj][n_?obj, insert[obj][m_?obj ,p_?(P[obj])]] :=
        insert[obj][m,insert[obj][n, p]] "provided" !OrderedQ[{n,m}],
     insert[obj][n_?obj, insert[obj][n_?obj,p_?(P[obj])]] := insert[obj][n,p]}
```

As a new ingredient, we need the naturality rules that say empty and insert commute with the induced set operations in the argument type. These induced operations are denoted by Pm[arr], where "arr" is any arrow in the argument sketch. Such an operation following empty or insert is always rewritten in the opposite order. This is done by a "Listable" function of one parameter which is applied to the arrows of the argument sketch.

```
setoprules[arr_] = {Pm[arr][empty[domain[arr]]] = empty[codomain[arr]],
                    Pm[arr][insert[domain[arr]][n_?(domain[arr]),
                                                p_?(domain[Pm[arr]])]]
                                = insert[codomain[arr]][arr[n],Pm[arr][p]]}
```

These predicates and operator rules are given in functional form, so they need not be mentioned when they are used. The function setof[-] is then defined which applies these functions to an arbitrary sketch A. One can show that if the rewrite rules for A are Church-Rosser, then so are the rewrite rules for setof[A].

Last of all, there is included a formatting function which formats the output of all of the operations as ordered lists without repetition. Notice that Format is given in terms of a recursively defined function "format". Format itself cannot be specified recursively.

4.3 Syntax of Setof(-)

Setof[-] is given by a parameterized grammar. The types are given by:

T :: T[arg] | P[T[arg]] | T[arg] × P[T[arg]].

Here "arg" stands for any sketch, T[arg] means the set of types of arg, and P[T[arg]] means P applied to every type of arg. The terms are introduced by type assignments as before. The constants for Setof[arg] consist of the following families.

i) All constants of arg are constants of Setof[arg].

ii) If t is a type of arg, then

empty[t] : one → P[t]

insert[t] : t × P[t] → P[t]

are constants of Setof[arg].

In addition to the basic type assignment axioms, there are the following axiom schemes.

$$\frac{\Gamma > p : t \to t'}{\Gamma > Pm[p] : P[t] \to P[t']}$$

$$\frac{\Gamma > p : t \to t'}{\Gamma > p \times Pm[p] : t \times P[t] \to t' \times P[t']}$$

The equations of Setof[arg] consist of all equations of arg together with the following equation schemes.

$$\frac{\Gamma > p == p' : t}{\Gamma > Pm[p] == Pm[p'] : P[t]}$$

$$\frac{\Gamma > n : t \to t',\ \Gamma > f : t' \to t''}{\Gamma > P[f[n]] == P[f][P[n]] : P[t] \to P[t'']}$$

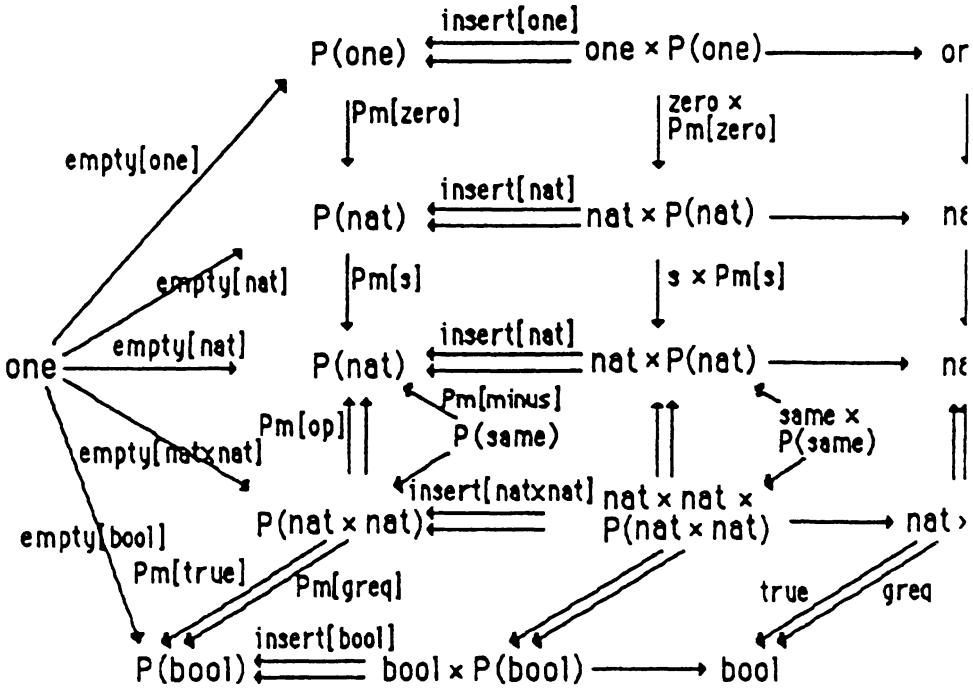
$$\frac{\Gamma > p : t \to t'}{\Gamma > Pm[p][empty[t]] == empty[t'] : one \to P[t]}$$

$$\frac{\Gamma > p : t \to t'}{\Gamma > Pm[p][insert[t] == insert[t'][p \times Pm[p]] : P[t \times P[t]] \to P[t']}$$

5. SetOf[NAT]

5.1 The sketch for SETOF(NAT)

The construction setof(-) is a functor from the category of sketches to itself. When it is applied to a sketch A, it produces a new sketch SETOF(A). Each object of A is replaced by a copy of setof data with the object in the place of "data". Each arrow of A determines new arrows between corresponding objects, as illustrated in the sketch SETOF(NAT)



In this drawing, the unlabeled arrows are either projections from products onto factors or inclusions of subobjects or arrows of the form arr x Pm[arr] which didn't fit in the picture. These latter arrows have not been implemented but the polymorphic code to do so is part of this Notebook. Also, the arrows "true" and "false" from one to bool, together with their associated arrows, are omitted for legibility. "Op" stands for "plus", "times", and any other binary operation that may have been implemented in NAT.

6. Setof[setof[NAT]]

6.1 The sketch for setof(setof(nat))

A drawing of the sketch for setof(setof(nat)), where nat is just the Dedekind- Lawvere can be found in [13]. A similar picture for the Peano natural numbers would be impossible to draw. However, its objects and arrows are calculated in the setof(setof(nat)) examples section in Appendix A. Note that the arrows of setof(setof(nat)) are the basic operations in this data type. This very large data type has been constructed by the program, not by us, from small components that are visibly correct by a method that is visibly correct. Hence, it is also correct.

■ 7. Appendix A: Examples

A data type presented by a sketch is a small functional programming language. These examples can be viewed either as illustrating some of the basic functions in these languages or as illustrating how the program reduces terms to normal form.

■ 7.1 Setof(NAT) Examples

□ 7.1.1 Objects and arrows for SETOF(NAT).

```
objects[SETOF[NAT]]
```

```
{ one, nat, X[nat, nat], bool, same, P[one], P[nat],
  P[X[nat, nat]], P[bool], P[same], X[one, P[one]],
  X[nat, P[nat]], X[X[nat, nat], P[X[nat, nat]]],
  X[bool, P[bool]], X[same, P[same]]}
```

```
arrows[SETOF[NAT]]
```

```
{ oo, zero, s, plus, times, greq, minus, True, False,
  Pm[oo], Pm[zero], Pm[s], Pm[plus], Pm[times],
  Pm[greq], Pm[minus], Pm[True], Pm[False],
  Xm[oo, Pm[oo]], Xm[zero, Pm[zero]], Xm[s, Pm[s]],
  Xm[plus, Pm[plus]], Xm[times, Pm[times]],
  Xm[greq, Pm[greq]], Xm[minus, Pm[minus]],
  Xm[True, Pm[True]], Xm[False, Pm[False]], empty[one],
  empty[nat], empty[X[nat, nat]], empty[bool],
  empty[same], insert[one], insert[nat],
  insert[X[nat, nat]], insert[bool], insert[same]}
```

□ 7.1.2 Try out Pm(s) and Pm(plus) for SETOF(NAT).

Here we check that Pm[operation] works correctly.

```
pairset  =  empty[nat~X~nat];
Do[pairset  =  insert[nat~X~nat][r[r[i,s[i]],
                                 pairset]],
     {i,8,0,-1}];
pairset
```

```
{r[0, 1], r[1, 2], r[2, 3], r[3, 4], r[4, 5], r[5, 6],

  r[6, 7], r[7, 8], r[8, 9]}
```

This output is a formatted set of pairs of natural numbers. Note that the inputs were given in reverse order, but the rules have put them in canonical order.

```
Pm[plus][pairset]
```

```
{1, 3, 5, 7, 9, 11, 13, 15, 17}
```

```
Pm[times][pairset]
```

```
{0, 2, 6, 12, 20, 30, 42, 56, 72}
```

```
Pm[s][Pm[plus][pairset]]
```

```
{2, 4, 6, 8, 10, 12, 14, 16, 18}
```

Thus, Pm[plus] applied to a set of pairs returns the set consisting of the sum of each pair. Next, we check how pairs are ordered and that Pm[greq] works correctly.

```
newpairset = insert[nat~X~nat][r[r[6,5],pairset]]
```

```
{r[0, 1], r[1, 2], r[2, 3], r[3, 4], r[4, 5], r[5, 6],

  r[6, 5], r[6, 7], r[7, 8], r[8, 9]}
```

Again, the canonical ordering has put r[6, 5] in the correct place.

```
Pm[greq][newpairset]
```

```
{False, True}
```

The output is the set of two truth values, not a list of 10 such values.

■ 7.2 Setof(setof(NAT)) Examples

□ 7.2.1 Objects and arrows of SETOF[SETOF[NAT]].

The output of the next two cells has been edited to save space and improve legibility.

```
objects[SETOF[SETOF[NAT]]]
```

```
{one, nat, X[nat, nat], bool, same,
P[one], P[nat], P[X[nat, nat]], P[bool], P[same],
X[one, P[one]], X[nat, P[nat]], X[X[nat, nat], P[X[nat, na
X[bool, P[bool]], X[same, P[same]],
P[P[one]], P[P[nat]], P[P[X[nat, nat]]], P[P[bool]], P[P[s
P[X[one, P[one]]], P[X[nat, P[nat]]], P[X[X[nat, nat],
P[X[nat, nat]]]], P[X[bool, P[bool]]], P[X[same, P[same]]]
X[P[one], P[P[one]]], X[P[nat], P[P[nat]]],
X[P[X[nat, nat]], P[P[X[nat, nat]]]], X[P[bool], P[P[bool]
X[P[same], P[P[same]]], X[X[one, P[one]], P[X[one, P[one]]
X[X[nat, P[nat]], P[X[nat, P[nat]]]],
X[X[X[nat, nat], P[X[nat,nat]]],P[X[X[nat, nat], P[X[nat,
X[X[bool, P[bool]], P[X[bool, P[bool]]]],
X[X[same, P[same]], P[X[same, P[same]]]]}
```

arrows[SETOF[SETOF[NAT]]]

{oo, zero, s, plus, times, greq, minus, True, False,
Pm[oo], Pm[zero], Pm[s], Pm[plus], Pm[times], Pm[greq],
Pm[minus], Pm[True], Pm[False],
Xm[oo, Pm[oo]], Xm[zero, Pm[zero]], Xm[s, Pm[s]], Xm[plus,
Xm[times, Pm[times]], Xm[greq, Pm[greq]], Xm[minus, Pm[min
Xm[True, Pm[True]], Xm[False, Pm[False]],
empty[one], empty[nat], empty[X[nat, nat]], empty[bool], e
insert[one], insert[nat], insert[X[nat, nat]],
insert[bool], insert[same],
Pm[Pm[oo]], Pm[Pm[zero]], Pm[Pm[s]], Pm[Pm[plus]], Pm[Pm[t
Pm[Pm[greq]], Pm[Pm[minus]], Pm[Pm[True]], Pm[Pm[False]],
Pm[Xm[oo, Pm[oo]]], Pm[Xm[zero, Pm[zero]]], Pm[Xm[s, Pm[s]
Pm[Xm[plus, Pm[plus]]], Pm[Xm[times, Pm[times]]],
Pm[Xm[greq, Pm[greq]]], Pm[Xm[minus, Pm[minus]]],
Pm[Xm[True, Pm[True]]], Pm[Xm[False, Pm[False]]],
Pm[empty[one]], Pm[empty[nat]], Pm[empty[X[nat, nat]]],
Pm[empty[bool]], Pm[empty[same]],
Pm[insert[one]], Pm[insert[nat]], Pm[insert[X[nat, nat]]],
Pm[insert[bool]], Pm[insert[same]],
Xm[Pm[oo], Pm[Pm[oo]]], Xm[Pm[zero], Pm[Pm[zero]]],
Xm[Pm[s], Pm[Pm[s]]], Xm[Pm[plus], Pm[Pm[plus]]],
Xm[Pm[times], Pm[Pm[times]]], Xm[Pm[greq], Pm[Pm[greq]]],
Xm[Pm[minus], Pm[Pm[minus]]], Xm[Pm[True], Pm[Pm[True]]],
Xm[Pm[False], Pm[Pm[False]]], Xm[Xm[oo, Pm[oo]], Pm[Xm[oo,
Xm[Xm[zero, Pm[zero]], Pm[Xm[zero, Pm[zero]]]],
Xm[Xm[s, Pm[s]], Pm[Xm[s, Pm[s]]]],

Xm[Xm[plus, Pm[plus]], Pm[Xm[plus, Pm[plus]]]],
Xm[Xm[times, Pm[times]], Pm[Xm[times, Pm[times]]]],
Xm[Xm[greq, Pm[greq]], Pm[Xm[greq, Pm[greq]]]],
Xm[Xm[minus, Pm[minus]], Pm[Xm[minus, Pm[minus]]]],
Xm[Xm[True, Pm[True]], Pm[Xm[True, Pm[True]]]],
Xm[Xm[False, Pm[False]], Pm[Xm[False, Pm[False]]]],
Xm[empty[one], Pm[empty[one]]],
Xm[empty[nat], Pm[empty[nat]]],
Xm[empty[X[nat, nat]], Pm[empty[X[nat, nat]]]],
Xm[empty[bool], Pm[empty[bool]]], Xm[empty[same], Pm[empty
Xm[insert[one], Pm[insert[one]]], Xm[insert[nat], Pm[inser
Xm[insert[X[nat, nat]], Pm[insert[X[nat, nat]]]],
Xm[insert[bool], Pm[insert[bool]]],
Xm[insert[same], Pm[insert[same]]],

```
empty[P[one]], empty[P[nat]], empty[P[X[nat, nat]]], empty
empty[P[same]], empty[X[one, P[one]]], empty[X[nat, P[nat]
empty[X[X[nat, nat], P[X[nat, nat]]]], empty[X[bool, P[boo
empty[X[same, P[same]]],

insert[P[one]], insert[P[nat]], insert[P[X[nat, nat]]],
insert[P[bool]], insert[P[same]], insert[X[one, P[one]]],
insert[X[nat, P[nat]]], insert[X[X[nat, nat], P[X[nat, nat
insert[X[bool, P[bool]]], insert[X[same, P[same]]]]
```

□ 7.2.2 Pm[Pm[operator]]

We construct a set of sets of pairs of natural numbers.

```
setsofpairs = empty[P[nat~X~nat]];
Do[setsofpairs = insert[P[nat~X~nat]][
     r[insert[nat~X~nat][
          r[r[4i+3,4i+4],
           insert[nat~X~nat][r[r[4i+1,4i+2],
                                empty[nat~X~nat]]]]],
       setsofpairs]],
   {i, 0, 3}];
setsofpairs

{{r[1, 2], r[3, 4]}, {r[5, 6], r[7, 8]},

  {r[9, 10], r[11, 12]}, {r[13, 14], r[15, 16]}}

Pm[Pm[plus]][setsofpairs]

{{3, 7}, {11, 15}, {19, 23}, {27, 31}}

Pm[Pm[times]][setsofpairs]

{{2, 12}, {30, 56}, {90, 132}, {182, 240}}
```

□ 7.2.3 Pm[insert[nat]]

We construct a set of pairs, each pair consisting of a natural number and a set of natural numbers. Then for each entry in the set, Pm[insert[nat]] will insert the natural number into the set of natural numbers.

```
amazing = empty[nat~X~P[nat]];
Do[amazing =
     insert[nat~X~P[nat]][r[r[3i+3,
               insert[nat][r[3i+2,
                    insert[nat][r[3i+1,empty[nat]]]]]],
          amazing ]], {i, 0, 3}];
amazing
```

```
{r[3, {1, 2}], r[6, {4, 5}], r[9, {7, 8}],

  r[12, {10, 11}]}

Pm[insert[nat]][amazing]

{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}}
```

□ 7.2.4 Programming example

We show how to use the language for SetOf[SetOf[-]] to write a simple program by constructing a function union[obj] that takes expressions of type P[P[obj]] to P[obj]. Elements of P[P[obj]] are families of sets of elements of obj. Union is to have its usual meaning for such a family of sets. Note that it is only necessary to give rules for union where the argument has the form empty[P[obj]] or the form insert[P[obj]][- - -] since the rewrite rules guarantee that every term of type P[P[obj]] has a normal form of one of these kinds. The operation of union does not exist at any lower level in the hierarchy of types. As a concept, it does not occur for obj or for SetOf[obj].

```
union[obj_][empty[P[obj_]]] = empty[obj]

union[obj_][insert[P[obj_]][r[empty[obj_],Q_]]]:=
     union[obj][Q]
union[obj_][insert[P[obj_]][
     r[p_, empty[P[obj_]]]]]:= p
union[obj_][insert[P[obj_]][
     r[insert[obj_][r[n_,p_]], Q_]]]:=
          insert[obj][r[n,
               union[obj][insert[P[obj]][r[p, Q]]]]]
```

To test this, consider an expression, threes, of type P[P[nat]].

```
triples = empty[nat~X~P[nat]];
Do[triples = insert[nat~X~P[nat]][r[r[3i+7,
               insert[nat][r[3i+4,
                    insert[nat][r[3i+1,empty[nat]]]]]],
          triples ]], {i, 1, 6}];
threes = Pm[insert[nat]][triples]

{{4, 7, 10}, {7, 10, 13}, {10, 13, 16}, {13, 16, 19},

  {16, 19, 22}, {19, 22, 25}}

union[nat][threes]

{4, 7, 10, 13, 16, 19, 22, 25}
```

■ 8. Appendix B: Implementation

■ 8.1 The natural numbers

□ 8.1.1 The polymorphic product structure

This constructs the polymorphic identity function

```
setupid[arg_]  :=
            {arg[id[x_?arg]]:= True,  id[x_?arg]  :=  x}
```

This constructs the product object arg1 x arg2 whose elements are records r[x,y] where x is of type arg1 and y is of type arg2.

```
setupprod[arg1_,arg2_] :=
        {X[arg1,arg2][r[x_?arg1,y_?arg2]]:= True,
         p1[r[x_?arg1,y_?arg2]] := x,
         p2[r[x_?arg1,y_?arg2]] := y }
```

This constructs a function f x g from arg1 x arg3 to arg2 x arg 4.

```
setupprodmap[arg1_,f_,arg2_,arg3_,g_,arg4_]:=
      {X[arg2,arg4][Xm[f,g][t_?(X[arg1,arg3])]]:=True
                              /;(arg2[f[x_?arg1]]==True &&
                                  arg4[g[y_?arg3]]==True),
       Xm[f,g][t_?(X[arg1,arg2])]  :=
                                r[f[p1[t]],g[p2[t]]] }
```

□ 8.1.2 The Peano natural numbers.

```
objects[NAT]=  {one,  nat,  (nat~X~nat),  bool,  same}
arrows[NAT]  =  {oo,  zero,  s,  plus,  times,  greq,
                    minus,  True,  False}

domain[oo]          =  one;  codomain[oo]       =  one
domain[True]        =  one;  codomain[True]     =  bool
domain[False]       =  one;  codomain[False]    =  bool
domain[zero]        =  one;  codomain[zero]     =  nat
domain[s]           =  nat;  codomain[s]        =  nat
domain[plus]        =  nat~X~nat;  codomain[plus]   =  nat
domain[times]       =  nat~X~nat;  codomain[times]  =  nat
domain[greq]        =  nat~X~nat;  codomain[greq]   =  bool
domain[minus]       =  same;  codomain[minus]  =  nat
```

These functions implement the predicates for bool, one, nat, nat~X~nat, and same.

```
one[oo]                          = True
one[_]                          := False
bool[True]                       = True
bool[False]                      = True
bool[_]                         := False

nat[zero]                        = True
nat[s[n_?nat]]                  := True
nat[_]                           := False

setupprod[nat,nat]
same[t_?(nat~X~nat)]    := True  /;
                                   greq[r[p1[t],p2[t]]]
```

These functions implement plus, times and minus as functions from nat~X~nat to nat, and greq as a function from nat~X~nat to bool .

```
nat[plus[t_?(nat~X~nat)]]   :=  True
nat[times[t_?(nat~X~nat)]]:=  True
nat[minus[t_?same]]          :=  True
bool[greq[t_?(nat~X~nat)]]:=  True
```

The values of plus,times,greq, and minus are defined recursively, and exceptions are propagated.

```
plus[r[n_?nat,  zero]]          := n
plus[r[n_?nat,   s[m_?nat]]]   :=  s[plus[r[n,m]]]

times[r[n_?nat,  zero]]         := zero
times[r[n_?nat,   s[m_?nat]]]:=
                              plus[r[times[r[n,m]],n]]

greq[r[n_?nat,zero]]            := True
greq[r[zero,s[n_?nat]]]         := False
greq[r[s[n_?nat],s[m_?nat]]]:=  greq[r[n,m]]

minus[t_?same]:=  p1[t]/;  p2[t]  ==  zero
minus[r[s[n_?nat],s[m_?nat]]]:=
                              minus[r[n,m]]/;same[r[n,m]]
```

□ 8.1.3 Integer arithmetic

```
nat[n_Integer]   :=  True
s[n_Integer]     :=  n + 1
plus[r[n_Integer,  m_Integer]]          :=  n + m

times[r[n_Integer,  m_Integer]]        :=  n  m
greq[r[n_Integer,  m_Integer]]         :=  (n >= m)
minus[r[n_Integer,m_Integer]]:=n-m/;   greq[r[n,m]]
```

□ 8.1.4 Variables

This defines varnat as a subtype of nat and formats variables with subscripts.

```
nat[n_?varnat]              := True
varnat[n[i_Integer]]        := True
Format[n[i_Integer]]        := Subscripted[n[i]]
```

■ 8.2 Setof(-)

□ 8.2.1 Underlying graph of Setof[-].

```
Attributes[SETOF]  =  {Listable}
Attributes[P]  =  {Listable}

objects[SETOF[arg_]]  :=  Flatten[{
          objects[arg],
          P[objects[arg]],
          Thread[X[objects[arg],P[objects[arg]]]]  }]

Attributes[empty]       =  {Listable}
Attributes[insert]      =  {Listable}
Attributes[Pm]          =  {Listable}
arrows[SETOF[arg_]]  :=  Flatten[{
          arrows[arg],
          Pm[arrows[arg]],
            Thread[Xm[arrows[arg],Pm[arrows[arg]]]],
          empty[objects[arg]],
         insert[objects[arg]]  }]

domain[empty[obj_]]      :=  one
domain[insert[obj_]]     :=  X[obj,P[obj]]
domain[Pm[arrow_]]       :=  P[domain[arrow]]
codomain[empty[obj_]]  :=  P[obj]
codomain[insert[obj_]]:=  P[obj]
codomain[Pm[arrow_]]     :=  P[codomain[arrow]]

Attributes[diagprod]  =  {Listable}
diagprod[obj_]  :=  setupprod[obj,P[obj]]
```

□ 8.2.2 Predicates and rules for Setof[-].

```
Attributes[predsetof] = {Listable}
predsetof[obj_] := {
  P/: P[obj][empty[obj]] := True,
  P/: P[obj][insert[obj][r[n_?obj,
                           p_?(P[obj])]]]:= True  }
```

```
Attributes[setinsertrules]  =  {Listable}
setinsertrules[obj_]  :=  {
   insert[obj][r[n_?obj,
      insert[obj][r[m_?obj  ,p_?(P[obj])]]]]  :=
      insert[obj][r[m,insert[obj][r[n,  p]]]]/;
                                        !OrderedQ[{n,m}],
       insert[obj][r[n_?obj,
            insert[obj][r[n_?obj,p_?(P[obj])]]]]  :=
                 insert[obj][r[n,p]]   }
Attributes[setoprules]  =  {Listable}
setoprules[arr_]  :=  {
     Pm[arr][empty[domain[arr]]]  :=
              empty[codomain[arr]],
      Pm[arr][insert[domain[arr]][
              r[n_?(domain[arr]),
                 p_?(domain[Pm[arr]])]]]  :=
         insert[codomain[arr]][r[arr[n],Pm[arr][p]]]  }
```

□ 8.2.3 The function setof[-].

```
setof[arg_]  :=Flatten[{
            diagprod[objects[arg]],
           predsetof[objects[arg]],
           setoprules[arrows[arg]],
          setinsertrules[objects[arg]]    }]
```

□ 8.2.4 Formatting rules for Setof[-].

```
format[empty[obj_]]  :=  {}
format[insert[obj_][r[n_,p_]]]  :=
                        Prepend[format[p],n]
Format[empty[obj_]]  :=  format[empty[obj]]
Format[insert[obj_][r[n_,p_]]]  :=
                         format[insert[obj][r[n,p]]]
```

■ 8.3 Evaluation of Setof[Nat] and Setof[Setof[Nat]]

```
setof[NAT];  setof[SETOF[NAT]];
```

9. References

[1]: M. Barr and C. Wells, *Toposes,Triples and Theories*,Springer-Verlag, New York, 1985.

[2] : A. Bastiani and C. Ehresmann, Categories of sketched structures, *Cahiers de Top. et Géom. Diff.* 13(1972), 103 - 214.

[3] : J. Benabou, Structures algébrique dans les catégories, *Cahiers de Top. et Géom. Diff.* 10(1968), 1 - 126.

[4]: R. Burstall and D. Rydeheard, *Computational Category Theory*, preprint, 1985.

[5]: L. Coppey and C. Lair, Leçons de Théorie des Esquisses (I), *Diagrammes*, Vol. 12, Paris, 1984.

[6]: H. Ehrig, H.-J. Kreowski, J. W. Thatcher, E. G. Wagner and J. B. Wright, Parameter passing in algebraic specification languages. *Theoretical Computer Science* 28(1984), 45 - 81.

[7]: H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications I. Equations and Initial Semantics*, EATCS Monographs 6, Springer- Verlag, New York 1985.

[8]: A. C. Ehresmann, Synopsis and comments, in : Charles Ehresmann, *Oeuvres Completes et Commentees, Part IV - 1*, Supp. 1, Vol. XXII (1981), and *Part IV - 2*, Supp. 2, Vol. XXIII (1982) to *Cahiers de Top. et Géom. Diff.*

[9]: P. Freyd, Algebra valued functors in general and tensor products in particular. *Colloq. Math.* 14(1966), 89 - 106.

[10]: J. Goguen, Parameterized programming, *IEEE Transactions on Software Engineering*, Vol. SE-10, No.5, 1984, 528 - 543.

[11]: J. Goguen and J. Meseguer, Initiality, induction, and computability, in *Algebraic Methods in Semantics*, M. Nivat and J. C. Reynolds (eds), Cambridge University Press, 1985.

[12]: J. W. Gray, *Formal Category Theory: Adjointness for 2-Categories*, Lecture Notes in Mathematics 391, Springer Verlag, New York, 1974.

[13]: J. W. Gray, Categorical aspects of data-type constructors, *Theoretical Computer Science*,50 (1987), 103 - 135

[14]: J. W. Gray, The category of sketches as a model for algebraic semantics, in *Categories in Computer Science and Logic*, Contemporary Mathematics 92, Amer. Math. Soc. 1989, 109 - 135.

[15]: G. M. Kelly, Structures defined by finite limits in the enriched context, I, *Cahiers de Top. et Géom. Diff.* 23(1982), 3 - 42.

[16]: G. M. Kelly, On the essentially-algebraic theory generated by a sketch. *Bull. Australian Math. Soc.* 26(1982), 45 - 56.

[17]: G. M. Kelly and R. Street, Lecture Notes in Mathematics 420, Springer-Verlag, New York, 1974.

[18]: C. Lair, Etude Générale de la Catégorie des Esquisses, *Esquisses Mathematiques* 23, Paris 1975, 1 - 62.

[19]: J. Lambek, and P. J. Scott, *Introduction to Higher Order Categorical Logic*, Cambridge Univ. Press, 1986.

[20]: F. W. Lawvere, Functorial semantics of algebraic theories, *Proc. Nat. Acad. Sci.* 50 (1963), 869 - 872.

[21]: S. Mac Lane, *Categories for the Working Mathematician*, Springer Verlag, New York, 1972.
[22]: J. C. Mitchell and P. J. Scott, Typed lambda models and cartesian closed categories, in *Categories in Computer Science and Logic*, Contemporary Mathematics 92, Amer. Math. Soc. 1989, 301 - 316.
[23]: J. W. Thatcher, E. G. Wagner and J. B. Wright, Data type specifications: Parameterizations and the power of specification techniques, *ACM Trans. on Programming Languages and Systems*, 4(1982), 711 - 732.