Algebras, Categories and Programming Languages

Eric G. Wagner
Wagner Mathematics
Old Albany Post Road
Garrison, NY 10524/USA
ericwagner@aol.com

April 27, 1994

Contents

1 Introduction

In this series of lectures we will explore the application of category theory (and universal algebra) to problems in computer science. In as much as possible we want to put the emphasis on the computer science in the sense that we want to show

category theory provides tools for computer science

rather than

computer science provides examples of categorical constructs.

My view is that category theory is useful to computer scientists looking for mathematical understanding of computer science constructs because it provides a clean treatment of the needed mathematical tools and makes it possible to

- avoid repeated invention of the wheel.
- reduce the proliferation of constructs.
- reduce the reproving of similar results in diverse areas.
- gain insight by knowing what kind of structure to look for.

Possible topics

- Lawvere Algebraic theories:
  - the basic definition and examples
  - relation to the work of Walters et al
  - some alternative (generally equivalent) definitions
  - the M-construction (Kleisli categories)
  -
- Indexed categories and fibrations:
  - the basic definitions
  - applications to algebras and algebraic specifications
  - applications to pointers and $(+, \times)$-recursive types
  -
- Monads:
  We use P.D. Wadler's paper, *Comprehending monads*, MSCS 2, 4 (1992), as our starting point and source of motivation.
- Cartesian Closed Categories:
- Toposes:
- Sketches:

1.1 About Category Theory

In [4] the ADJ group presented a sequence of four guidelines aimed at providing an initial intuition of the basic categorical concepts. They are worth repeating here for the same reasons.

1. Any species of mathematical structure is represented by a *category*, whose *objects* "are of that structure", and whose *morphisms* "preserve" it. (Note, we do not assert the converse.)
2. Any mathematical scheme for constructing objects of one type from objects of another type as data, is represented by a *functor* between the corresponding categories.
3. Any natural translation from one construction to another, between the same categories, is represented by a *natural transformation* between the corresponding functors.
4. Any natural (canonical) construction is represented by an *adjoint functor*.

The following seem worth adding, though they may not really deserve the honor of being called guidelines.

7. Limits and Colimits comprise a particular broad class of canonical constructions found frequently, though not always, in interesting categories.
8. Many important mathematical structures correspond to categories in which all, or most, of the structure is in the morphisms rather than in the objects. Indeed many of the categories refered to in the first guideline have objects of precisely this type, and that is one reason that we do not assert the converse of guideline 1.

Another set of guidelines (slogans) can be found in Lambek and Scott [9].

1.2 About Category Theory and Computer Science

Why is category theory relevant to computer science? Since I have been promoting the use of categorical concepts in computer science for at least fifteen years I obviously think that the categorical approach is a good thing. But this does not mean that it is a good for all computer scientists. Nor, for that matter, does it mean that a "true category theorist" would say that I have really been using category theory. Be that as it may, I will briefly spell out five reasons I have found for taking a categorical approach in theoretical computer science. These might be titled: abstraction, guidance, prescription, cleanliness and necessity.

Abstraction Repeatedly we find general concepts in computer science that are informally defined in terms of specific examples, representations, or implementations. Category theory provides a framework for giving formal definitions of these concepts that are representation, and implementation independent – that is, abstract. Category theory based definitions are naturally defined up-to-isomorphism, while, in contrast, set theoretic definitions generally employ specific representations, and other *ad hoc* detail. This kind of categorical abstraction is the basis for the algebraic treatments of abstract data types. The use of initiality in [5] to provide semantics for data type specifications may appear trivial from a mathematical viewpoint but it provides exactly the level of abstraction needed to capture what the data type does without saying how it does it.

Terminal algebra semantics for data types, as found in Wand [21], again employs a simple categorical concept to capture another abstraction. The use of pushouts in my paper [20] provides a way to "create pointers" without having to introduce an *ad hoc* set theoretic model of how the "next pointer" is chosen.

Guidance As Doctrine 1 suggests, if you have computer science structure then there should be a corresponding category of such structures. It is worthwhile trying to spell out the details of this category. The description of the category may not be a publishable result, but if the category does not exist it is very likely that you have the wrong model. I had this problem in my work on recursive definitions, the model given of a recursive definition in [16] does not yield a category of definitions in the sense that while the definitions could be composed the composition was not associative. This deficiency stimulated the eventual development of the superior treatment given in [22]. On a somewhat higher level we note that "Nice" categories have a lot of structure (limits, colimits, factorizations, etc.). Looking for and at this structure generally produces worthwhile results. On the one hand these structures are likely to be meaningful, and, on the other hand, a formulation of some computer construct in terms of such structure is likely to be correct. In short, categorical concepts provide guidance.

Prescription In some cases I believe that categorical concepts not only guide one's efforts, they actually prescribe what one should do. An example of this would be the use of products and coproducts to describe, respectively, records and variants (constructs analogous to the variant records of PASCAL or the union types of ALGOL60). The categorical description is not only short and precise, but it automatically provides the appropriate put and fetch operations and neatly avoids the hazards built into the informal PASCAL concept of a variant. In short, the categorical approach prescribes how the concepts should be formulated.

Cleanliness The categorical framework provides a very clean framework for dealing abstractly with matters that could be very messy when approached combinatorially. I was originally attracted to category theory because Lawvere algebraic theories [10], as formulated in Eilenberg-Wright [2], provided a clean, rigorous, framework for manipulating sets of recursion equations. The category theory in that case ensured the meaningfulness of the notation, and, more important, made it possible to prove the desired results. The paper [17] contains a proof of the associativity, etc., of substitution that illustrates both cleanliness and abstraction.

Necessity There seem to be some situations where category theory provides the only reasonable framework for working on a problem. While some of the work that the ADJ group did on data types uses so little of the categorical framework that it can be presented in a manner that completely suppresses the category theory, I believe that categorical ideas provided the essential starting point. Areas where a category theory approach seems to me to be necessary include some of the work on the semantics of the polymorphic λ-calculus, and the work on institutions of Goguen and Burstall [6].

2 Why Algebras

The goal of this section is to introduce, and provide computer science motivation for, the concept of a many-sorted algebra.

2.1 Some Notation, etc.

Let $\omega = \{0, 1, 2, \ldots\}$ denote the set of *natural numbers*. For $n \in \omega$ let $[n] = \{1, \ldots, n\}$. So $[0] = \emptyset$.

Definition 2.1 Let S be a set, then a *string u on S of length n* is a mapping $u : [n] \to S$. Let S^n be the set of all strings on S of length n, let $S^* = \bigcup\langle S^n \mid n = 0, 1, 2, \ldots\rangle$. Let $\lambda : [0] \to S$ be the unique mapping from $\emptyset$ to S, call λ *the empty string*. If $u \in S^n$, and $i \in [n]$ then we often write u_i for $u(i)$, and we also write $u = u_1 \cdots u_n$.

Sometimes we want to denote strings over a set S, where the elements of S are strings, e.g., $S = \{\texttt{Bool}, \texttt{Int}\}$. In this case we will use "$\cdot$" as a separator. That is, we will write

$$\texttt{Bool} \cdot \texttt{Int} \cdot \texttt{Bool}$$

for the string

$$\begin{aligned} u : [3] \to S = \{&\texttt{Bool}, \texttt{Int}\} \\ 1 &\mapsto \texttt{Bool} \\ 2 &\mapsto \texttt{Int} \\ 3 &\mapsto \texttt{Bool} \end{aligned}$$

□

2.2 Algebras

Informally speaking, a many-sorted algebra is a collection of named sets, together with a collection of named operations (total functions) between those sets.

Definition 2.2 Let S be a set (of *sorts*).

An *S-sorted signature* is an $(S^* \times S)$-indexed family of sets

$$\Sigma = \langle \Sigma_{w,s} \mid w \in S^*, s \in S\rangle.$$

An element $\sigma \in \Sigma_{w,s}$ is an *operator symbol* of *rank* $\langle w, s\rangle$, *arity* w, and *sort* s.

Let Σ_1 and Σ_2 be S-sorted signature. Then an *S-sorted signature morphism* $\underline{f} : \Sigma_1 \to \Sigma_2$ is a triple $\langle \Sigma_1,\ \overline{f},\ \Sigma_2\rangle$ where $\overline{f}$ is an $(S^* \times S)$-indexed family of mappings, $\overline{f} = \langle\ f_{w,s} : (\Sigma_1)_{w,s} \to (\Sigma_2)_{w,s}\rangle$. □

Definition 2.3 For an S-sorted signature Σ, a *Σ-algebra* A consists of

- An S-indexed family of sets $U(A) = \langle A_s \mid s \in S\rangle$.
- A mapping $\sigma_A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ for each $w = s_1 \cdots s_n$ in S^*, $s \in S$ and $\sigma \in \Sigma_{w,s}$.

The set A_s is the *carrier of sort s*, σ_A is the *operation* in A named by the operator σ, if $\sigma \in \Sigma_{\lambda,s}$ then σ_A is said to be a *constant* of A. For $w = s_1 \cdots s_n \in S^*$ we will often write A^w for $A_{s_1} \times \ldots \times A_{s_n}$. □

Definition 2.4 For Σ-algebras A and B, a Σ-*homomorphism* $h : A \to B$ is a triple

$$\langle A,\ \langle h_s : A_s \to B_s \mid s \in S\rangle,\ B\rangle$$

such that, for every $\sigma \in \Sigma_{w,s}$ with $w = w_1 \cdots w_n$ and for all $\langle a_1, \ldots, a_n\rangle \in A^w$

$$h_s(\sigma_A(a_1, \ldots, a_n)) = \sigma_B(h_{w_1}(a_1), \ldots, h_{w_n}(a_n)).$$

We write $\mathbf{Alg}_\Sigma$ to denote the category of all Σ-algebras together with the Σ-homomorphisms between them. $\square$

Example 2.5 The collection of primitive (given) types and operations in a programming language form a many sorted algebra. For example, consider a language with types (sorts)

$$S = \{\texttt{Bool}, \texttt{Int}, \texttt{Charstr}\}$$

and operators

$$\Sigma_{\lambda,\texttt{Bool}} = \{\mathit{True}, \mathit{False}\}$$

$$\Sigma_{\texttt{Bool},\texttt{Bool}} = \{\neg\}$$

$$\Sigma_{\texttt{Bool}\cdot\texttt{Bool},\texttt{Bool}} = \{\wedge, \vee\}$$

$$\Sigma_{\lambda,\texttt{Int}} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$$

$$\Sigma_{\texttt{Int},\texttt{Int}} = \{\mathit{minus}\}$$

$$\Sigma_{\texttt{Int}\cdot\texttt{Int},\texttt{Int}} = \{+, *, -\}$$

$$\Sigma_{\texttt{Bool}\cdot\texttt{Int}\cdot\texttt{Int},\texttt{Int}} = \{\mathit{IfN}\}$$

$$\Sigma_{\texttt{Int}\cdot\texttt{Int},\texttt{Bool}} = \{\mathit{eq}, \leq\}$$

$$\Sigma_{\lambda,\texttt{Charstr}} = \{\texttt{A}, \texttt{B}, \texttt{C}, \ldots\}$$

$$\Sigma_{\texttt{Charstr}\cdot\texttt{Charstr},\texttt{Charstr}} = \{\mathit{Concat}\}$$

$$\Sigma_{\texttt{Charstr},\texttt{Int}} = \{\mathit{Length}\}$$

$$\Sigma_{\texttt{Charstr}\cdot\texttt{Charstr},\texttt{Bool}} = \{\mathit{eqcs}\}$$

$$\Sigma_{\texttt{Bool}\cdot\texttt{Charstr}\cdot\texttt{Charstr},\texttt{Charstr}} = \{\mathit{IfC}\}$$

This signature is presented pictorially in figure 1.

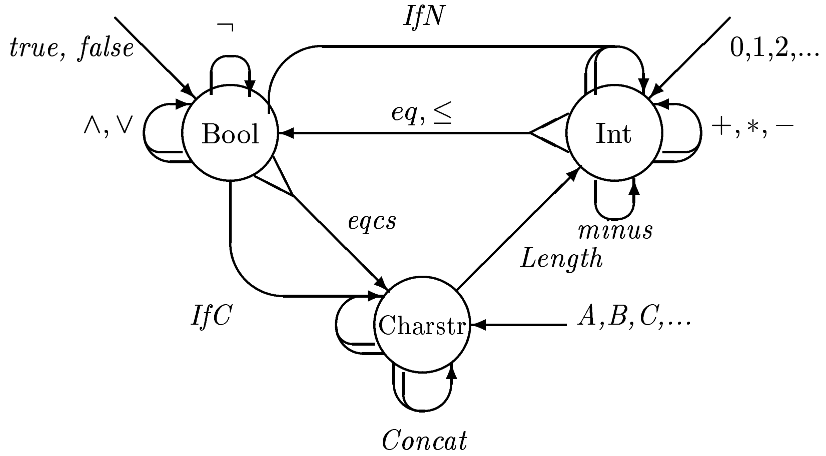


Figure 1: Picture of Signature

To make an algebra A with this signature we choose appropriate sets and operations.
Abstract Version

$A_{\texttt{Int}}$ = the set of integers, $\mathbf{Z}$.

$A_{\texttt{Bool}} = \{True, False\}$

$A_{\texttt{Charstr}} = \{\texttt{A}, \texttt{B}, \texttt{C}, \ldots, \texttt{Z}\}^*$ (i.e., the set of all strings on the alphabet)

Pragmatic Version:

$A_{\texttt{Int}}$ = all integers z with absolute value $\mid z \mid \leq 2^{32}$

$A_{\texttt{Bool}} = \{True, False\}$

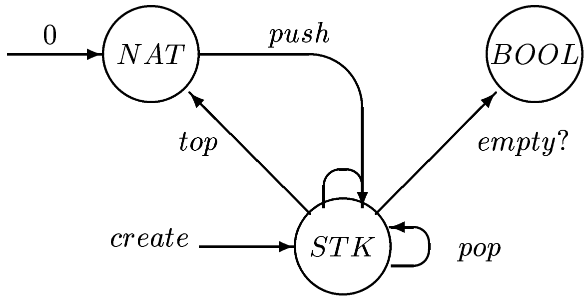
$A_{\texttt{Charstr}} = \{\texttt{A}, \texttt{B}, \texttt{C}, \ldots, \texttt{Z}\}^{256}$

with appropriate operations, e.g.,

integer addition, $+ : \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$
or
addition modulo 2^{32}

□

Example 2.6 Here is a second example of a data type signature, namely one containing the types *NAT*, *BOOL* and *STK*, the latter being intended to be interpreted as *STACKS-OF-NAT*.



One "reasonable" algebra to consider here would be the algebra A where

$$A_{NAT} = \omega = \{0, 1, 2, \ldots\}$$
$$A_{BOOL} = \{true,\ false\}$$
$$A_{STK} = \omega^*$$

and the stack operations, *push*, *pop*, *top*, *create*, and *empty?*, are defined as follows:

$$push : \omega \times \omega^* \to \omega^*$$
$$\langle n,\ s\rangle \mapsto n \cdot s$$

$$pop : \omega^* \to \omega^*$$
$$\lambda \mapsto \lambda$$
$$n \cdot s \mapsto s$$

$$top : \omega^* \to \omega$$
$$\lambda \mapsto 0$$
$$n \cdot s \mapsto n$$

$$create : \to \omega^*$$
$$(\text{i.e., } create \in \omega^*)$$
$$create = \lambda$$

$$empty? : \omega^* \to \{true,\ false\}$$
$$\lambda \mapsto true$$
$$n \cdot s \mapsto false.$$

This is not the only algebra that we could use for stacks of natural numbers. For example we could use an algebra B where:

$$B_{NAT} = \omega = A_{NAT}$$
$$B_{BOOL} = \{true,\ false\} = A_{BOOL}$$
$$B_{STK} = \{\langle s, n\rangle \in \omega^* \times \omega \mid 0 \le n \le |s|\}$$

and where:

$$push : \omega \times (\omega^* \times \omega) \to \omega^*$$
$$\langle p, \langle s,\ n\rangle\rangle \mapsto \langle s',\ n+1\rangle$$
$$\text{where } u_{n+1} = p \text{ and } u_i = s_i \text{ for all other } i$$

$$
\begin{aligned}
pop : \omega^* \times \omega &\to \omega^* \times \omega \\
\langle s, 0\rangle &\mapsto \langle s, 0\rangle \\
\langle s, n+1\rangle &\mapsto \langle s, n\rangle
\end{aligned}
$$

$$
\begin{aligned}
top : \omega^* \times \omega &\to \omega \\
\langle s, 0\rangle &\mapsto 0 \\
\langle s, n+1\rangle &\mapsto s_{n+1}
\end{aligned}
$$

$$
\begin{aligned}
create &:\to \omega^* \times \omega \\
create &= \langle \lambda, 0\rangle
\end{aligned}
$$

$$
\begin{aligned}
empty? : \omega^* \times \omega &\to \{true,\ false\} \\
\langle s, 0\rangle &\mapsto true \\
\langle s, n+1\rangle &\mapsto false.
\end{aligned}
$$

These two algebras are clearly different but they are related in the sense that there is a homomorphism $h : B \to A$, $h(\langle s, n\rangle) = s_1 \cdot s_2 \cdots s_n$. □

Questions:

1. How can we specify algebras such as those given above?
2. Can we specify data types such as STACKs, RECORDs, INTEGERs, SETs-of-INTEGERS, AIRLINE-RESERVATION, PAYROLL-RECORD, etc.

Example 2.7 What we want to show in this example is, first of all, that a context free grammar corresponds to a signature Σ, and then that the Σ-algebras provides a "semantics" for the language generated by the grammar. Of course, not all algebras correspond to "meaningful semantics", however the "meaningful semantics" do always correspond to algebras. We will illustrate this by means of a fairly complex example of a programming language. However, let us first review the basic definitions.

Definition 2.8 A *context free grammar*, G, consists of

N, a set of *non-terminals symbols.*

T, a set of *terminal symbols.*

P, a set of *productions*

$$P \subseteq N \times (N \cup T)^*$$

where we assume $N \cap T = \emptyset$.

$S \in N$, the *start symbol*

We often view a production p as a pair of strings

$$p = \langle A_0, \quad a_0 A_1 a_1 \cdots a_{n-1} A_n a_n \rangle$$

where $A_i \in N$, $a_i \in T^*$, and $n \geq 0$.

A grammar G defines a subset $L(G) \subseteq T^*$. We call $L(G)$ *the context free language generated by* G. The set $L(G)$ is defined in terms of the relation $\rightarrow_G$: $(N \cup T)^* \rightarrow (N \cup T)^*$, where

$$w \rightarrow_G u$$

iff there exists a production $p \in P$, $p = \langle L, R \rangle$, and $\alpha, \beta \in (N \cup T)^*$, such that $w = \alpha L \beta$ and $u = \alpha R \beta$. Let $\Rightarrow_G$ be the transitive closure of $\rightarrow_G$. Then

$$L(G) = \{u \in T^* \mid S \Rightarrow_G u\}$$

$\square$

Given a context free grammar G we can define a corresponding signature. The signature will have a sort for each non-terminal. Each production of the form

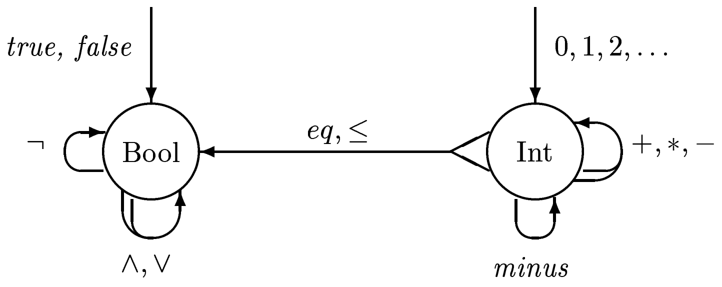
$$A_0 ::= a_0 A_1 a_1 \cdots a_{n-1} A_n a_n$$

(A_i a non-terminal symbol, a_i a string of terminal symbols) will correspond to an operator of rank $\langle A_1 \cdots A_n, A_0 \rangle$.

This provides an approach to programming language semantics. Here is an example using a simple programming language in which there are three non-terminals: statements ($\langle st \rangle$), arithmetic expressions ($\langle ae \rangle$), and boolean expressions ($\langle be \rangle$). We assume we are given an underlying signature Σ of arithmetic and boolean operations, and a set X of identifiers. We give each production a name, which will serve as the corresponding operator.

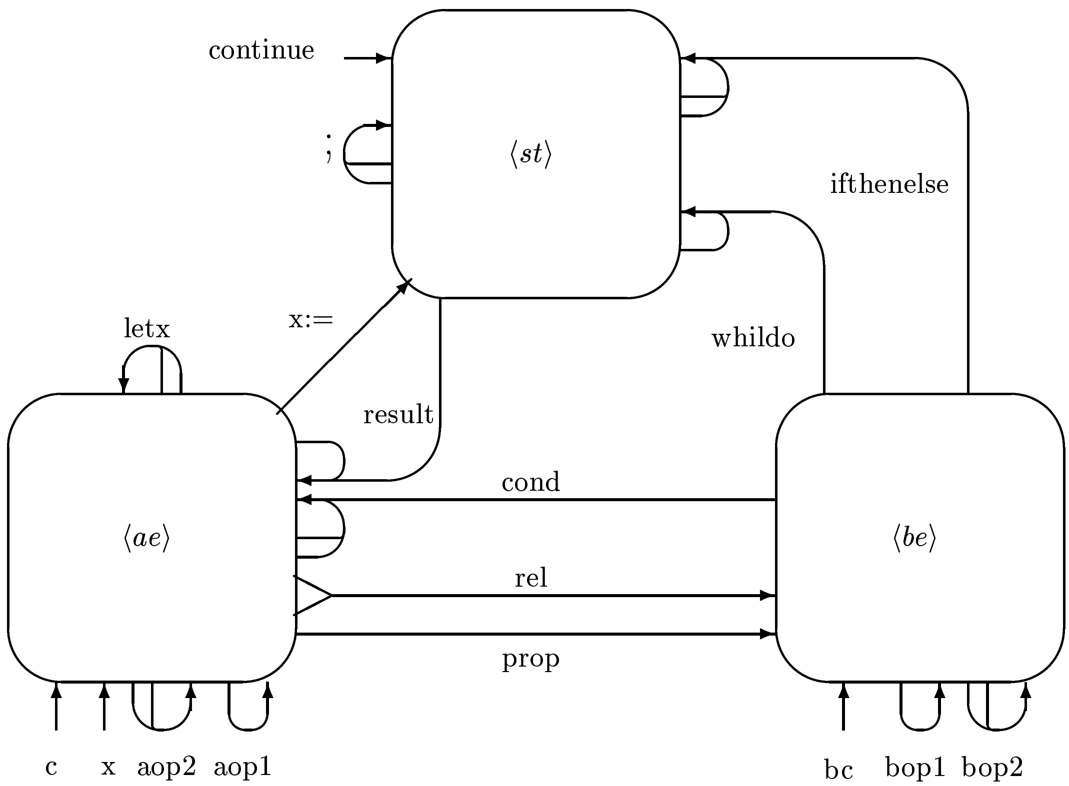
continue	$\langle st \rangle$ ::= **continue**	
x:=	$\langle st \rangle$ ::= **x:=** $\langle ae \rangle$	for $x \in X$
ifthenelse	$\langle st \rangle$ ::= **if** $\langle be \rangle$**then** $\langle ae \rangle$ **else** $\langle ae \rangle$	
;	$\langle st \rangle$ ::= $\langle st \rangle$; $\langle st \rangle$	
whildo	$\langle st \rangle$ ::= **while** $\langle be \rangle$**do** $\langle st \rangle$	
c	$\langle ae \rangle$ ::= **c**	for $c \in \Sigma_{\lambda,\texttt{Int}}$
x	$\langle ae \rangle$ ::= **x**	for $x \in X$
aop1	$\langle ae \rangle$ ::= **aop1**$\langle ae \rangle$	for aop1$\in \Sigma_{\texttt{Int},\texttt{Int}}$
aop2	$\langle ae \rangle$ ::= $\langle ae \rangle$**aop2** $\langle ae \rangle$	for aop2$\in \Sigma_{\texttt{Int.Int},\texttt{Int}}$
cond	$\langle ae \rangle$ ::= **if** $\langle be \rangle$**then** $\langle ae \rangle$**else** $\langle ae \rangle$	
result	$\langle ae \rangle$ ::= $\langle st \rangle$**result** $\langle ae \rangle$	
letx	$\langle ae \rangle$ ::= **letxbe** $\langle ae \rangle$**in** $\langle ae \rangle$	for $x \in X$
bc	$\langle be \rangle$ ::= **bc**	for bc$\in \Sigma_{\lambda,\texttt{Bool}}$
prop	$\langle be \rangle$ ::= **prop** $\langle ae \rangle$	for prop$\in \Sigma_{\texttt{Int},\texttt{Bool}}$
rel	$\langle be \rangle$ ::= $\langle ae \rangle$**rel** $\langle ae \rangle$	for rel$\in \Sigma_{\texttt{Int.Int},\texttt{Bool}}$
bop1	$\langle be \rangle$ ::= **bop1**$\langle be \rangle$	for bop1$\in \Sigma_{\texttt{Bool},\texttt{Bool}}$
bop2	$\langle be \rangle$ ::= $\langle be \rangle$**bop2**$\langle be \rangle$	for bop2$\in \Sigma_{\texttt{Bool.Bool},\texttt{Bool}}$

To make this more concrete let Σ be the following subsignature of the signature given in Example 2.5.



Then aop1 = *minus*, each aop2 $\in \{+, *, -\}$, bop1 = $\neg$, each bop2 $\in \{\wedge, \vee\}$, each rel $\in \{eq, \leq\}$.

The new signature corresponding to the above context free grammar has the following picture:



What do we want for the corresponding algebra? There are at least two answers. One providing a syntactic answer, one providing a semantic answer.
ANSWER ONE: The syntactic algebra, *Syn*. Given Σ as above, take the carrier of *Syn* of sort

$\langle st \rangle$ to be the set of all statements viewed as strings on the terminal alphabet (including Σ).

$\langle ae \rangle$ to be the set of all arithmetic expressions viewed as strings on the

$\langle st \rangle$ to be the set of all boolean expressions viewed as strings on the terminal alphabet (including Σ).

Then, for example, if $u, v, w, x, y, z \in X$, we have

$$0 \leq z$$

is the carrier $Syn_{\langle be \rangle}$

$$y * z \text{ and } z - 1$$

are in the carrier $Syn_{\langle ae \rangle}$

$$x := y * z\,\dot{;}\, z := z - 1$$

is in the carrier $Syn_{\langle st \rangle}$, and the operation corresponding to the operator symbol "whildo" is such that

$$whildo_{Syn}(0 \leq z,\ x := y * z\,\dot{;}\, z := z - 1) = \textbf{while } 0 \leq z \textbf{ do x} := y * z\,\dot{;}\,\textbf{z} := z - 1 \textbf{ od}$$

ANSWER TWO: A Semantic algebra, *Sem.* Let *Env* denote the set $[X \to \mathbf{Z}]$ of all total mappings from X into $\mathbf{Z}$ the set of integers. Then take

$Sem_{\langle st \rangle}$ to be the set $[Env \rightharpoonup Env]$ of all partial mappings from *Env* to itself.

$Sem_{\langle ae \rangle}$ to be the set $[Env \rightharpoonup Env \times \mathbf{Z}]$

$Sem_{\langle be \rangle}$ to be the set $[Env \rightharpoonup Env \times [2]]$

It remains to define appropriate operations for each of the operators given in the signature. We will only show on example of how this can be done, the reader interested in seeing the definition of all the operations will find them in [14].

The example we will show is again that for "whiledo". For $whiledo_{Sem}$ we want a mapping

$$whiledo_{Sem} : Sem_{\langle be \rangle} \times Sem_{\langle st \rangle} \to Sem_{\langle st \rangle}$$

. We claim that the desired mapping is such that for any $\beta \in Sem_{\langle be \rangle}$ (so $\beta : Env \rightharpoonup Env \times [2]$) and $\tau \in Sem_{\langle st \rangle}$ (so $\tau : Env \rightharpoonup Env$)

$$whiledo_{Sem}(\beta,\ \tau) \ = \ ((\tau + 1_{Env}) \bullet \beta)^{\dagger},$$

where, for any $f : A \rightharpoonup A \times [2]$, we define $f^{\dagger} : A \rightharpoonup A$ to be the partial function which is the least upper bound of the sequence $\langle f^{(k)} \mid k \in \omega \rangle$ such that $f^{(0)} = \emptyset$ (i.e., the everywhere undefined function), and $f^{(k+1)} = [f^{(k)}, 1_A] \bullet f$ where, for any $\langle a, i \rangle \in (A \times [2])$ and $f_1, f_2 : A \rightharpoonup B$,

$$[f_1,\ f_2](a,\ i) \ = \ f_i(a)$$

and

$$(f_1 + f_2)(a,\ i) = \langle f_i(a),\ i\rangle.$$

It can be shown that the syntactic algebra *Syn* has the property that for *any* algebra A with the same signature there exists a unique homomorphism $h_A : Syn \to A$. In particular, this is true for the algebra *Sem*. Indeed, $h_{Sem} : Syn \to Sem$ takes each "expression" in a carrier of *Syn* to its "meaning" in the corresponding carrier of *Sem*. Thus h_{Sem} gives the semantics for the language. This kind of semantics is called *initial algebra semantics* for reasons that will become apparent later.

□

Example 2.9 In the above example the syntactic algebra, *Syn*, has the property that for any other algebra A with the same signature there is a unique homomorphism $h : S \to A$. This would not be the case if the grammar, G, was ambiguous. However we would still be able to construct an algebra, *Abs*, with the property that for every algebra A with the same signature there was a unique homomorphism $h_A : I \to A$. The algebra I would correspond to the "abstract syntax" of the language (sets of strings) generated by the grammar, rather than to the language *per se*. That is, the carriers of *Abs* would consist of "parse trees" rather than strings.

A grammar G is ambiguous if some elements of $L_G = \{u \in T^* \mid \langle exp\rangle \Rightarrow u\}$ can be generated in more than one manner.

□

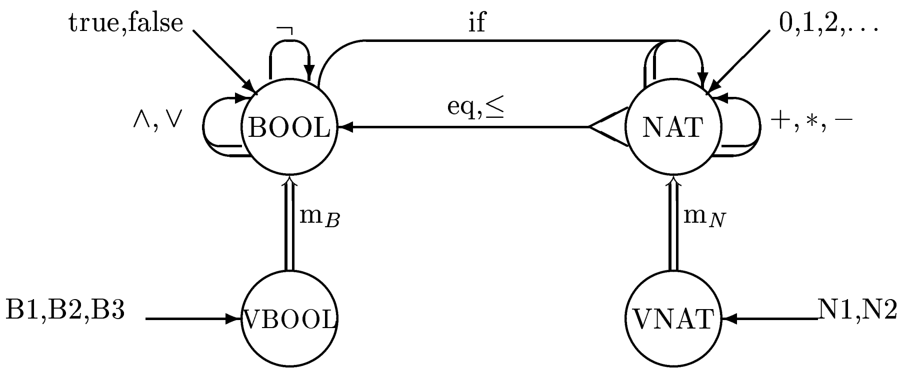
Example 2.10 Here is yet another way in which algebras can appear in connection with programs.

Consider a programming language in which declaration is dynamic in the sense that new data types, new variables, and new pointers can be created as the program goes along.

Any language with pointers has at least some dynamic declarations as pointers are dynamically created (and destroyed). Any language with block structure (i.e., with local variables) has dynamic declaration on the intuitive level but the implementation may be more static. Not very many languages permit dynamic data type creation over a wide range but it is possible (e.g., see ML).

In such a language with some form of dynamic declaration, the "current state" can be represented as an algebra in which all the currently declared data types, variables, pointers, etc. appear. Then a change of state corresponds to a change of the algebra, that is it is an operation *on* the algebra rather than, as in our last example, an operation *in* the algebra.

Here is a simple example of this idea applied to the treatment of variables. In the following picture we have have an algebra with the sorts BOOL and NAT and their associated operations *plus* two additional sorts VBOOL and VNAT and two additional operators m_B and m_N.



The intended interpretation is that the new sorts are the names of the sets of variables for original types, that the constants of the new sorts are the declared variables of those sorts, and the new operators are the names of the functions that give the values of the variables. In the above picture we have variables B1, B2, and B3 of type BOOL, and variables N1 and N2 of type NAT. The picture only shows the signature. An algebra A with this signature that provides the "usual" interpretation for BOOL, NAT and their operations will also an interpretations $(\mathrm{m}_B)_A$, $\mathrm{B1}_A$, $\mathrm{B2}_A$, and $\mathrm{B3}_A$ for, respectively, M_B, B1, B2, and B3, and interpretations $(\mathrm{m}_N)_A$, $\mathrm{N1}_A$, and $\mathrm{N2}_A$, for, respectively, m_N, N1 and N2. We can view the algebra A as denoting a state in which, for example, the variable N1 has value $(\mathrm{m}_N)_A(N1_A)$, the variable N2 has value $(\mathrm{m}_N)_A(N2_A)$, the variable B1 has value $(\mathrm{m}_B)_A(B1_A)$, etc. Assigning a new value to a variable requires changing the algebra by changing the appropriate operation. Declaring, or forgetting a variable (as on leaving a block) requires changing the signature by adding or deleting constant operators from the appropriate part of the signature.

The separation between the signature and the algebra plays a significant role in this model of state. For example, as variable N1≠N2, but we can easily choose an algebra A such that $\mathrm{N}_A = \mathrm{N2}_A$, and this corresponds then to a state in which "N1 and N2 are aliases". □

Later we will show how the same framework can be used to provide a treatment of pointers and a treatment of structures such as INTEGER-ARRAYS where there components of an array are viewed as "integer-holding-locations" rather than as integers, *per se*.

3 Introduction to Categories

3.1 Categories: Some Basic Definitions and examples

Definition 3.11 A *small category*, $\mathbf{C}$, consists of:

$Obj(\mathbf{C})$, a set of *objects*,

$Mor(\mathbf{C})$, a set of *morphisms*, together with mappings

$source, target : Mor(\mathbf{C}) \to Obj(\mathbf{C})$

$Id : Obj(\mathbf{C}) \to Mor(\mathbf{C})$

and,

$Comp : Mor(\mathbf{C}) \times Mor(\mathbf{C}) \to Mor(\mathbf{C})$ called *composition.*

Where, writing

$$1_A \text{ for Id(A)}$$
$$\beta \bullet \alpha \text{ for Comp}(\alpha,\, \beta)$$

these data are subject to the following axioms

Ax.1 $\beta \bullet \alpha$ is defined iff $target(\alpha) = source(\beta)$.

Ax.2 If $\beta \bullet \alpha$ is defined then $source(\beta \bullet \alpha) = source(\alpha)$ and $target(\beta \bullet \alpha) = target(\beta)$.

Ax.3 Composition is associative, i.e., if $(\gamma \bullet \beta) \bullet \alpha$ is defined then $(\gamma \bullet \beta) \bullet \alpha = \gamma \bullet (\beta \bullet \alpha)$.

Ax.4 $\alpha \bullet 1_{source(\alpha)} = \alpha = 1_{target(\alpha)} \bullet \alpha$.

□

Notation: Given a category $\mathbf{C}$ we shall generally write $|\mathbf{C}|$ for $Obj(\mathbf{C})$, and just $\mathbf{C}$ for $Mor(\mathbf{C})$. Given $X, Y \in |\mathbf{C}|$ we write $\mathbf{C}(X, Y)$ for the set of morphisms with source X, and target Y. We often write $\alpha : X \to Y$, or $X \overset{\alpha}{\to} Y$ for $\alpha \in \mathbf{C}(X, Y)$. We call $\beta \bullet \alpha$, "the composite of α and β", and we call 1_A, "the identity morphism on A".

We are also going to want to consider some categories which are not small, but *large*, in the sense that they have a proper class of objects and/or morphisms. An example is the category **Set** having all small sets as objects and all functions as morphisms. Obviously such large categories could lead to foundational problems (paradoxes) but we shall not worry about that in these notes. Indeed we shall adopt the sloppy, but convenient, approach of speaking of all the categories we encounter as if they were actually small, in that we will write expressions such as $X \in |\mathbf{Set}|$, and $\alpha : X \to |\mathbf{Set}|$ even though $|\mathbf{Set}|$ is a proper class.

Using the arrow notation for morphisms we can express the axioms pictorially. The idea is that in each of the following diagrams, any two paths with the same starting and ending points (source and target) represent equivalent morphisms (diagrams with this property are said to *commute*).

1,2)

$$\begin{tikzcd}
A \arrow[r, "\alpha"] \arrow[rr, bend right, "\beta \bullet \alpha"'] & B \arrow[r, "\beta"] & C
\end{tikzcd}$$

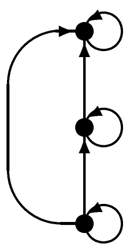
3)

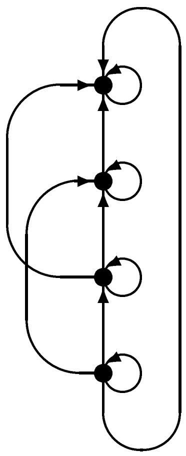
$$\begin{tikzcd}
A \arrow[r, "\alpha"] \arrow[rr, bend left, "\beta \bullet \alpha"] \arrow[rrr, bend left=50, "\gamma \bullet (\beta \bullet \alpha)"] \arrow[rrr, bend right=50, "(\gamma \bullet \beta) \bullet \alpha"'] & B \arrow[r, "\beta"] \arrow[rr, bend right, "\gamma \bullet \beta"'] & C \arrow[r, "\gamma"] & D
\end{tikzcd}$$

4)

$$\begin{tikzcd}
A \arrow[r, "1_A"] \arrow[rr, bend right, "\alpha"'] & A \arrow[r, "\alpha"] \arrow[rr, bend left, "\alpha"] & B \arrow[r, "1_B"] & B
\end{tikzcd}$$

Example 3.12 The simplest of all categories is the category **0** with no objects and no morphisms. Slightly larger is the category **1** with one object together with its identity morphism. Next we have the category **2** with two objects, call them A and B, the identity morphisms 1_A and 1_B, and a single morphism from A to B. And, while we're at it, here are the categories **3** and **4**.





The composition operations for **3** and **4** are straight forward – in each case where there are two morphisms that can be composed there is exactly one morphism with the right source and target to be their composite. Such very small categories have a role to play in computer science applications of category theory. For example, they play a role in the proofs of some of the fundamental results on parameterized data types, see, for example, [18]. For a simple application of **1** see the examples of comma categories in the section on Functors. □

Example 3.13 Let S be any set, then define the *discrete category* determined by S, to be the category $\mathbf{S}$ with objects, $|\mathbf{S}| = S$, and with no morphisms other than the identity morphisms $1_s : s \to s$, $s \in S$. $\square$

Example 3.14 Small categories of sets: **U-set**: Let U be some fixed set of sets, then define **U-set** to be the category with

$$|\textbf{U-set}| = U$$

and, for all $X, Y \in U$

$$\textbf{U-set}(X, Y) = \text{the set of all mappings } f : X \to Y.$$

The composition operation will just be the usual composition of functions, and the identity morphisms will be the identity functions. Note that for this to be a category, each function must have a source and target, thus a function $f : X \to Y$ cannot be identified with its graph, $G(f) = \{\langle x, y \rangle \mid f(x) = y\}$, as it is in many set theoretic treatments. Thus, in this paper, a function $f : X \to Y$ will always be regarded as a triple $\langle X, G(f), Y \rangle$. $\square$

Example 3.15 For each natural number $n \in \{0, 1, 2, \ldots\}$, let $[n] = \{1, 2, \ldots, n\}$. So, in particular, $[0] = \emptyset$, the empty set.

Define $\mathbf{N}$ to be the category with and for all n and p,

$$\mathbf{N}(n, p) = \text{the set of all mappings } f : [n] \to [p].$$

Composition is just functional composition. $\square$

Example 3.16 Preordered sets: Let $\mathbf{P} = \langle P, \leq \rangle$ be a pre-ordered set (so P is a set, and $\leq$ is a transitive and reflexive relation on P). Then we can view $\mathbf{P}$ as a category with

$$|\mathbf{P}| \;=\; P$$

and, for all $X, Y \in P$,

$$P(X, Y) = \begin{cases} \{\langle X, Y \rangle\} & \text{if } X \leq Y \\ \emptyset & \text{if } X \not\leq Y \end{cases}$$

Given $\langle X, Y \rangle : X \to Y$, and $\langle Y, Z \rangle : Y \to Z$, their composite is $\langle Y, Z \rangle \bullet \langle X, Y \rangle = \langle X, Z \rangle$. That $\langle X, Z \rangle \in \mathbf{P}(X, Z)$ is, of course, an immediate consequence of the transitivity of $\leq$.

Note that the categories $\mathbf{0}$, $\mathbf{1}$, $\mathbf{2}$, $\mathbf{3}$ and $\mathbf{4}$ are all categories corresponding to linearly ordered sets of the indicated cardinalities. $\square$

Definition 3.17 Given a set S, let S^*, the *set of strings on* S, be the set of all mappings $w : [n] \to S$, $n \in \omega$, where, as above, $[n]$ denotes the set $\{1, \ldots, n\}$. A string $u : [n] \to S$, is said to be of *length* n. Let $\varepsilon : [0] \to S$ denote the unique mapping from $[0]$ to S – ε is called *the empty string.* $\square$

Example 3.18 We form a category $\mathbf{Str}_S$ with objects $|\mathbf{Str}_S| = S^*$, where for $w, u \in |\mathbf{Str}_S|$, with $w : [n] \to S$, and $u : [p] \to S$, a *string morphism* $f : w \to u$ is a triple $\langle w, f, u \rangle$ where $f : [n] \to [p]$ is a mapping such that $u \bullet f = w$.

$$\begin{tikzcd} {[n]} \arrow[rr, "f"] \arrow[dr, "u"'] & & {[p]} \arrow[dl, "w"] \\ & S & \end{tikzcd}$$

The composition of $\langle u, f, v \rangle$ and $\langle v, g, w \rangle$ is $\langle u, g \bullet f, v \rangle$. In later applications of strings in this paper we will often denote a string morphism $\langle u, f, v \rangle$, ambiguously, by its middle component, f. Such abbreviated notation is common and convenient, but it is important to remember that a "real" morphism always has a designated source and target and that they can play a critical role in some arguments. $\Box$

Example 3.19 There is also another category of "strings on S". Let $\mathbf{Mon}_S$ be the category where $|\mathbf{Mon}_S| = \{*\}$, a one element set, and $\mathbf{Mon}_S(*, *) = S^*$. That is, the morphisms in $\mathbf{Mon}_S$ are examplectly the strings on S. Composition is concatenation, that is, given strings $u, v \in S^*$, where $u : [n] \to S$ and $v : [p] \to S$, then their composite is the string $uv : [n+p] \to S$ such that

$$uv(i) = \begin{cases} u(i) & \text{if } i \leq n \\ v(i-n) & \text{if } i > n \end{cases}$$

The identity on $*$ is just the empty string λ. $\Box$

Example 3.20 Let S be a set, then we have a category $\mathbf{Sig}_S$ of S-sorted signatures, where the objects are the S-sorted signatures and the morphisms are the S-sorted signature morphisms as defined in the preceding section. Composition of morphisms is defined componentwise, that is, if $f_1 : \Sigma_1 \to \Sigma_2$ and $f_2 : \Sigma_2 \to \Sigma_3$ are signature morphisms with $f_i = \langle \Sigma_1,\ \overline{f_i},\ \Sigma_{i+1} \rangle$, $\overline{f_i} = \langle (f_1)_{w,s} \mid w \in S^*, s \in S \rangle$ then $f_2 \bullet f_1 = \langle \Sigma_1,\ \overline{h_i},\ \Sigma_{i+1} \rangle$, $\overline{h_i} = \langle (f_2)_{w,s} \bullet (f_1)_{w,s} \mid w \in S^*, s \in S \rangle$. $\Box$

Signatures are a special case of a more general concept, the S-sorted set or S-index family of sets. The difference is more of intention rather than extension.

Example 3.21 Let S be a set, then we define *the category of S-indexed sets*, denoted $\mathbf{Set}^S$ (for reasons that will later become more apparent) to be the category with, as objects, all S-indexed families of sets $A = \langle A_s \mid s \in S \rangle$, and with, as morphisms between S-indexed sets A and B, all S-indexed families, $f = \langle f_s : A_s \to B_s \mid s \in A \rangle$ of mappings. $\Box$

All the examples given so far have been of small categories. We now look at some important examples of large categories.

Example 3.22 Let **Set** be the category of all small sets, with, as morphisms, all total functions between them. The identities are identity functions, composition is the usual functional composition.

Let **Pfn** be the category of all small sets with all total, and partial, functions between them. Again we use the obvious identities and composition.

We call **Set** *the category of sets*, and **Pfn** *the category of partial functions.*

Given a S-sorted signature Σ let $\mathbf{Alg}_\Sigma$ be the category of all Σ-algebras together with the homomorphisms between them. Given Σ-algebras A, B, and C with homomorphism $f : A \to B$ and $g : B \to C$ the composite $g \bullet f$ of f and g is the homomorphism $\langle A, \overline{h}, B\rangle$ where $\overline{h} = \langle h_s = g_s \cdot f_s \mid s \in S\rangle$. □

We close this section by defining a number of special types of morphisms: monomorphisms, epimorphisms, isomorphisms, retractions, and coretractions. Monomorphisms and coretractions are generalizations of the concept of an injective mapping (one-to-one mapping), epimorphisms and retractions generalize the concept of a surjective mapping (onto mapping), an isomorphism generalizes the concept of a bijection (one-to-one and onto mapping).

Definition 3.23 Let $\mathbf{C}$ be a category, then we say that a morphism $\alpha : X \to Y$ in $\mathbf{C}(X,Y)$ is

i) a *monomorphism* if for all $Z \in |\mathbf{C}|$, and all morphisms $f, g : Z \to X$, $\alpha \bullet f = \alpha \bullet g$ implies $f = g$.

ii) an *epimorphism* if for all $Z \in |\mathbf{C}|$, and all morphisms $f, g : Y \to Z$, $f \bullet \alpha = g \bullet \alpha$ implies $f = g$.

iii) an *isomorphism* if there exists a morphism $\beta : Y \to X$ such that, $\alpha \bullet \beta = 1_Y$, and $\beta \bullet \alpha = 1_X$.

iv) a *retraction* if there exists $\beta : Y \to X$ such that $\alpha \bullet \beta = 1_Y$.

iv) a *coretraction* (or *section*) if there exists $\beta : Y \to X$ such that $\beta \bullet \alpha = 1_X$.

□

Proposition 3.24 *Every coretraction is a monomorphism and every retraction is an epimorphism.*

Proof We prove only the second part. Let $\alpha : X \to Y$ be a retraction. Then there exists $\beta : Y \to X$ such that $\alpha \bullet \beta = 1_Y$. Let $f, g :, Y \to Z$ such that $f \bullet \alpha = g \bullet \alpha$, then

$$f = f \bullet 1_Y = f \bullet \alpha \bullet \beta = g \bullet \alpha \bullet \beta = g \bullet 1_Y = g,$$

so α is an epimorphism. □

In **Set** the epimorphisms are the surjective mappings, and every epimorphism is a retraction. On the other hand, while the monomorphisms are the injective mappings, a monomorphism $m : A \to B$ is a coretraction iff $A \neq \emptyset$. Further useful refinements of the notions of monomorphism and epimorphism are found in the mathematical and computer science literature, see, for example, [8], [13].

4 More Algebra

4.1 Review of Basic definitions

We continue with the introduction to universal algebra. Recall the following definition of signature and algebra.

Definition 4.25 Let S be a set (of *sorts*).

An *S-sorted signature* is an $(S^* \times S)$-indexed family of sets

$$\Sigma = \langle \Sigma_{w,s} \mid w \in S^*, s \in S \rangle.$$

An element $\sigma \in \Sigma_{w,s}$ is an *operator symbol* of *rank* $\langle w, s \rangle$, *arity* w, and *sort* s.

For an S-sorted signature Σ, a *Σ-algebra* A consists of

- An S-indexed family of sets $U(A) = \langle A_s \mid s \in S \rangle$.
- A mapping $\sigma_A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ for each $w = s_1 \cdots s_n$ in S^*, $s \in S$ and $\sigma \in \Sigma_{w,s}$.

The set A_s is the *carrier of sort* s, σ_A is the *operation* in A named by the operator σ, if $\sigma \in \Sigma_{\varepsilon,s}$ then σ_A is said to be a *constant* of A. For $w = s_1 \cdots s_n \in S^*$ we will often write A^w for $A_{s_1} \times \ldots \times A_{s_n}$. □

As noted in the preceding section, Σ-algebras form a category where the morphisms are the Σ-homomorphisms.

4.2 Word Algebras and Initial Algebras

Definition 4.26 For Σ-algebras A and B, a *Σ-homomorphism* $h : A \to B$ is a triple

$$\langle A,\ \langle h_s : A_s \to B_s \mid s \in S \rangle,\ B \rangle$$

such that, for every $\sigma \in \Sigma_{w,s}$ with $w = w_1 \cdots w_n$ and for all $\langle a_1, \ldots, a_n \rangle \in A^w$

$$h_s(\sigma_A(a_1, \ldots, a_n)) = \sigma_B(h_{w_1}(a_1), \ldots, h_{w_n}(a_n)).$$

We write $\mathbf{Alg}_\Sigma$ to denote the category of all Σ-algebras together with the Σ-homomorphisms between them. □

We now introduce a special Σ-algebra known as the Σ-word-algebra. Word algebras, and the algebras isomorphic to them, play a special role in many computer science applications.

Definition 4.27 Let Σ be an S-sorted signature.

For each $s \in S$ defne $(T_\Sigma)_s$, *the set of s-ary Σ-terms*, to the the smallest set of strings on the set $\bigcup \langle \Sigma_{u,t} \mid u \in S^*, t \in S \rangle \cup \{(,)\}$ such that

1. $\Sigma_{\varepsilon,s} \subseteq (T_\Sigma)_s$.
2. If $w = w_1 \cdots w_n$, $\sigma \in \Sigma_{w,s}$ and $t_i \in (T_\Sigma)_{w_i}$, $i = 1, \ldots n$ then the string $\sigma(t_1 \cdots t_n)$ is in $(T_\Sigma)_s$.

.

The Σ-*word-algebra* is the Σ-algebra, $\mathbf{T}_\Sigma$ with carriers $\langle (T_\Sigma)_s \mid s \in S\rangle$, and where, for each $\sigma \in \Sigma_{w,s}$ where $w = w_1 \cdots w_n$,

$$\begin{aligned} \sigma_{T_\Sigma} : (T_\Sigma)_{w_1} \times \cdots \times (T_\Sigma)_{w_n} &\to (T_\Sigma)_s \\ \langle t_1, \ldots, t_n\rangle &\mapsto \sigma(t_1 \cdots t_n) \end{aligned}$$

□

Definition 4.28 An object T in a category $\mathbf{C}$ is *initial* in $\mathbf{C}$ if for every $A \in |\mathbf{C}|$ there is exactly one morphism

$$f_A : I \to A.$$

□

Fact 4.29 *If I_1 and I_2 are inital objects in category $\mathbf{C}$ then they are isomorphic.*

Proof Since I_1 is initial there exists a unique morphism $f_1 : I_1 \to I_2$. Since I_2 is initial there exists a unique morphism $f_2 : I_2 \to I_1$. Now $f_2 \bullet f_1 : I_1 \to I_1$, but also $1_{I_1} : I_1 \to I_1$ by the definition of category. But then $f_2 \bullet f_1 = 1_{I_1}$ by the definition of initiality. Similarly $f_1 \bullet f_2 = 1_{I_2}$. Thus f_1 (and also f_2) is an isomorphism. □

Notation: If $h : A \to B$ is an isomorphism then write $h : A \cong B$, and let $h^{-1} : B \to A$ be the homomorphism such that $h^{-1} \bullet h = 1_A$ and $h \bullet h^{-1} = 1_B$.

Fact 4.30 *If $h : A \cong B$ and A is initial then B is initial.*

Proof If $h : A \cong B$ and A is initial then for any object C we have $f_A : A \to C$, whence $f_A \bullet h^{-1} : B \to C$ so there is at least one morphism from B to C. Now say $f, g : B \to C$, then$f \bullet h \;=\; g \bullet h$ by the initiality of A, but then

$$f \;=\; f \bullet h \bullet h^{-1} \;=\; g \bullet h \bullet h^{-1} \;=\; g.$$

□

Notation: Given a homomorphism $h : A \to B$

$$h \;=\; \langle A,\ \langle h_s : A_s \to B_s \mid s \in S\rangle,\ B\rangle$$

let

$$U(h) \;=\; \langle h_s : A_s \to B_s \mid s \in S\rangle$$

a morphism between S-indexed sets. Let

$$U(A) \;=\; \langle A_s \mid s \in S\rangle$$

the S-indexed set of carriers of A.

Fact 4.31 *For any homomorphisms $f : A \to B$ and $g : B \to C$, we have $U(g \bullet f) = U(g) \bullet U(f)$.*

Theorem 4.32 *Let Σ be an S-sorted signature. Then $\mathbf{Alg}_\Sigma$ has initial objects. Indeed $\mathbf{T}_\Sigma$ is initial in $\mathbf{T}_\Sigma$.*

Proof The proof goes by structural induction on the definition of $\mathbf{T}_\Sigma$. Given a Σ-algebra A we want

$$h_A : \mathbf{T}_\Sigma \to A.$$

But it is then necessary and sufficient to take

$h_A(\sigma) = \sigma_A$ for $\sigma \in \Sigma_{\varepsilon,s}$

$h_A(\sigma(t_1 \cdots t_n)) = \sigma_A(h_A(t_1) \cdots h_A(t_n))$, for $\sigma \in \Sigma_{w,s}$, $w = w_1 \cdots w_n$, and $t_i \in \Sigma_{w,w_i}$, $i = 1, \ldots, n$. □

□

In the above we defined Σ-terms as strings, an alternative is to define them as tree labeled with elements of Σ and X. Here is a precise definition of such trees.

Definition 4.33 Let Σ be an S-sorted signature. For each $n \in [\omega]$ and $u \in S^n$ let $X_u = \{x_1^u, \ldots, x_n^u\}$ be an $[n]$-indexed set (of *variables*) disjoint from $\Sigma_{w,s}$ for all $w \in S^*$ and $s \in S$, and disjoint from all other X_v for $v \neq u$. Call $x_i^u \in X_u$ the *ith variable of* X_u. Define $\Sigma \cup X_u$ to be the S-sorted signature with $(\Sigma \cup X_u)_{\lambda,s} = \Sigma_{\lambda,s} \cup \{x_i^u \mid u_i = s\}$, and with $(\Sigma \cup X_u)_{v,s} = \Sigma_{v,s}$ for $v \neq \lambda$.

For each $w \in S^*$ define the set $T_\Sigma(X_w)$ of *total finite w-ary S-sorted Σ-trees* as the set of all partial functions $t : [\omega]^* \multimap\!\!\!\to \Sigma \cup X_w$ such that,

1. for all $v \in [\omega]^*$, and $j \in [\omega]$, if $t(vj) = \xi \in (\Sigma \cup X_w)_{v,s}$, then there exists $n \geq j$, and $u \in S^n$, $q \in S$, and $\alpha \in (\Sigma \cup X_w)_{u,q}$, such that $u_j = s$ and $t(v) = \alpha$.
2. for all $v \in [\omega]^*$, if $t(v) = \alpha \in \Sigma_{u,s}$, where $u \in S^n$, then for each $i \in [n]$, $t(vi)$ is defined.
3. t is defined on a finite, nonempty, subset of $[\omega]^*$.

A w-ary tree t will be said to be of *sort* s if $t(\lambda) \in \Sigma_{u,s} \cup \{x_i^w \mid w(i) = s\}$ for some $u \in S^*$. □

4.3 First Applications of Initiality

It might seem that the existence of initial algebras in the category of Σ-algebras is a fact so trivial, and so abstract, that it can't have any practical applications. However, it turns out that it such a basic fact that you have probably used it without knowing about it. In particular, many familiar inductively defined functions can be described as the unique homomorphism from $\mathbf{T}_\Sigma$ to an appropriate Σ-algebra:

Example 4.34 Let $S = \{1\}$ and identify each $n \in \mathbf{N}$ with the unique string in $\{1\}^n$. Let Σ be the $\{1\}$-sorted signature with

$\Sigma_{0,1} = \mathbf{N}$

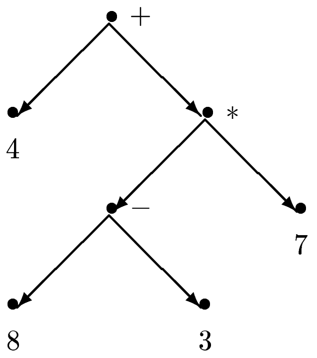
$\Sigma_{1,1} = \{succ, pred\}$

$\Sigma_{2,1} = \{+, *, -\}$

Now take A to be the Σ algebra with one carrier, $\mathbf{N}$ where all the operators are given "the usual interpretation", i.e.,

> The operator $n \in \Sigma_{0,1}$ is interpreted as the natural number n.
>
> the operator *succ* is interpreted as the successor function
>
> $$\begin{gathered} succ : \mathbf{N} \to \mathbf{N} \\ n \mapsto n+1 \end{gathered}$$
>
> the operator *pred* is interpreted as the predecessor function
>
> $$\begin{gathered} pred : \mathbf{N} \to \mathbf{N} \\ n \mapsto n-1, \text{ if } n > 0 \\ 0 \mapsto 0 \end{gathered}$$
>
> $+$ goes to addition, $*$ goes to multiplication, $-$ goes to subtraction.

Then the unique homomorphism takes each tree t to its arithmetic evaluation. For example, the tree



to the natural number $39 = (4 + ((8 - 3) * 7))$. $\square$

Example 4.35 For background on infix, Polish, and Cambridge notation see Pratt.

The same kind of homomorphism "explains" the relationship between trees and expressions: Let Σ be as in the preceding example and consider the algebra, B, in which the carrier is the set of all strings on the alphabet (i.e. set) $K = \{$ `0, 1, 2,`...,`(, ), ,+, *, -`$\}$ (note that between the '`)`' and the '`+`' there is a '` `' (a blank)). Now interpret the operators so that

> $n \mapsto$ `n` (i.e. $3 \mapsto$ `3`, etc.)
>
> and for strings $u, v \in K^*$,
>
> $$succ_B(u) = \texttt{succ(}u\texttt{)}$$
>
> $$pred_B(u) = \texttt{pred(}u\texttt{)}$$

$$+_B(u,v) = (u \texttt{ + } v)$$

$$*_B(u,v) = (u \texttt{ * } v)$$

$$-_B(u,v) = (u \texttt{ - } v)$$

Then the unique homomorphism takes the tree in the preceding example to the infix expression

`(4 + ((8 - 3) * 7)).`

If we change to a somewhat different algebra of expressions, C, with

$n \mapsto \mathtt{n}$ (i.e. $3 \mapsto \mathtt{3}$, etc.)

and for strings $u, v \in K^*$,

$$succ_B(u) = u \texttt{ succ}$$

$$pred_B(u) = u \texttt{ pred}$$

$$+_B(u,v) = u \ v \texttt{ +}$$

$$*_B(u,v) = u \ v \texttt{ *}$$

$$-_B(u,v) = u \ v \texttt{ -}$$

then the unique homomorphism translates the same tree into reverse Polish notation as

`4 8 3 - 7 * +.`

Similarly, the algebra D with

$n \mapsto \mathtt{n}$ (e.g., $3 \mapsto \mathtt{3}$, etc.)

and for strings $u, v \in K^*$,

$$succ_B(u) = (\texttt{succ } u)$$

$$pred_B(u) = (\texttt{pred } u)$$

$$+_B(u,v) = (\texttt{+ } u \ v)$$

$$*_B(u,v) = (\texttt{* } u \ v)$$

$$-_B(u,v) = (\texttt{- } u \ v)$$

we get a homomorphism that translates the tree into Cambridge notation as

```
(+ 4 (* (- 8 3) 7)).
```

□

Example 4.36 Here is yet another use of initiality to define a meaningful function on trees. Given an S-sorted signature Σ let H be the Σ-algebra where

For each $s \in S$, $H_s = \omega$.

If $\sigma \in \Sigma_{\lambda,s}$ then $\sigma_H = 1$,

If $\sigma \in \Sigma_{u,s}$ and $\langle n_1, \ldots, n_p \rangle \in A^u$, (so each $n_i \in \mathbf{N}$) then

$$\sigma_H(n_1, \ldots, n_p) = 1 + max\langle n_1, \ldots, n_p \rangle.$$

We claim that the unique homomorphism takes each λ-tree to its height! □

We shall make repeated use of this kind of construction. More complex, and more interesting, applications of initiality in various categories will play an important role in several sections of these notes. In particular we will employ initiality to give semantics to specifications of data types and to provide the mathematical foundation for the denotational semantics of programming languages.

4.4 Free Algebras

We can exploit and generalize the notion of initality in a straightforward manner to introduce a notion of "variables".

Given an S-indexed family $X = \langle X_s \mid s \in S \rangle$ of sets and an S-sorted signature Σ, where for all $s, s' \in S$ and $w \in S^*$, $s \neq s$ implies X_s, $X_{s'}$ and $\Sigma_{\varepsilon,s}$ are pairwise disjoint, we define a new signature $\Sigma(X)$:

$$\Sigma(X)_{\varepsilon,s} = \Sigma_{\varepsilon,s} \cup X_s$$

$$\Sigma(X)_{w,s} = \Sigma_{w,s} \text{ for } w \neq \varepsilon$$

Let $\mathbf{T}_{\Sigma(X)}$ be the $\Sigma(X)$-word-algebra. We can easily view $\mathbf{T}_{\Sigma(X)}$ as a Σ-algebra, denoted $\mathbf{T}_\Sigma(X)$, where

$$(\mathbf{T}_\Sigma(X))_s = (\mathbf{T}_{\Sigma(X)})_s$$

and

$$\sigma_{\mathbf{T}_\Sigma(X)} = \sigma_{\mathbf{T}_{\Sigma(X)}}.$$

We call $\mathbf{T}_\Sigma(X)$ *the Σ-algebra freely generated by X*, or *the free Σ-algebra generated by X.*

Fact 4.37 *There is an inclusion morphism*

$$\begin{gathered}\eta_X : \langle X_s \mid s \in S \rangle \to \langle \mathbf{T}_\Sigma(X)_s \mid s \in S \rangle = U(T_\Sigma(X)) \\ X_s \subseteq (\mathbf{T}_\Sigma(X))_s \\ x \mapsto x_{T_{\Sigma(X)}}\end{gathered}$$

Theorem 4.38 *Let A be a Σ-algebra and let $f : X \to U(A)$, $f = \langle f_s : X_s \to A_s \mid s \in S \rangle$. Then there exists a unique homomorphism $\overline{f} : \mathbf{T}_\Sigma(X) \to A$ such that $U(\overline{f}) \bullet \eta_X \; = \; f$.*

$$\begin{tikzcd} X \arrow[r, "\eta_X"] \arrow[dr, "f"'] & U(\mathbf{T}_\Sigma(X)) \arrow[d, "U(\overline{f})"] \\ & U(A) \end{tikzcd} \qquad \begin{tikzcd} \mathbf{T}_\Sigma(X) \arrow[d, "\overline{f}"] \\ A \end{tikzcd}$$

Proof Regard A as an $\Sigma(X)$ algebra A' with $x_{A'} = f_s(x)$ for every $s \in X_s$, $s \in S$. Then $\mathbf{T}_{\Sigma(X)}$ initial implies there exists a unique $\Sigma(X)$-homomorphism $h' : \mathbf{T}_{\Sigma(X)} \to A'$. But then, $h = \langle \mathbf{T}_\Sigma(X),\ U(h'),\ A \rangle$ is a homomorphism such that $U(h) \bullet \eta_X \; = \; f$. But the argument goes both ways. □

4.5 Congruences

Definition 4.39 An *s-ary Σ-equation in X* is a pair

$$e \; = \; \langle L, R \rangle \in (\mathbf{T}_\Sigma(X))_s \times (\mathbf{T}_\Sigma(X))_s.$$

For $e \in T_{\Sigma(X)}$ let $var(e)$ be the family of elements of X appearing in e.

Exercise 4.40 Define var as a homomorphism from $\mathbf{T}_{\Sigma(X)}$ to an appropriate $\Sigma(X)$-algebra. □

A Σ-algebra A *satisfies* e if $\overline{f}(L) = \overline{f}(R)$ for all $f : var(e) \to A$

$$\begin{tikzcd} var(e) \arrow[r, "\eta_{var(e)}"] \arrow[dr, "f"'] & U(\mathbf{T}_\Sigma(var(e))) \arrow[d, "U(\overline{f})"] \\ & U(A) \end{tikzcd}$$

□

Definition 4.41 A *presentation* or *equational specification* $P = \langle S,\ \Sigma,\ E \rangle$ consists of a set S of sorts, an S-sorted signature Σ and a set E of Σ-equations. We may sometimes write only $\langle \Sigma, E \rangle$ leaving the determination of S to the context.

A $\langle S,\ \Sigma,\ E \rangle$-algebra (a P-algebra) is a Σ-algebra satisfying every $e \in E$.

Let $\mathbf{Alg}_{\Sigma,E}$ or $\mathbf{Alg}_P$ denote the category of all $P = \langle S, \Sigma, E \rangle$-algebras together with the Σ-homomorphisms between them. We will sometimes refer to the morphisms in $\mathbf{Alg}_P$ as *P-homomorphisms.* □

Fact 4.42 $\mathbf{Alg}_{\Sigma,E} \subseteq \mathbf{Alg}_\Sigma$.

Definition 4.43 Given $\langle S, \Sigma\rangle$, a *congruence* on $\mathbf{T}_\Sigma$ is a family

$$\equiv \;=\; \langle \equiv_s \mid s \in S\rangle$$

of equivalence relations ($\equiv_s$ on $(\mathbf{T}_\Sigma)_s$) such that if $\sigma \in \Sigma_{s_1\cdots s_n,s}$, and $t_{i,1}, t_{i,2} \in (\mathbf{T}_\Sigma)_s$, $t_{i,1} \equiv_s t_{i,2}$, $i = 1, \ldots, n$, then

$$\sigma(t_{1,1}\cdots t_{1,n}) \equiv_s \sigma(t_{2,1}\cdots t_{2,n}).$$

□

Definition 4.44 Given S-sorted signature Σ, and congruence $\equiv$ on $\mathbf{T}_\Sigma$, define $\mathbf{T}_\Sigma/\equiv$ to be the Σ-algebra with

$$(\mathbf{T}_\Sigma/\equiv)_s = T_{\Sigma,s}$$

for every $s \in S$. For $t \in T_{\Sigma,s}$ let $[t]$ denote its $\equiv_s$-equivalence class. Then for $\sigma \in \Sigma_{s_1\cdots s_n,s}$

$$\sigma_{\mathbf{T}_\Sigma/\equiv}([t_1],\ldots,[t_n]) \;=\; [\sigma(t_1\cdots t_n)].$$

□

Note that it is necessary to show that the above definition gives a well-defined algebra.

Definition 4.45 For a specification $\langle S,\ \Sigma,\ E\rangle$, a congruence $\equiv$ is said to *satisfy* E if, and only if, for all $e = \langle L, R\rangle \in E$, and $\theta : var(e) \to T_\Sigma$, $\overline{\theta}(L) \equiv \overline{\theta}(R)$. □

Proposition 4.46 *For each specification $P = \langle S,\ \Sigma,\ E\rangle$ there exists a smallest congruence $\equiv_P$ on $\mathbf{T}_\Sigma$ satisfying E, and $\mathbf{T}_\Sigma/\equiv_P$ is initial in $\mathbf{Alg}_{\Sigma,E}$.*

4.6 Generalized homomorphisms

We have defined homomorphisms between Σ-algebras for fixed Σ, we now want to extend this to define homomorphisms between algebras with different presentations. The first step is to introduce a concept of a morphism between presentations.

Definition 4.47 Let $P_1 = \langle S_1,\ \Sigma_1,\ E_1\rangle$ and $P_2 = \langle S_2,\ \Sigma_2,\ E_2\rangle$ be presentations then a *presentation morphism* from P_1 to P_2 is a four-tuple

$$\langle P_1,\ f_S,\ f_\Sigma,\ P_2\rangle$$

where

$$f_S : S_1 \to S_2$$

and

$$f_\Sigma = \langle (f_\Sigma)_{w,s} : (\Sigma_1)_{w,s} \to (\Sigma_2)_{f^*(w),f(s)} \mid w \in S_1^*,\ s \in S_1\rangle$$

where $f^* : S_1^* \to S_2^*$ is the evident extension of f to S_1^*, i.e., $f^*(w_1\cdots w_n) = f(w_1)\cdots f(w_n)$. This data is required to be such that "f_Σ preserves E". That is, if $\langle L, R\rangle \in (\mathbf{T}_{\Sigma_1}(X)) \times (\mathbf{T}_{\Sigma_1}(X))$ is an s-ary Σ_1-equation in X then the $f(s)$-ary Σ_2-equation $\langle L', R'\rangle \in (\mathbf{T}_{\Sigma_2}(X)) \times (\mathbf{T}_{\Sigma_2}(X))$ resulting from replacing all occurences of each $\sigma \in \Sigma_1$ by $f_\Sigma(\sigma)$ in L and R should be satisfied by all P_2-algebras. □

Definition 4.48 Let $P_1 = \langle S_1,\ \Sigma_1,\ E_1\rangle$ and $P_2 = \langle S_2,\ \Sigma_2,\ E_2\rangle$ be presentations, let $f = \langle f_S, f_\Sigma\rangle$ be a presentation morphism from P_1 to P_2, then, for every P_2-algebra A define $f^\flat(A)$ to be the P_1-algebra where,

$$(f^\flat(A))_s = A_{f_S(s)} \text{ for each } s \in S_1$$

$$\sigma_{f^\flat(A)} = ((f_\Sigma)_{w,s}(\sigma))_A \text{ for each } \sigma \in \Sigma_{w,s},\ w \in S_1^*,\ s \in S_1.$$

The conditions on f ensure that all the sources and targets are appropriate.

We can also define $f^\flat$ on P_2-homomorphisms, that is, if $h : A \to B$ is a homorphism between two P_2-algebras then

$$f^\flat(h) = \langle (f^\flat(h))_s : A_{f(s)} \to B_{f(s)} \mid s \in S_1\rangle$$

is a $P-1$-homomorphism from $f^\flat(A)$ to $f^\flat(B)$. □

Definition 4.49 Let $P_1 = \langle S_1,\ \Sigma_1,\ E_1\rangle$ and $P_2 = \langle S_2,\ \Sigma_2,\ E_2\rangle$ be presentations, and, for $i = 1, 2$, let A_i be a P_i-algebra, then by a *generalized homomorphism* from A_1 to A_2 we mean a four-tuple

$$\langle A_1,\ f : P_1 \to P_2,\ h : A_1 \to f^\flat(A_2),\ A_2\rangle$$

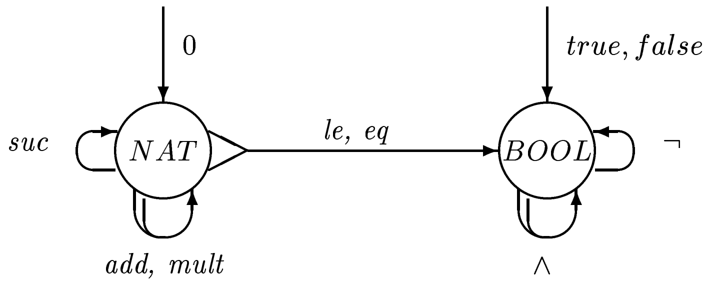
where $f = \langle f_S, f_\Sigma\rangle$ is a presentation morphism from P_1 to P_2, and h is a P_1-homomorphism from A_1 to $f^\flat(A_2)$. □

5 Algebraic Specification of Data Types

5.1 equational specifications

In this section we will give algebraic specification of a number of data types. The idea is that the specified data type is (the isomorphism class) of “the” initial algebra for the specification.

Example 5.50 We start with a specification of the natural numbers (NAT) and the booleans (BOOL). The picture for the signature is as follows:



The axioms (equations) are as follows

$$add(n,\ 0) = n$$

$$add(n,\ suc(p)) = suc(add(n,\ p))$$

$$\wedge(true,\ b) = b$$

$$\wedge(false,\ b) = false$$

$$\wedge(b,\ c) = \wedge(c,\ b)$$

$$\neg(true) = false$$

$$\neg(false) = true$$

$$le(0,\ n) = true$$

$$le(suc(n),\ 0) = false$$

$$le(n,\ p) = le(suc(n),\ suc)p))$$

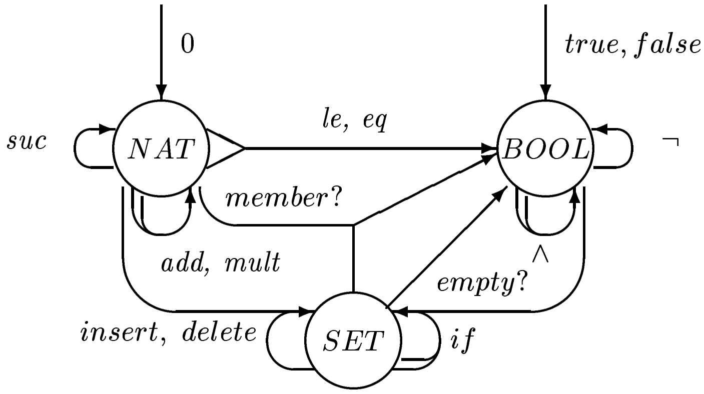
$$eq(n,p) = le(n,p) \wedge le(p,n)$$

$$mult(n,\ 0) = 0$$

$$mult(n,\ suc(p)) = add(n,\ mult(n,\ p))$$

The desired algebra is the initial algebra. $\square$

Example 5.51 As our second example we look at finite sets of natural numbers, with various assorted operations as indicated in the signature:



Here the axioms (in addition to those above) are:

$if(true,\ S1,\ S2) = S1$

$if(false,\ S1,\ S2) = S2$

$insert(N1,\ insert(N2,\ S)) = if(eq(N1,\ N2),\ insert(N2,\ S),\ insert(N2,\ insert(N1,\ S)))$

$delete(N1,\ create) = create$

$delete(N1,\ insert(N2,\ S)) = if(eq(N1,\ N2),\ delete(N1,\ S),\ insert(N2,\ delete(N1,\ S)))$

$member?(N1,\ create) = false$

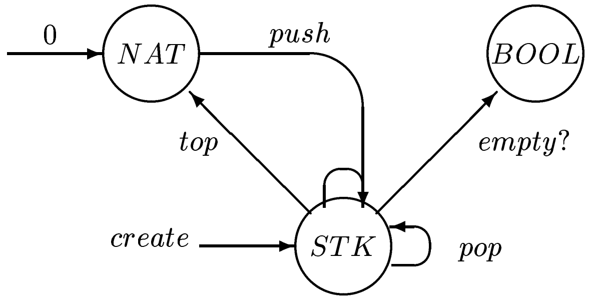
$member?(N1,\ insert(N2,\ S)) = if(eq(N1,\ N2),\ true,\ member?(N1,\ s))$

$empty?(create) = true$

$empty?(insert(N1,\ S)) = false$

□

Example 5.52 Here is a specification for stack-of-nat.



With axioms

$pop(push(N,\ S)) = S$

$top(push(N,\ S)) = N$

$empty?(create) = true$

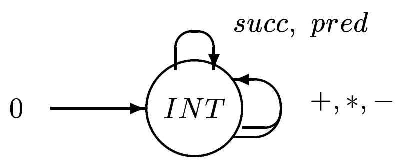
$empty?(push(N,\ S)) = false$

$pop(create) = create$

$top(create) = 0$

Note that the last axiom is rather arbitrarily chosen. The problem of what do with exceptions such as $top(create)$ is one of the difficulties with algebraic specifications. One solution is to go to partial algebras, that is, algebras where some of the operations are partial functions. □

Example 5.53 As our final example we give a specification for the integers:



Axioms:

$pred(succ(X)) = X$

$succ(pred(X)) = X$

$N + 0 = N$

$N1 + succ(N2) = succ(N1 + N2)$

$N1 + pred(N2) = pred(N1 + N2)$

$N - 0 = N$

$N1 - succ(N2) = pred(N1 - N2)$

$N1 - pred(N2) = succ(N1 - N2)$

$N * 0 = 0$

$N1 * succ(N2) = (N1 * N2) + N1$

$N1 * pred(N2) = (N1 * N2) - N1.$

□

5.2 Correctness of Specifications

How can we check to see if a a Σ-algebra A is a correct mathematical model of a data type specification $P = \langle S,\ \Sigma,\ E\rangle$? First of all, what do we mean by "correct"? We define A to be *correct with respect to* P if A is an initial P-algebra.

We now look at some ideas that have been useful in proving correctness. The key concept here is that of a canonical term algebra.

Definition 5.54 A $\langle S,\ \Sigma,\ E\rangle$-algebra C is said to be a *canonical term algebra* for $\langle S,\ \Sigma,\ E\rangle$ if, and only if,

1. $C_s \subseteq (\mathbf{T}_\Sigma)_s$, for all $s \in S$.
2. If $\sigma(t_1 \cdots t_n) \in C$ then $t_i \in C$ and

$$\sigma_C(t_1, \ldots, t_n) = \sigma(t_1 \cdots t_n).$$

□

Proposition 5.55 *Let $P = \langle S,\ \Sigma,\ E\rangle$ be a specification, then there exists an initial P-algebra which is a canonical term algebra.*

Proposition 5.56 *Let $P = \langle S,\ \Sigma,\ E\rangle$ be a specification and let C be a canonical term P-algebra. Then $C \cong \mathbf{T}_\Sigma$ if, and only if*

1. *C satisfies E.*
2. *For all $\sigma \in \Sigma_{s_1 \cdots s_n, s}$ and $t_i \in C_{s_i}$*

$$\sigma(t_1 \cdots t_n) \equiv \sigma_C(t_1, \ldots, t_n).$$

The canonical term algebras are often the way we think of the data types intuitively. The following examples bear this out.

Example 5.57 The canonical term algebra for the natural numbers consists of terms of the form

$$suc^n(0) \;=\; \underbrace{suc \cdots suc}_{n}(0)$$

□

Example 5.58 The canonical term algebra for stacks-of-nat consists of terms of the form

$$push(n_p,\ push(\cdots push(n_1,\ create)\cdots)).$$

□

Example 5.59 The canonical term algebra for integers consists of terms of the form

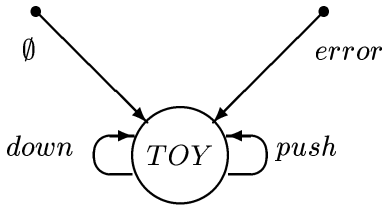
$$0 \text{ or } succ^n(0) \text{ or } pred^n(0)$$

□

5.3 Limits of Specifications

It might seem from our examples that any data type can be given a finite equational specification. We now show that that is not the case. For this purpose we introduce a "toy" example of an algebra that is easy enough to describe but can not be finitely specified. However, we can show that it can be specified by adding an additional operator and axioms. Such an additional operator is generally called a "hidden" operator. We will then sketch some results showing that any computable data type (to be defined) can be realized with the help of such hidden operators.

Here is the specification for our TOY data type. The signature is



Now consider the algebra with the above signature and with carrier

$$L \;=\; \{\infty\} \cup \{\langle p,\ s\rangle \mid 0 \leq p \leq s\}$$

and with the operations given as follows

$$error \;=\; \infty.$$

$$empty \;=\; \langle 0,\ 0\rangle.$$

$$down(\infty) \;=\; push(\infty) \;=\; \infty.$$

$$down(\langle p,\ s\rangle) \;=\; if\ p \geq 0\ then\ \langle p-1,\ s\rangle\ else\ \infty.$$

$$push(\langle p,\ s\rangle) \;=\; if\ p = s\ then\ \langle p+1,\ s+1\rangle\ else \infty.$$

Proposition 5.60 *The algebra toy has no finite equational specification.*

The proof can be found in: Thatcher, Wagner, Wright, [15].

But now consider the effect of adding an operator *shove* where

$$shove(\infty) \;=\; \infty.$$

$$shove(\langle p,\ s\rangle) \;=\; \langle p,\ s+1\rangle.$$

then we get the following equational specification

$$push(down(X)) = error$$

$$down(empty) = error$$

$$push(error) = down(error) = error$$

$$shove(error) = error$$

$$push(shove(X)) = error$$

$$down(shove(X)) = shove(down(X))$$

$$down(push(empty)) = shove(empty)$$

$$down(push(push(X))) = shove(push(X)).$$

Having shown that hidden operations are necessary we ask if they are sufficient. Here are some results (from Bergstra and Tucker, [1]) on this issue. First a definition

Definition 5.61 An *effective presentation* of an S-sorted Σ-algebra A is a pair $\langle \alpha,\ \Omega \rangle$, where Ω is an S-sorted family $\Omega = \langle \Omega_s \subseteq \omega \mid s \in S \rangle$ of recursive subsets of ω, and $\alpha = \langle \alpha_s : \Omega_s \to A_s \rangle$ is a family of surjective mappings, and for each $\sigma_A : A_{s_1} \times \cdots \times A \to A_s$ there is a recursive mapping $\overline{\sigma} : \Omega_{s_1} \times \cdots \times \Omega_{s_n} \to \Omega_s$ such that the diagram

$$\begin{tikzcd}
A_{s_1} \times \cdots \times A_{s_n} \arrow[r, "\sigma_A"] & A_s \\
\Omega_{s_1} \times \cdots \times \Omega_{s_n} \arrow[u, "\alpha_{s_1} \times \cdots \times \alpha_{s_n}"] \arrow[r, "\overline{\sigma}"'] & \Omega_s \arrow[u, "\alpha_s"']
\end{tikzcd}$$

commutes. Now, for each $s \in S$ and all $x, y \in \Omega_s$ define $\equiv_s$ to be the relation on Ω_s such that $x \equiv_s y$ if, and only if, $\alpha(x) = \alpha(y)$.

We say that A is *computable* if each relation $\equiv_s$ is computable, and we say that S is *semicomputable* if each relation $\equiv_s$ is semicomputable. □

Since (algebraic) data types are algebras we can now speak of computable and semicomputable data types. Bergstra and Tucker have shown the following:

Theorem 5.62 *Any computable data type can be algebraically specified by a finite set of equations involving a finite signature (with hidden operators) using initial algebra semantics.*

Theorem 5.63 *A data type A is semicomputable if, and only if, it can be specified by a finite set of equations involving $\langle S,\ \Sigma \rangle$ with hidden sorts and operators.*

5.4 Conditional Specifications

Definition 5.64 Given a signature $\langle S,\ \Sigma \rangle$, a *positive conditional axiom* for this signature is an $(n+1)$-tuple

$$c = \langle e_1, \ldots, e_n, e_{n+1} \rangle$$

of Σ-equations. Where $e_i = \langle L_i,\ R_i \rangle$, we can write c as:

$$L_1 = R_1 \ \& \ \cdots \ \& \ L_n = R_n \implies L_{n+1} = R_{n+1}$$

A Σ-algebra A *satisfies* $c = \langle e_1, \ldots e_n, e_{n+1} \rangle$ if, and only if, for each $\theta : var(c) \to A$, if $\overline{\theta}(L_i) = \overline{\theta}(R_i)$ for $i = 1, \ldots, n$ then $\overline{\theta}(L_{n+1} = \overline{\theta}(R_{n+1})$.

A *conditional specification* is a triple $\langle S,\ \Sigma,\ C \rangle$ where S is a set, Σ is an S-sorted signature, and C is a set of Σ-conditional axioms. □

Theorem 5.65 *Every conditional specification $P = \langle S,\ \Sigma,\ C \rangle$ defines a category $\mathbf{Alg}_P$ of Σ-algebras satisfyig C, and $\mathbf{Alg}_P$ has initial algebras.*

Proof We say a congruence $\equiv$ on $\mathbf{T}_\Sigma$ satisfies C if, and only if, for every $c \in C$, if $c = \langle e_1, \ldots, e_n, e_{n+1} \rangle$, $e_j = \langle L_j, R_j \rangle$, and for all $\theta : var(c) \to U(\mathbf{T}_\Sigma)$, if $\{ovl\theta(L_j) \equiv \overline{\theta}(R_j)$, for $i = 1, \ldots, n$, then $\overline{\theta}(L_{n+1}) \equiv \overline{\theta}(R_{n+1})$.

Now that $\equiv_C$ to be the smallest congruence satisfying C. Then $\mathbf{T}_\Sigma / \equiv_C$ will be the initial algebra in $\mathbf{Alg}_P$. □

Example 5.66 It is easy to give a specification of the equality and the less-than-or-equal relations on integers using conditional axioms

$$LE(N,\ N) = true$$

$$LE(succ(N),\ N) = false$$

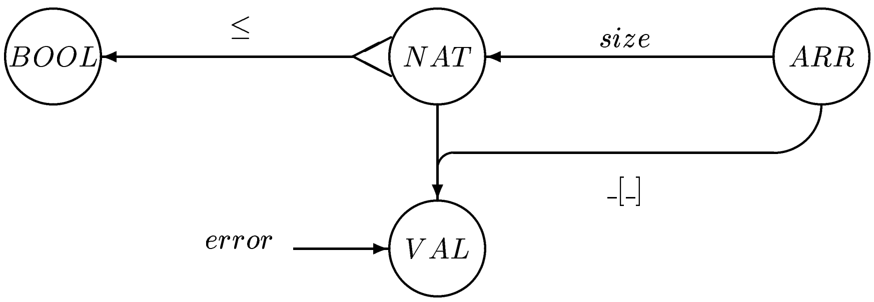
$$LE(I1,\ I2) = true \implies LE(I1,\ succ(I2)) = true$$

$$LE(I1,\ I2) = false \implies LE(succ(I1),\ I2) = false$$

$$EQ(I1,\ I2) = (LE(I1,\ I2) \wedge LE(I2,\ I1)$$

□

Example 5.67 Here is an application to the (partial) specification of arrays where there is a special value *error* for index-out-of-bound conditions:

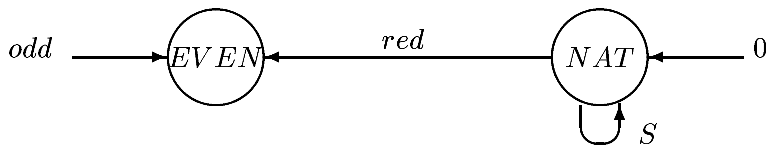


The desired axiom is:

$$((size(A) + 1) \leq n) \implies A[n] = error.$$

□

Conditional specifications are more expressive than equational specifications in the sense that there are algebras that are finitely axiomatizable with conditional axioms that have only infinite equational specifications. Here is one such



Consider the algebra A with the above signature and such that

$$A_{NAT} = \{0, 1, 2, \dots\}$$

$$A_{EVEN} = \{odd, 0, 2, 4, \dots, 2n, \dots\}$$

$$red(2n) = 2n$$

$$red(2n+1) = odd$$

It is easy to see that this algebra has the following conditional specfication:

$$red(S(0)) = odd$$

$$red(X) = odd \implies red(S(S(X))) = odd$$

Fact 5.68 *The above data type has no finite equational specification using only the given operations.*

For a proof see: Thatcher, Wagner, Wright [15]. 711-732.

6 More About Categories – Functors

In this section we want to introduce the categorical concepts of functors. Before doing that we will introduce the category of automata and the category of automata behaviors. These categories will provide useful examples.

6.1 Automata

Example 6.69 Automata: (After Goguen [3]) An *automaton* is specified by a 6-tuple, $\langle X, S, Y, \delta, \lambda, \sigma\rangle$, where

X is a set (of *inputs*),

S is a set (of *states*),

Y is a set (of *outputs*),

$\delta : X \times S \to S$ is the *transition function*,

$\lambda : S \to Y$ is the *output function*.

$\sigma \in S$ is the *starting state*.

Given two automata A_1 and A_2, where A_i has specification, $A_i = \langle X_i,\ S_i,\ Y_i,\ \delta_i, \lambda_i, \sigma_i\rangle$, define a morphism f from A_1 to A_2, to be a 5-tuple

$$f \;=\; \langle A_1,\ f_X : X_1 \to X_2,\ f_S : S_1 \to S_2,\ f_Y : Y_1 \to Y_2,\ A_2\rangle$$

with the properties that for all $x \in X_1$, and $s \in S_1$,

$f_S(\delta_1(x, s)) = \delta_2(f_X(x), f_S(s))$,

$f_Y(\lambda_1(s)) = \lambda_2(f_S(s))$,

and,

$f_S(\sigma_1) = \sigma_2$.

The first two requirements were written out using arguments x and s, such arguments can be eliminated by just using the composition operation and writing the requirements as follows

$$f_S \bullet \delta_1 = \delta_2 \bullet (f_X \times f_S)$$

and,

$$f_Y \bullet \lambda_1 = \lambda_2 \bullet f_S.$$

Such equations are often presented pictorially by saying that certain diagrams commute – i.e., that all paths with the same source and target correspond to equivalent diagrams. The commuting diagrams for the above equations are the following:

$$
\begin{tikzcd}
X_1 \times S_1 \arrow[r, "\delta_1"] \arrow[d, "f_X \times f_S"'] & S_1 \arrow[d, "f_S"] \\
X_2 \times S_2 \arrow[r, "\delta_2"'] & S_2
\end{tikzcd}
$$

$$
\begin{tikzcd}
S_1 \arrow[r, "\lambda_1"] \arrow[d, "f_S"'] & Y_1 \arrow[d, "f_Y"] \\
S_2 \arrow[r, "\lambda_2"'] & Y_2
\end{tikzcd}
$$

The class of all automata together with all automata morphisms between them, form a category **Auto**.

Let the set of inputs, X, be fixed, and define an X-*automaton* to be an automaton with set of inputs X. The class of all X-automata together with all morphisms f in which $f_X = 1_X$ (the identity on X) form a category $\mathbf{Auto}_X$. $\square$

Example 6.70 Automata Behaviors: Let A be an X-automata specified by the data $\langle X, S, Y, \delta, \lambda, \sigma \rangle$. Extend $\delta : X \times S \to S$ to $\delta^* : X^* \times S \to S$ "in the usual way", that is, if $w \in X^*$, $w : [n] \to S$, then

$$
\delta^*(w, s) = \begin{cases} s & \text{if } n = 0 \text{ (so } w = \varepsilon) \\ \delta^*(\overline{w},\ \delta(w(1), s)) & \text{if } n \geq 1 \end{cases}
$$

where

$$
\begin{aligned}
\overline{w} &: [n-1] \to S \\
i &\mapsto w(i+1).
\end{aligned}
$$

Now define

$$
\begin{aligned}
\delta^+ &: X^* \to S \\
w &\mapsto \delta^*(w, \sigma).
\end{aligned}
$$

The *behavior of* A, beh_A, is defined to be the mapping,

$$
beh_A \ = \ \lambda \bullet \delta^+ : X^* \to Y.
$$

More generally define an X-*behavior* to be a pair $\langle \beta : X^* \to Y, Y \rangle$ for any $Y \in\mid \mathbf{Set} \mid$. For any two X-behaviors $B_1 = \langle \beta_1, Y_1 \rangle$ and $B_2 = \langle \beta_2, Y_2 \rangle$ define an X-*behavior morphism* $f : B_1 \to B_2$ to be a 3-tuple $\langle B_1,\ f : Y_1 \to Y_2,\ B_2 \rangle$ such that $f \bullet \beta_1 = \beta_2$, i.e., such that the diagram

$$
\begin{tikzcd}
 & X^* \arrow[dl, "\beta_1"'] \arrow[dr, "\beta_2"] & \\
Y_1 \arrow[rr, "f"'] & & Y_2
\end{tikzcd}
$$

commutes. Then the class of all X-behaviors together with all morphisms between them forms a category $\mathbf{Beh}_X$. □

6.2 Functors

Definition 6.71 Let $\mathbf{C}$ and $\mathbf{D}$ be categories, then a *functor*, $F : \mathbf{C} \to \mathbf{D}$, consists of a mapping $|F| : |\mathbf{C}| \to |\mathbf{D}|$, together with an $(|\mathbf{C}| \times |\mathbf{D}|)$-indexed family

$$F \;=\; \langle F_{X,Y} : \mathbf{C}(X,Y) \to \mathbf{D}(|F|(X), |F|(Y)) \rangle$$

of mappings that preserve the categorical structure, i.e.,

$|F|(\mathit{source}(f)) = \mathit{source}(F(f))$, for all $X, Y \in |\mathbf{C}|$, and $f : X \to Y$.

$|F|(\mathit{target}(f)) = \mathit{target}(F(f))$, for all $X, Y \in |\mathbf{C}|$, and $f : X \to Y$.

$F(1_X) = 1_{|F|(X)}$ for all $X \in |\mathbf{C}|$

$F(g \bullet f) = F(g) \bullet F(f)$ for all $X, Y, Z \in |\mathbf{C}|$, $f : X \to Y$ and $g : Y \to Z$.

As is commonly done, we shall sometimes write $F(X)$ for $|F|(X)$. □

Example 6.72 The construction of the X-behavior of an X-automaton described in Example 6.70, corresponds to a functor

$$
\begin{gathered}
\mathit{beh}. : \mathbf{Auto}_X \to \mathbf{Beh}_X \\
A \mapsto \mathit{beh}_A \\
\langle A,\ 1_X,\ f_S,\ f_Y,\ B\rangle \mapsto \langle \mathit{beh}_A,\ f_Y,\ \mathit{beh}_B\rangle
\end{gathered}
$$

□

Definition 6.73 An important example of a large category is the category **Cat** of all small categories together with the functors between them. □

Definition 6.74 Let $\mathbf{C}$ and $\mathbf{D}$ be categories. Then $\mathbf{D}$ *is a subcategory* of $\mathbf{C}$ if $\mid \mathbf{D} \mid \subseteq \mid \mathbf{C} \mid$, $\mathit{Mor}(\mathbf{D}) \subseteq \mathit{Mor}(\mathbf{C})$, for each $\alpha \in \mathbf{D}$, $\mathit{source}_D(\alpha) = \mathit{source}_C(\alpha)$ and $\mathit{target}_D(\alpha) = \mathit{target}_C(\alpha)$, and $\alpha \bullet_D \beta$ defined implies $\alpha \bullet_D \beta = \alpha \bullet_C \beta$. □

Definition 6.75 Given that $\mathbf{D}$ is a subcategory of $\mathbf{C}$ we have an *inclusion functor* $I : \mathbf{D} \to \mathbf{C}$. □

Definition 6.76 Define a *graph* to consist of two sets V, and E together with a pair of mappings $s, t : E \to V$. V is the set of *vertices*, E is the set of *edges*, and an edge $e \in E$ has *source* $s(e)$, and *target* $t(e)$. $\square$

Example 6.77 Path Categories: Let $G = \langle V, E, s, t \rangle$ be a graph, let $n \in \omega$, and let $v, w \in V$, then by a *path in G of length n from v to w* we mean a triple $\langle v, p, w \rangle$ where p is a mapping $p : [n] \to E$ such that, if $n \geq 1$ then $s(p(1)) = v$, $t(p(n)) = w$, and for all $i = 1, \ldots, n-1$, $t(p(i)) = s(p(i+1))$, while if $n = 0$, then $v = w$ and $p = \varepsilon : [0] \to E$, the unique mapping from $[0] = \emptyset$ to E.

Given a path $\overline{p} = \langle u, p, v \rangle$ of length n from u to v together with a path $\overline{q} = \langle v, q, w \rangle$ of length m from v to w, define $\overline{q} \bullet \overline{p} = \langle u, r, w \rangle$ to be the path of length $(n+m)$ from u to w such that

$$r(i) = \begin{cases} p(i) & \text{if } i \leq n \\ q(i) & \text{if } i > n. \end{cases}$$

For each $v \in V$ let 1_v be the unique path, $\langle v, \varepsilon, v \rangle$, of length 0 from v to v.

It is easy to see that the paths in G form a category, $\mathbf{Path}_G$, where the set of objects is the set V, the set of morphisms is the set of all paths in G, the composition is the above "$\bullet$" operation, and the identity morphisms are the paths 1_v, $v \in V$. $\square$

Example 6.78 Given graphs $G_i = \langle V_i, E_i, s_i, t_i \rangle$, $i = 1, 2$, we define a *graph morphism* $f : G_1 \to G_2$ to be a quadruple $\langle G_1,\ f_V : V_1 \to V_2,\ f_E : E_1 \to E_2,\ G_2 \rangle$ such that

$$s_2 \bullet f_E = f_V \bullet s_1,$$

and

$$t_2 \bullet f_E = f_V \bullet t_1.$$

Or, equivalently, such that the following diagrams commute:

$$\begin{tikzcd} E_1 \arrow[r, "f_E"] \arrow[d, "s_1"'] & E_2 \arrow[d, "s_2"] \\ V_1 \arrow[r, "f_V"'] & V_2 \end{tikzcd} \qquad \begin{tikzcd} E_1 \arrow[r, "f_E"] \arrow[d, "t_1"'] & E_2 \arrow[d, "t_2"] \\ V_1 \arrow[r, "f_V"'] & V_2 \end{tikzcd}$$

Given graph morphisms $\langle G_1, f_V, f_E, G_2 \rangle$ and $\langle G_2, g_V, g_E, G_3 \rangle$ their composite is $\langle G_1, g_V \bullet f_V, g_E \bullet f_E, G_3 \rangle$ It is easy to see that the graphs, together with the graph morphisms between them, form a large category, **Graph**, of graphs. $\square$

Example 6.79 The construction of the path category of a graph, given in Example 6.77, corresponds to a functor

$$\mathit{path} : \mathbf{Graph} \to \mathbf{Cat}$$

There is an even simpler functor

$$gph : \mathbf{Cat} \to \mathbf{Graph}$$

which, given a category $\mathbf{C}$ just "forgets" the composition operation, Comp, and identity designator, Id, of the category so that all that is left is $|\mathbf{C}|$, $Mor(\mathbf{C})$, and the functions *source* and *target* which is exactly the data needed to specify a graph. □

Definition 6.80 Throughout the preceding development we have been talking about diagrams, and commuting diagrams, in an informal, pictorial, manner. Informally speaking a diagram in a category $\mathbf{C}$ is a graph in which the vertices are labeled by objects from $\mathbf{C}$ and the edges are labeled with morphisms from $\mathbf{C}$, where multiple vertices, and/or multiple edges, may have the same labels. Such an informal concept can not be manipulated in a rigorous, mathematical manner, but it is easy to give a precise categorical definition of diagram as follows: Let $\mathbf{S}$ and $\mathbf{C}$ be categories, then by *a diagram in* $\mathbf{C}$ *of shape* $\mathbf{S}$ we mean a functor $D : \mathbf{S} \to \mathbf{C}$. Because of the close relationship between graphs and path categories (of which more anon) the two definitions are intuitively very close. We shall make use of diagrams as functors in later sections of this paper. □

Definition 6.81 Let $\mathbf{C}$ and $\mathbf{D}$ be categories, then by $\mathbf{C} \times \mathbf{D}$, the *product (object) of* $\mathbf{C}$ and $\mathbf{D}$, we mean the category with objects $\mid \mathbf{C} \times \mathbf{D} \mid = \mid \mathbf{C} \mid \times \mid \mathbf{D} \mid$, morphisms $Mor(\mathbf{C}) \times Mor(\mathbf{D})$ (where "$\times$" denotes Cartesian product), and where for $\langle \alpha, \beta \rangle \in \mathbf{C} \times \mathbf{D}$, $source(\langle \alpha, \beta \rangle) = \langle source(\alpha), source(\beta) \rangle$, $target(\langle \alpha, \beta \rangle) = \langle target(\alpha), target(\beta) \rangle$, and, when defined,

$$\langle \alpha_2, \beta_2 \rangle \bullet_{\mathbf{C} \times \mathbf{D}} \langle \alpha_1, \beta_1 \rangle = \langle \alpha_2 \bullet_C \alpha_1, \beta_2 \bullet_D \beta_1 \rangle.$$

□

6.3 Comma Categories

Definition 6.82 Given categories $\mathbf{A}$, $\mathbf{B}$,and $\mathbf{C}$, and functors $F : \mathbf{A} \to \mathbf{C}$, and $G : \mathbf{B} \to \mathbf{C}$, we construct the *comma category* $(F \downarrow G)$ with, as objects all triples $\langle X,\ \alpha,\ Y \rangle$ where $X \in |\mathbf{A}|$, $Y \in |B|$, and $\alpha : F(X) \to G(Y)$, and as morphisms from $\langle X,\ \alpha,\ Y \rangle$ to $\langle W,\ \beta,\ Z \rangle$, pairs of morphisms $\langle f : X \to W,\ g : Y \to Z \rangle$ from $\mathbf{A}(X, W) \times \mathbf{B}(Y, Z)$ such that the diagram

$$\begin{tikzcd}
F(X) \arrow[r, "F(f)"] \arrow[d, "\alpha"'] & F(W) \arrow[d, "\beta"] \\
G(Y) \arrow[r, "G(g)"'] & G(Z)
\end{tikzcd}$$

commutes.

Note that this definition is abbreviated in that the source and targets of the morphisms are not specified. □

Example 6.83 In Definition 6.76 we defined a graph in terms of the data $\langle V,\ E,\ s,\ t \rangle$. An alternative definition is given by the data $\langle V,\ E,\ \delta : E \to V \times V \rangle$ where for any $e \in E$, $\delta(e) = \langle s(e), t(e) \rangle$. This definition corresponds to the comma category $(1_{Set} \downarrow \Delta)$ where $1_{Set} : \mathbf{Set} \to \mathbf{Set}$ is the identity functor and Δ is the *diagonal functor*

$$\Delta : \mathbf{Set} \to \mathbf{Set}$$
$$S \mapsto S \times S.$$

□

Some important special cases of comma categories, $(F \downarrow G)$, are those where one of the two defining functors has as its source the category $\mathbf{1}$ (with one object and one morphism). A functor $F : \mathbf{1} \to \mathbf{A}$ will, in effect, pick out an element $A \in |\mathbf{A}|$. Thus we can identify the functor with this designated element and write, say, $(A \downarrow G)$ rather than $(F \downarrow G)$. In what follows we will generally write $\ulcorner A \urcorner : \mathbf{1} \to \mathbf{A}$ for such a functor and write $(\ulcorner A \urcorner \downarrow G)$ for the comma category.

Example 6.84 Given a set S, the category $\mathbf{Str}_S$ of strings over S, defined in Example 3.18, can be viewed as the comma category $(I_N \downarrow \ulcorner S \urcorner)$ where $I_N : \mathbf{N} \to \mathbf{Set}$ is the functor of $\mathbf{N}$ into $\mathbf{Set}$ ($\mathbf{N}$ as in example 3.15) taking n to $[n] = \{1, \ldots, n\}$, and taking each morphism in $\mathbf{N}(n,p)$ to the corresponding function $f : [n] \to [p]$, and where $\ulcorner S \urcorner : \mathbf{1} \to \mathbf{Set}$ is the functor taking the object of $\mathbf{1}$ to the set S ($\mathbf{1}$ as in Example 3.12). The details are as follows: Let $*$ denote the sole object of $\mathbf{1}$, and let 1_* denote its sole morphism. Then the objects of $(I_N \downarrow \ulcorner S \urcorner)$ are triples $\langle n,\ u : [n] \to S,\ 1_* \rangle$, and, since $\ulcorner S \urcorner(1_*) = 1_S$, a morphism from $\langle n,\ u : [n] \to S,\ 1_* \rangle$ to $\langle p,\ v : [p] \to S,\ 1_* \rangle$ is given by a pair $\langle f : [n] \to [p],\ 1_S \rangle$ such that $v \bullet f = 1_S \bullet u = u$. We leave it to the reader to spell out the isomorphism between $\mathbf{Str}_S$ and $(I_N \downarrow \ulcorner S \urcorner)$. □

Here is an example of a comma category of the form $(\ulcorner A \urcorner \downarrow 1_{\mathbf{C}})$.

Example 6.85 The category $\mathbf{Beh}_X$ is another example of a comma category. $\mathbf{Beh}_X = (\ulcorner (X^*) \urcorner \downarrow 1_{\mathbf{Set}})$ where $\ulcorner (X^*) \urcorner : \mathbf{1} \to \mathbf{Set}$ taking the object of $\mathbf{1}$ to the set X^* of strings on X, and $1_{\mathbf{Set}} : \mathbf{Set} \to \mathbf{Set}$ is is the identity functor on the category $\mathbf{Set}$. □

Given a category $\mathbf{C}$, a comma category of the form $(1_{\mathbf{C}} \downarrow \ulcorner A \urcorner)$ is frequently called a *slice category*. The notation $\mathbf{C}/A$ is often used in the literature to denote such a slice category. It is worthwhile spelling out the definition of slice category in detail.

Definition 6.86 Let $\mathbf{C}$ be a category and let $A \in |\mathbf{C}|$. Then the slice category $\mathbf{C}/A$ has as objects all morphisms $f : X \to A$ in $\mathbf{C}$. Given objects $f : X \to A$ and $g : Y \to A$ in $\mathbf{C}/A$, a $(\mathbf{C}/A)$-morphism between them is a morphism $h \in \mathbf{C}(X,Y)$ such that $g \bullet h = f$, that is, such that the diagram

$$\begin{tikzcd} X \arrow[rr, "h"] \arrow[dr, "f"'] & & Y \arrow[dl, "g"] \\ & A & \end{tikzcd}$$

commutes. □

6.4 Op-categories and Hom-functors

Definition 6.87 Given any category $\mathbf{C}$ with composition operation $\bullet_C$, we can form another category, $\mathbf{C}^{op}$ just by "turning around all the arrows", that is is, we take $\mathbf{C}^{op}$ be the category with

$$| \mathbf{C}^{op} | = | \mathbf{C} |$$

and, for all $X, Y \in | \mathbf{C}^{op} |$,

$$\mathbf{C}^{op}(X, Y) = \mathbf{C}(Y, X),$$

and for $\alpha \in \mathbf{C}^{op}(X, Y)$ and $\beta \in \mathbf{C}^{op}(Y, Z)$

$$\beta \bullet_{C^{op}} \alpha = \alpha \bullet_C \beta.$$

Note that the identity morphisms in $\mathbf{C}^{op}$ are just those of $\mathbf{C}$. □

Definition 6.88 Let $\mathbf{C}$ be any small category, then for each $X \in |\mathbf{C}|$ we have the *covariant hom functor*

$$\mathbf{C}(X, \_) : \mathbf{C} \to \mathbf{Set}$$

which takes each object $Y \in |\mathbf{C}|$ to the set $\mathbf{C}(X, Y)$, and takes each morphism $\alpha : Y \to Z$ to the mapping $(\mathbf{C}(X, \_))(\alpha)$, denoted $\mathbf{C}(X, \alpha)$

$$\begin{aligned} \mathbf{C}(X, \alpha) : \mathbf{C}(X, Y) &\to \mathbf{C}(X, Z) \\ f &\mapsto \alpha \bullet f. \end{aligned}$$

Similarly, for each $X \in |\mathbf{C}|$, we get the *contravariant hom functor*,

$$\mathbf{C}(\_, X) : \mathbf{C}^{op} \to \mathbf{Set}$$

which takes each object $Y \in |\mathbf{C}^{op}|$ to the set $\mathbf{C}(Y, X)$, and takes each morphism $\alpha \in \mathbf{C}^{op}(Y, Z) (= \mathbf{C}(Z, Y))$ to the mapping $(\mathbf{C}(\_, X))(\alpha)$, denoted $\mathbf{C}(\alpha, X)$

$$\begin{aligned} \mathbf{C}(\alpha, X) : \mathbf{C}(Y, X) &\to \mathbf{C}(Z, X) \\ f &\mapsto f \bullet \alpha \end{aligned}$$

Combining the two yields a functor

$$\mathbf{C}(\_, \_) : \mathbf{C}^{op} \times \mathbf{C} \to \mathbf{Set}$$

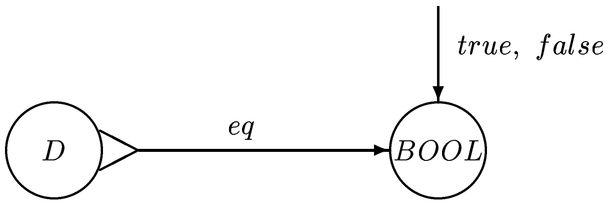
taking $\langle X, Y \rangle$ to $\mathbf{C}(X, Y)$, and for $\alpha_1 \in \mathbf{C}^{op}(X_1, Y_1)$ $(= \mathbf{C}(Y_1, X_1))$, and $\alpha_2 \in \mathbf{C}(X_2, Y_2)$, taking $\langle \alpha_1, \alpha_2 \rangle$ to the mapping $(\mathbf{C}(\_, \_))(\langle \alpha_1, \alpha_2 \rangle)$, denoted $\mathbf{C}(\alpha_1, \alpha_2)$

$$\begin{aligned} \mathbf{C}(\alpha_1, \alpha_2) : \mathbf{C}(X_1, X_2) &\to \mathbf{C}(Y_1, Y_2) \\ f &\mapsto \alpha_2 \bullet f \bullet \alpha_1 \end{aligned}$$

□

7 Parameterized Specifications

A close look at the specification of SET_OF_NAT given in Example 5.51 shows that it does not really depend on any of the operations defined in the specfication of NAT except for the eq operation and the specification of $BOOL$. Thus the specification should work for any data type with an appropriate eq operation and $BOOL$. Thus we ought to be able to write a specification P_1 with signature with sorts D and $BOOL$ an operation eq and constants $true$ and $false$ as follows:



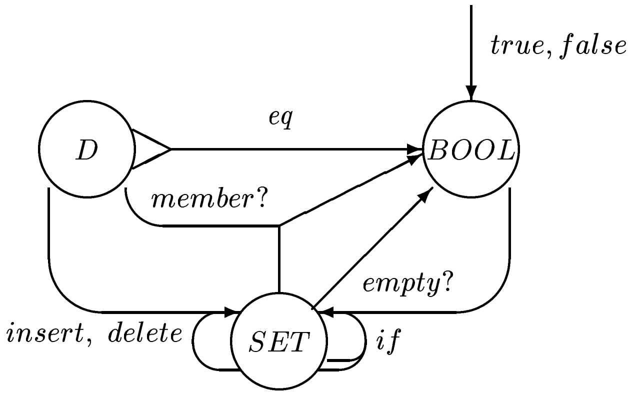
with axioms

$$eq(x, x) = true$$

$$eq(x, y) = true \;\&\; eq(y, z) = true \implies eq(x, z) = true$$

$$eq(x, y) = true \implies eq(y, x) = true$$

Then we should be able to write a specification P_2 for SET_OF_D, having the following signature:



and the axioms (in addition to those given in P_1):

$$if(true,\ S1,\ S2) = S1$$

$$if(false,\ S1,\ S2) = S2$$

$$insert(D1,\ insert(D2,\ S)) = if(eq(D1,\ D2),\ insert(D2,\ S),\ insert(D2,\ insert(D1,\ S)))$$

$$delete(D1,\ create) = create$$

$$delete(D1,\ insert(D2,\ S)) = if(eq(D1,\ D2),\ delete(D1,\ S),\ insert(D2,\ delete(D1,\ S)))$$

$member?(D1,\ create) = false$

$member?(D1,\ insert(D2,\ S)) = if(eq(D1,\ D2),\ true,\ member?(D1,\ s))$

$empty?(create) = true$

$empty?(insert(D1,\ S)) = false$

and this should provide a "parameterized" specification for SET_OF_D in the sense that we ought to be able to "put in" for D any data type A that appropriately satisfies the axioms of specification P_1, and get the "desired result", namely the data type SET_OF_A. This is, in fact, the case, but we have to make precise just what it means to "appropriately satisfy the axioms of P_1, what it means to "put in" the data type A for D in P_2, and what we mean by the "desired result".

First of all, not all algebras (data types) A satisfying the axioms for D are interesting. In particular, the initial P_1-algebra I_{P_1} is not very interesting since $(I_{P_1})_D = \emptyset$. But more important, we are really only interested in P_1-algebras A in which $A_{BOOL} = \{true_A,\ false_A\}$, $true_A \neq false_A$, and thus that eq_A is, in effect, an equivalence relation on A_D. Thus our concept of "appropriately satisfy the axioms of P_1" is that $BOOL$ and eq are "as expected". The question of how to specify such restrictions on a class of algebras is a subject which has received considerable study. For one approach see Wagner and Ehrig, *TCS*, 50 (1987) 323-349.

To answer the question of what it means to "put in" the data type A for D in P_2 we start by giving a general notion of parameterized algebraic data type.

Definition 7.89 A *parameterized data type specification* consists of a pair of

$P_1 = \langle S_1,\ \Sigma_1,\ E_1\rangle$, a presentation.

$P_2 = \langle S_2,\ \Sigma_2,\ E_2\rangle$, a presentation.

$H : P_1 \to P_2$, a presentation morphism, which is an inclusion morphism (so, in effect, $S_1 \subseteq S_2$; $(\Sigma_1)_{w,s} \subseteq (\Sigma_2)_{H(w),H(s)}$, for all $w \in S_1^*$, and $s \in S_1$; and $H(e) \in E_2$ for all $e \in E_1$ (this is not the most general notion of presentation morphism (see section **??**), but it will do for our present purposes)).

$$H : \langle S_1,\ \Sigma_1,\ E_1\rangle \hookrightarrow \langle S_2,\ \Sigma_2,\ E_2\rangle$$

□

Then, defining the functor $H^\flat : \mathbf{Alg}_{P_2} \to \mathbf{Alg}_{P_1}$ as in Definition 4.41, we get the following proposition:

Proposition 7.90 *Given*

$$H^\flat : \mathbf{Alg}_{P_2} \to \mathbf{Alg}_{P_1}$$

as above, there exists a functor $H^\sharp : \mathbf{Alg}_{P_1} \to \mathbf{Alg}_{P_2}$ *such that for all* $A \in |\mathbf{Alg}_{P_1}|$, $B \in |\mathbf{Alg}_{P_2}|$ *and* $h : A \to H^\flat(B)$ *there exists a unique morphism* $\overline{h} : H^\sharp(A) \to B$ *such that* $H^\flat(\overline{h}) = h$. *For reasons that will be apparent later we picture this with the following diagram*

$$\begin{tikzcd}
A \arrow[r, "\eta_A = 1_A"] \arrow[dr, "h"'] & H^\flat(H^\sharp(A)) \arrow[d, "H^\flat(\overline{h}) = h"] & H^\sharp(A) \arrow[d, "\overline{h}"] \\
 & H^\flat(B) & B
\end{tikzcd}$$

Claim: For P_1 and P_2 as in the example at the beginning of this section, if A is any P_1-algebra satisfying the "appropriate conditions" as described above, then $(H^\sharp(A))_{SET}$ is, up-to-isomorphism, precisely precisely the set of finite subsets of A_D modulo the equivalence relation eq_A.

Fact 7.91 *The eq operator on D is unavoidable. That is, we can not produce a computable data type $SETS\_OF\_A$ for a data type A where eq_A is not computable.*

Proof If we can define SET_OF_A without eq_A then we can define eq_A from SET_OF_A as follows:

$$eq(d_1,\ d_d) \;=\; empty?(delete(d_1,\ insert(d_2,\ create))).$$

$\square$

Ordinary (non-parameterized) data types are just a special case of parameterized data types. To see this consider the *empty presentation* $\langle \emptyset,\ \emptyset,\ \emptyset \rangle$. There is exactly one $\langle \emptyset,\ \emptyset,\ \emptyset \rangle$-algebra (the unique algebra with no carriers and no operations). Let $\star$ denote this algebra. Now, given any presentation $P = \langle S,\ \Sigma,\ E \rangle$, we clearly have a unique inclusion $H : \langle \emptyset,\ \emptyset,\ \emptyset \rangle \hookrightarrow \langle S,\ \Sigma,\ E \rangle$. Our claim then is that $H^\sharp(\star)$ is an initial P_1-algebra. To see this note first that, of necessity, $H^\flat(A) = \star$ for every P_1-algebra A, since $\star$ is the only $\langle \emptyset,\ \emptyset,\ \emptyset \rangle$-algebra. So, in particular, $H^\flat(H^\sharp(\star)) = \star$, and $\eta_\star = 1_\star$. But then $H^\sharp \dashv H^\flat$ says that there exists exactly one morphism (namely $\overline{1_\star}$) from $H^\sharp(\star)$ to A, thus $H^\sharp(\star)$ is initial in $\mathbf{Alg}_P$.

8 More About Categories - Natural Transformations

Given categories $\mathbf{C}$ and $\mathbf{D}$ there may, of course, be many functors from $\mathbf{C}$ to $\mathbf{D}$. To be able to make this class of functors into a category we introduce the concept of a natural transformation.

Definition 8.92 Let $\mathbf{C}$ and $\mathbf{D}$ be categories and let $F : \mathbf{C} \to \mathbf{D}$ and $G : \mathbf{C} \to \mathbf{D}$ be functors, then by a *natural transformation* $\eta : F \Rightarrow G$, we mean a mapping

$$\begin{aligned} \eta : |\mathbf{C}| &\to \mathbf{D} \\ X &\mapsto (\eta_X : F(X) \to G(X)) \end{aligned}$$

such that for all $X, Y \in |\mathbf{C}|$ and $f : X \to Y$ in $\mathbf{C}$, the following diagram commutes.

$$\begin{tikzcd} F(X) \arrow[r, "\eta_X"] \arrow[d, "F(f)"'] & G(X) \arrow[d, "G(f)"] \\ F(Y) \arrow[r, "\eta_Y"'] & G(Y) \end{tikzcd}$$

We shall use the double arrow notation, $\eta : F \Rightarrow G$, for natural transformations. Many other notations can be found in the literature. □

Definition 8.93 Given categories $\mathbf{C}$ and $\mathbf{D}$, we define *the functor category*, $\mathbf{Funct}(\mathbf{C}, \mathbf{D})$ or $\mathbf{D}^{\mathbf{C}}$, to have, as objects, all functors $F : \mathbf{C} \to \mathbf{D}$, and for morphisms between functors $F, G : \mathbf{C} \to \mathbf{D}$ all natural transformations $\eta : F \Rightarrow G$. Given $\eta_1 : F \Rightarrow G$ and $\eta_2 : G \Rightarrow H$ we define their *(vertical) composite* to be the natural transformation

$$\begin{aligned} \eta_2 \bullet \eta_1 : F &\Rightarrow H \\ X &\mapsto (\eta_2)_X \bullet (\eta_1)_X \end{aligned}$$

It is easy to see, from consideration of the following diagram, that this definition does indeed yield a natural transformation.

$$\begin{tikzcd} F(X) \arrow[r, "(\eta_1)_X"] \arrow[d, "F(f)"'] & G(X) \arrow[r, "(\eta_2)_X"] \arrow[d, "G(f)"'] & H(X) \arrow[d, "H(f)"'] \\ F(Y) \arrow[r, "(\eta_1)_Y"'] & G(Y) \arrow[r, "(\eta_2)_Y"'] & H(Y) \end{tikzcd}$$

Clearly the assumption that the squares commute forces the whole diagram to commute. □

Example 8.94 Given categories $\mathbf{S}$ and $\mathbf{C}$ the diagrams with shape $\mathbf{S}$ and "labels" in $\mathbf{C}$, see Definition 6.80, form a functor category. □

Many signficant categories can be described as special categories of diagrams or, equivalently, as special subcategories of functor categories.

Example 8.95 Let $\mathbf{D}$ be the category generated by the graph

$$\begin{tikzcd}
 & O \arrow[dr, "\sigma"] & & \\
X & & S \arrow[r, "\lambda"'] & Y \\
 & P \arrow[ul, "\pi_X"'] \arrow[ur, "\pi_S"] \arrow[ur, bend right, "\delta"'] & &
\end{tikzcd}$$

Let **Auto** be the full subcategory of functor $\mathbf{Set}^{\mathbf{D}}$ consisting of all functors F such that $F(O)$ is a singleton set and $F(P) = F(X) \times F(S)$, the cartesian product of $F(X)$ and $F(S)$, with projections $F(\pi_X)$ and $F(\pi_S)$.

We claim that **Auto** is, as the name suggests, isomorphic to the category of automata defined in Example 6.69. □

Example 8.96 There are many situations in which we want to employ indexed families of sets. Given a fixed index set S, we can view the S-indexed sets as a functor category. First, given S, let $\mathbf{S}$, denote the discrete category with $|\mathbf{S}| = S$, and no morphisms other than the identity morphisms (one for each $s \in S$). Then define $\mathbf{Set}^S$ to be the functor category $\mathbf{Funct}(\mathbf{S}, \mathbf{Set})$. Note that if F and $G \in |\mathbf{Set}|^{\mathbf{S}}$ (so, $F, G : \mathbf{S} \to \mathbf{Set}$), then a morphism from F to G, that is, a natural transformation $\eta : F \Rightarrow G$, is just an S-indexed family of mappings,

$$\eta = \langle \eta_s : F(s) \to G(s) \mid s \in S \rangle.$$

□

Example 8.97 Let S be a fixed set and consider the functor

$$(\_^S \times S) : \mathbf{Set} \to \mathbf{Set}$$

which takes a set X to the set $X^S \times S$, where $X^S = \mathbf{Set}(S, X)$, and which takes a mapping $f : X \to Y$, to the mapping $(\_^S \times S)(f)$, denoted $(f^S \times S)$,

$$\begin{gathered}(f^S \times S) : X^S \times S \to Y^S \times S \\ \langle h, s \rangle \mapsto \langle f \bullet h, s \rangle.\end{gathered}$$

Then we have a natural transformation

$$eval : (\_^S \times S) \Rightarrow 1_{\mathbf{Set}}$$

$$\begin{gathered}eval_X : X^S \times S \to X \\ \langle h, s \rangle \mapsto h(s).\end{gathered}$$

That is, given a pair $\langle h, s \rangle$¿ consisting of function $h : S \to X$, and an argument $s \in S$ for h, then we may "naturally transform" them into the value $h(s)$ of h at s. □

8.1 Horizontal Composition of Natural Transformations

We earlier defined the "vertical composition" of natural transformations, we will now define a "horizontal composition" and the connection between these two composites.

Given categories **A, B** and **C**, functors $F_1, F_2 : \mathbf{A} \to \mathbf{B}$, $G_1, G_2 : \mathbf{B} \to \mathbf{C}$, and natural transformations $\tau : F_1 \to F_2$ and $\eta : G_1 \to G_2$

$$\begin{tikzcd} \mathbf{A} \arrow[r, bend left, "F_1"] \arrow[r, bend right, "F_2"'] \arrow[r, phantom, "\Downarrow\tau"] & \mathbf{B} \arrow[r, bend left, "G_1"] \arrow[r, bend right, "G_2"'] \arrow[r, phantom, "\Downarrow\eta"] & \mathbf{C} \end{tikzcd}$$

we define the following additional natural transformations:

1. For $i = 1, 2$ define $G_i\tau : G_iF_1 \to G_iF_2$ by $(G_i\tau)_a = G_i(\tau_a)$ for each $a \in |\mathbf{A}|$.
2. For $i = 1, 2$ define $\eta D_i : G_1F_i \to G_2F_i$ by $(\eta F_i)_a = (\eta_{F_i(a)}$ for each $a \in |\mathbf{A}|$.
3. Define $\eta\tau : G_1F_1 \to G_2F_2$ by $(\eta\tau)_a = (\eta F_2)_a(G_1\tau)_a = (G_2\tau)_a(\eta F_1)_a$

   We call $\eta\tau$ the *horizontal composite* of τ and η.

Later, in the discussion of the application of Cartesian closed categories to λ-calculus, we will see further application of the natural transformations $G_i\tau$ and ηF_i.

8.2 The Yoneda Lemma

We now present a "well-known" lemma, *the Yoneda Lemma.* which has many applications, but which we will not attempt to motivate at this time.

Proposition 8.98 *Let* $\mathbf{C}$ *be a category with small hom sets (i.e.* $\mathbf{C}(X,Y)$ *is always a set), let* $F : \mathbf{C} \to \mathbf{Set}$ *be a functor, let* $c \in |\mathbf{C}|$, *and let* $\mathbf{Nat}(\mathbf{C}(c, \_), F)$ *denote the set of natural transformations from the hom-functor* $\mathbf{C}(c, \_)$ *to* F. *Then there is a bijection*

$$\begin{gathered} \phi : \mathbf{Nat}(\mathbf{C}(c, \_), F) \cong F(c) \\ \eta \mapsto \eta_c(1_c). \end{gathered}$$

Proof Let $\eta : \mathbf{C}(c, \_) \to F$ be a natural transformation. Then, for every $s \in |\mathbf{C}|$, η_X is a mapping from the set of morphisms $\mathbf{C}(c, x)$ to the set $F(x)$. Given $f \in \mathbf{C}(c, x)$ we get the the following diagram:

$$\begin{tikzcd} \mathbf{C}(c,c) \arrow[r, "\eta_c"] \arrow[d, "\mathbf{C}(c,\_)(f)"'] & F(c) \arrow[d, "F(f)"] \\ \mathbf{C}(c,x) \arrow[r, "\eta_x"'] & F(x) \end{tikzcd}$$

which commutes because η is a natural transformation. Going down and across we have

$$1_c \mapsto 1_c \bullet f = f \mapsto \eta_x(f)$$

going across and down we have

$$1_c \mapsto \eta_c(1_c) \mapsto F(f)(\eta_c(1_c)$$

so $\eta_x(f) = F(f)(\eta_c(1_c)$. That is, η is completely determined by $\eta_c(1_c)$.

Conversely, given $y \in F(x)$, we get a natural transformation $\phi : \mathbf{C}(c, \_) \Rightarrow F$ where, for each $x \in |\mathbf{C}|$

$$\begin{aligned} \phi_x : \mathbf{C}(c, x) &\to F(x) \\ f &\mapsto F(f)(y). \end{aligned}$$

To see that ϕ is a natural transformation let $f : x \to z$ and $g \in \mathbf{C}(c, x)$, then $(F(f) \bullet \phi_x)(g) = F(f)(F(g)(y)) = F(f \bullet g)(y) = \phi_y(f \bullet g) = \phi_y(\mathbf{C}(c, f)(g)) = (\phi_y \bullet \mathbf{C}(c, \_)(f))(g)$, as required. Finally, we have

$$\phi_c(1_c) = F(1_c) \bullet y = 1_{F(c)} \bullet y = y,$$

and thus the mapping

$$\begin{aligned} \mathbf{Nat}(\mathbf{C}(c, \_),\ F) &\to F(c) \\ \eta &\mapsto \eta_c(1_c) \end{aligned}$$

is an isomorphism. $\square$

9 Limits and Colimits

Limits and colimits are constructions that are found in many categories. They are always worth looking for since, in general, they will be interesting and useful constructions. In many cases they yield very fundamental constructs such as the addition and multiplication of natural numbers. Conversely, a category in which there are neither limits nor colimits may be of little interest just because it provides so little structure to play with.

9.1 Cones and Limits

Definition 9.99 Let $D : \mathbf{S} \to \mathbf{C}$ be a diagram in $\mathbf{C}$ of shape $\mathbf{S}$. By a *cone for D in $\mathbf{C}$* we mean an object $K \in |\mathbf{C}|$ together with a mapping

$$\begin{gathered} k : |\mathbf{S}| \to \mathbf{C} \\ X \mapsto (k_X : K \to D(X)) \end{gathered}$$

such that, for each $\alpha \in \mathbf{S}(X, Y)$, we have

$$D(\alpha) \bullet k_X = k_Y$$

If $\langle J, j\rangle$ and $\langle K, k\rangle$ are cones for D, then a *cone morphism* $\rho : \langle J, j\rangle \to \langle K, k\rangle$ is a morphism $\rho \in \mathbf{C}(J, K)$ such that for all $X \in |\mathbf{S}|$,

$$k_X \bullet \rho = j_X.$$

A cone $\langle K, k\rangle$ for diagram D is a *limiting cone for D* (or, for short, *a limit for D*) if for each cone $\langle J, j\rangle$ for D there exists a unique cone morphism $\rho = \rho_{\langle J,j\rangle} : \langle J, j\rangle \to \langle K, k\rangle$.

$$\begin{tikzcd}
 & & D(X) \arrow[dd, "D(\alpha)"] \\
J \arrow[r, "\rho"] \arrow[urr, "j_X"] \arrow[drr, "j_Y"'] & K \arrow[ur, "k_X"] \arrow[dr, "k_Y"'] & \\
 & & D(Y)
\end{tikzcd}$$

Given a limiting cone $\langle K, k\rangle$, we often refer to K as *the limit object*, and to the unique cone morphism $\rho_{\langle J,j\rangle} : \langle J, j\rangle \to \langle K, k\rangle$ as a *mediating morphism*, or *mediator.* □

We next give the definitions of a number of specific limits (namely, terminal objects, products, equalizers, and pullbacks and the dual (colimit) notions of initial objects, coproducts, coequalizers, and pushouts) together with some simple examples which indicate the ubiquity of the concepts. We also make reference to some of the computer science applications of limits and colimits.

Definition 9.100 Let $\mathbf{S} = \mathbf{0}$, then D must be the empty functor, and a cone for D is just an element of $\mathbf{C}$. But a limiting cone is then an object I of $\mathbf{C}$ such that for every object X in $\mathbf{C}$ there is exactly one morphism $\rho_X : X \to I$ in $\mathbf{C}$. An object with this property is called a *terminal object of* $\mathbf{C}$. □

Example 9.101 Any singleton set (set with exactly one element) is a terminal object in **Set**. A poset $\mathbf{P} = \langle P, \leq \rangle$, viewed as a category, has a terminal object iff it has a top element.

One approach to the algebraic specification of data types, sometimes refered to as terminal algebra semantics or "loose semantics" regards a specification (a presentation) as defining a category of algebras corresponding to the implementations of the type having the desired external behavior. The terminal algebra in this category of implementations generally corresponds to the "true type". For example, given a "loose specification" of the data type finite-sets-of-integers, the semantic category will include strings-of-integers and finite-bags-of-integers, and its terminal object will be the actual finite-sets-of-integers. □

Example 9.102 The cones for $D : \mathbf{S} \to \mathbf{C}$, together with their cone morphisms, form a category of cones for D. The limiting cones for D are exactly the terminal objects in this category. □

Definition 9.103 A *product* is a limit for a diagram whose shape is a discrete category. In particular, a *binary product* is a limit for a diagram of the form

$$\begin{array}{c} \bullet A \\ \\ \bullet B \end{array}$$

For such a binary product we generally write the limiting object as $A \times B$ and the limiting cone as

$$\langle \pi_A : A \times B \to \ A, \pi_B : A \times B \to B \rangle$$

$$\begin{tikzcd}
A \\
A \times B \arrow[u, "\pi_A"] \arrow[d, "\pi_B"'] \\
B
\end{tikzcd}$$

The morphisms π_A and π_B are called *projections*. A cone for A and B consists of an object C together with morphisms $f : C \to A$ and $g : C \to B$. The mediating morphism from C to $A \times B$ for such a cone is often written as $\langle f, g \rangle : C \to A \times B$. □

Example 9.104 The familiar Cartesian product of sets ($A \times B = \{\langle a, b\rangle \mid a \in A \; and \; b \in B\}$) is a example of a product in the category, **Set**, of sets. The category **Pfn**, of partial functions, also has products, but they are not the cartesian products, rather, given sets A and B their categorical product is $A \times B = A + (A \otimes B) + B$, where $\otimes$ denotes the Cartesian product of A and B and $+$ denotes disjoint union (see Definition 9.115, below).

Let **P** be a poset $\langle P, \leq \rangle$, viewed as a category (see Example 3.16), then the product $p \times q$ of elements $p, q \in P$ is, if it exists, exactly the greatest lower bound of p and q with respect to $\leq$.

Let **N** be as in Example 3.15, then for n, and p in $|\mathbf{N}|$, their product object is $n * p$, the arithmetic product of the natural numbers p and q. To see how this is so, identify n with $[n]$ and p with $[p]$, then the necessary projections $\pi_n : [n * p] \to [n]$, and $\pi_p : [n * p] \to [p]$, are defined using the isomorphism

$$\begin{aligned} [n] \times [p] &\to [n * p] \\ \langle i, j \rangle &\mapsto ((i-1) * p) + j. \end{aligned}$$

Products provide the appropriate abstract model for RECORDs as found in programming languages such as PASCAL. The product projections provide the operations for accessing the fields of the RECORD, and the mediating morphisms provide the operations for putting information into the RECORD. □

Definition 9.105 An *equalizer* is a limit for a diagram of the form

$$\begin{tikzcd} A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \end{tikzcd}$$

Thus an equalizer is a morphism $e : E \to A$ such that $f \bullet e = g \bullet e$, and for any $h : H \to A$ such that $f \bullet h = g \bullet h$, there exists a unique $\overline{h} : H \to E$ such that $e \bullet \overline{h} = h$.

$$\begin{tikzcd} E \arrow[r, "e"] & A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \\ H \arrow[u, "\overline{h}"] \arrow[ur, "h"'] & & \end{tikzcd}$$

□

Example 9.106 For two morphisms $f, g : A \to B$ in **Set**, an equalizer is the inclusion mapping from the subset $\{a \in A \mid f(a) = g(a)\}$ of A into A. □

Definition 9.107 A *pullback* is a limit for a diagram of the form

$$\begin{tikzcd}
 & A \arrow[d, "f"] \\
B \arrow[r, "g"'] & C
\end{tikzcd}$$

Thus the pullback is an object $P \in |\mathbf{C}|$ together with morphisms $\overline{f} : P \to B$ and $\overline{g} : P \to A$ such that $f \bullet \overline{g} = g \bullet \overline{f}$ and for each $Q \in |\mathbf{C}|$ and $f' : Q \to B$ and $g' : Q \to A$ such that $f \bullet g' = g \bullet f'$, there exists a unique $q : Q \to P$ such that $\overline{f} \bullet q = f'$ and $\overline{g} \bullet q = g'$.

$$\begin{tikzcd}
Q \arrow[dr, "q"] \arrow[drr, "g'"] \arrow[ddr, "f'"'] & & \\
 & P \arrow[r, "\overline{g}"'] \arrow[d, "\overline{f}"] & A \arrow[d, "f"] \\
 & B \arrow[r, "g"'] & C
\end{tikzcd}$$

□

Example 9.108 Let $f : A \to C$ and $g : B \to C$ be monomorphisms (injective mappings) in **Set**, then the pullback object P for f and g is, in effect, the intersection of $f(A)$ and $g(B)$.

Special pullbacks can be used to model Dijkstra's weakest preconditions in a comparatively general settings – see Wagner [19], and Manes [12]. □

Proposition 9.109 *Let $D : \mathbf{S} \to \mathbf{C}$ be a diagram in $\mathbf{C}$, then the limit of D is unique up to isomorphism. That is, if η_1 and η_2 are limiting cones for D with respective limit objects L_1 and L_2 then there is an isomorphism $\iota : L_1 \cong L_2$.*

Proof Because η_1 and η_2 are limiting cones it follows that there is a unique morphism $\iota_1 : L_1 \to L_2$ such that $(\eta_2)_S \bullet \iota_1 = (\eta_1)_S$ for each $S \in |\mathbf{S}|$, and there is a unique morphism $\iota_2 : L_2 \to L_1$ such that $(\eta_1)_S \bullet \iota_2 = (\eta_2)_S$ for each $S \in |\mathbf{S}|$. But then $\iota_1 \bullet \iota_2 : L_2 \to L_2$ is such that

$$(\eta_2)_S \bullet (\iota_1 \bullet \iota_2) = ((\eta_2)_S \bullet \iota_1) \bullet \iota_2 = (\eta_1)_S \bullet \iota_2 = (\eta_2)_S.$$

for each $S \in |\mathbf{S}|$. But, by the uniqueness property, this implies $\iota_1 \bullet \iota_2 = 1_{L_2}$. Similarly, $\iota_2 \bullet \iota_1 = 1_{L_1}$. Thus ι_1 is an isomorphism. □

A cone $\langle K, k \rangle$ can be looked at as a natural transformation. Given a diagram $D : \mathbf{S} \to \mathbf{C}$, let $\Delta_K : \mathbf{S} \to \mathbf{C}$ be the functor such that $\Delta_K(X) = K$, and $\Delta_K(\alpha) = 1_K$, for all $X \in |\mathbf{S}|$, and $\alpha \in \mathbf{S}$. Then $\langle K, k \rangle$ is just the natural transformation

$$
\begin{array}{c}
k : \Delta_K \Rightarrow D \\
X \mapsto (k_X : K \to D(X))
\end{array}
$$

i.e.,

$$X \mapsto (k_X : \Delta_K(X) \to D(X)).$$

Definition 9.110 A category $\mathbf{C}$ is said to be *complete* if every diagram $D : \mathbf{S} \to \mathbf{C}$ where $\mathbf{S}$ is small, has a limit; *finitely complete* if for every finite $\mathbf{S}$ (i.e., every category $\mathbf{S}$ with $Mor(\mathbf{S})$ a finite set), every diagram $D : \mathbf{S} \to \mathbf{C}$ has a limit. □

Proposition 9.111 *If* $\mathbf{C}$ *has binary products and equalizers then* $\mathbf{C}$ *is finitely complete.*

Proof Let $D : \mathbf{S} \to \mathbf{C}$ be a finite diagram in $\mathbf{C}$.

Let $Q = \prod\langle D(t(\alpha)) \mid \alpha \in \mathbf{S}\rangle$ (i.e. Q is the product of images, under D, of the targets of the arrows in $\mathbf{S}$) and, for each $\alpha \in \mathbf{S}$, let $q_\alpha : Q \to D(t(\alpha))$ be the corresponding product projection.

Let $P = \prod\langle D(X) \mid X \in |\mathbf{S}|\rangle$ (i.e., P is the product of the objects (vertices) in $\mathbf{S}$) and, for each $X \in |\mathbf{S}|$, let $p_X : P \to X$ be the corresponding product projection.

Then, by the definition of product, we have:

1. There exists a unique morphism $f : P \to Q$ such that $q_\alpha \bullet f = p_{t(\alpha)}$ for each $\alpha \in \mathbf{S}$.
2. There exists a unique morphism $g : P \to Q$ such that $q_\alpha \bullet g = D(\alpha) \bullet p_{s(\alpha)}$ for each $\alpha \in \mathbf{S}$.

Let $e : E \to P$ be an equalizer for f and g. Define

$$\mu = \langle \mu_X = p_X \bullet e \mid X \in |\mathbf{S}|\rangle$$

We claim that $\langle E, \mu\rangle$ is the desired limiting cone for D.

Now $\langle E, \mu\rangle$ is a cone for D since for every $\alpha : X \to Y \in \mathbf{S}$ we have

$$
\begin{array}{ll}
D(\alpha) \bullet \mu_X & \\
= & D(\alpha) \bullet p_X \bullet e \\
= & q_\alpha \bullet g \bullet e \\
= & q_\alpha \bullet f \bullet e \\
= & p_{t(\alpha)} \bullet e \\
= & \mu_Y
\end{array}
$$

Now, given any cone $\langle c, \tau\rangle$ for D we must show that there exists a unique morphism $\tau^\sharp : c \to E$ such that $\mu_X \bullet \tau^\sharp = \tau_X$ for every $X \in |\mathbf{S}|$.

Now, since P is the product of the $D(X)$, $X \in |\mathbf{S}|$, it follows that fori each cone $\langle c, \tau\rangle$ for D, there exists a unique morphism $\overline{\tau} : c \to P$ such that $p_X \bullet \overline{\tau} = \tau_X$ for each $X \in |\mathbf{S}|$. But then

$$q_\alpha \bullet f \bullet \overline{\tau} = p_{t(\alpha)} = \tau_Y$$

and

$$q_\alpha \bullet g \bullet \overline{\tau} = D(\alpha) \bullet p_{s(\alpha)} \bullet \overline{\tau} = D(\alpha) \bullet \tau_{s(\alpha)} = \tau_Y$$

so $f \bullet \overline{\tau} = g \bullet \overline{\tau}$ by the product property for Q. But then, by the equalizer property for e, there exist unique morphism $\tau^\sharp$ such that $e \bullet \tau^\sharp = \tau$. We claim this is the desired morphism. We first note that for every $X \in |\mathbf{S}|$

$$\mu_X \bullet \tau^\sharp = p_X \bullet e \bullet \tau^\sharp = p_X \bullet \overline{\tau} = \tau_X,$$

as required. It remains only to show that $\tau^\sharp$ has the necessary uniqueness property. So suppose that $h : c \to E$ such that $\mu_X \bullet h = \tau_X$ for all $X \in |\mathbf{S}|$, then we necessarily have $\mu_X \bullet h = p_X \bullet e \bullet h$ for all $X \in |\mathbf{S}|$ which implies $e \bullet h = \overline{\tau}$ by the uniqueness of $\overline{\tau}$, but then $e \bullet h = \overline{\tau}$ implies $h = \tau^\sharp$ by the uniqeness property of $\tau^\sharp$ with respect to e. □

9.2 Cocones and Colimits

By reversing all the arrows in the above discussion of cones and limits we get the dual notions of cocones and colimits.

Definition 9.112 Let $D : \mathbf{S} \to \mathbf{C}$ be a diagram in $\mathbf{C}$ of shape $\mathbf{S}$. By a *cocone for D in* $\mathbf{C}$ we mean an object $K \in |\mathbf{C}|$ together with with a mapping

$$\begin{gathered} k : |\mathbf{S}| \to \mathbf{C} \\ X \mapsto (k_X : D(X) \to K) \end{gathered}$$

such that, for each $\alpha \in \mathbf{S}(X, Y)$, we have

$$k_Y \bullet D(\alpha) = k_X.$$

If $\langle K, k \rangle$ and $\langle J, j \rangle$ are cocones for D, then a *cocone morphism* $\rho : \langle K, k \rangle \to \langle J, j \rangle$ is a morphism $\rho \in \mathbf{C}(K, J)$ such that for all $X \in |\mathbf{S}|$,

$$\rho \bullet k_X = j_X.$$

A cocone $\langle K, k \rangle$ for diagram D is a *colimiting cocone for D* if for each cocone $\langle J, j \rangle$ for D there exists a unique cocone morphism $\rho = \rho_{\langle J, j \rangle} : \langle K, k \rangle \to \langle J, j \rangle$.

$$\begin{tikzcd}
D(X) \arrow[dd, "D(\alpha)"'] \arrow[dr, "k_X"] \arrow[drr, "j_X"] & & \\
& K \arrow[r, "\rho"] & J \\
D(Y) \arrow[ur, "k_Y"'] \arrow[urr, "j_Y"'] & &
\end{tikzcd}$$

□

Definition 9.113 Let $\mathbf{S} = \mathbf{0}$, then D must be the empty functor, and a cocone for D is just an element of $\mathbf{C}$. But a colimiting cocone is then an object I of $\mathbf{C}$ such that for every object X in $\mathbf{C}$ there is exactly one morphism $\rho_X : I \to X$ in $\mathbf{C}$. An object with this property is called an *initial object of* $\mathbf{C}$. □

Example 9.114 The empty set, $\emptyset$, is initial in **Set**; the empty string, λ, is initial in $\mathbf{Str}_S$. A poset, viewed as a category, has an initial object iff it has a bottom object.

While initiality is an extremely simple concept it has played a surprisingly important role in computer science, namely in the algebraic specification of data types (e.g. see [5]), and in denotational and/or initial algebra semantics for programming languages (e.g. see [7]). Briefly, a context free grammar for a programming language gives rise to a corresponding initial algebra with an operation for each production in the grammar. Each possible semantics for the language then corresponds to a homomorphism from this initial algebra to an appropriate semantic algebra with the same signature. □

Definition 9.115 A *coproduct* is a colimit for a diagram whose shape is a discrete category. In particular, a *binary coproduct* is a colimit for a diagram of the form

$$\begin{array}{c} \bullet A \\ \\ \bullet B \end{array}$$

For such a binary coproduct we generally write the colimiting object as $A+B$ and the colimiting cone as

$$\langle \iota_A : A \to A+B,\ \iota_B : B \to A+B \rangle$$

$$\begin{tikzcd} A \arrow[d, "\iota_A"'] \\ A+B \\ B \arrow[u, "\iota_B"] \end{tikzcd}$$

The morphisms ι_A and ι_B are called *injections.* A cocone for A and B consists of an object C together with morphisms $f : A \to C$ and $g : B \to C$. The mediating morphism from $A+B$ to C for such a cocone is often written as $[f,g] : A+B \to C$.

It is also quite common to write $\kappa_{A,B}$ for ι_A, and $\lambda_{A,B}$ for ι_B. □

Fact 9.116 *The category* $\mathbf{Str}_S$ *has coproducts, indeed, given strings* $u : [p] \to S$ *and* $v : [q] \to S$*, then their coproduct object is the string*

$$uv = u+v : [p+q] \to S$$
$$uv(i) = IF\ i \leq p\ THEN\ u(i)\ ELSE\ v(i-p)$$

The coproduct injections are then $\langle u, \iota_u, u+v \rangle$ *and* $\langle v, \iota_v, u+v \rangle$ *where*

$$\iota_u : [p] \to [p+q]$$
$$i \mapsto i$$

$$\begin{aligned} \iota_v : [q] &\to [p+q] \\ i &\mapsto p+i \end{aligned}$$

If $f : u \to w$ and $g : v \to w$ where $w : [n] \to S$, then the mediating morphism for f and g is $\langle u+v, [f,g], w\rangle$ where

$$\begin{gathered} [f,g] : [p+q] \to [n] \\ i \mapsto f(i) \textit{ for } i \leq p \\ i \mapsto g(i-p) \textit{ for } i > p. \end{gathered}$$

If $f : u \to s$ and $g : v \to t$ where $s : [n] \to S$ and $t : [m] \to S$, then $(f+g)$ is given by the mapping

$$\begin{gathered} (f+g) : [p+q] \to [n+m] \\ (f+g)(i) = IF\ i \leq p\ THEN\ f(i)\ ELSE\ g(i-p). \end{gathered}$$

The string $u+v$ is generally called the concatenation of u and v.

Example 9.117 In the category **Set** of sets, the coproduct of two sets, A and B, is their disjoint union, $(A \times \{0\}) \cup (B \times \{1\})$. The coproducts in **Pfn** are the same as those in **Set**.

Let $\mathbf{P}$ be a poset $\langle P, \leq\rangle$, viewed as a category, then the coproduct $p+q$ of elements $p, q \in P$, if it exists, is exactly the least upper bound of p and q with respect to $\leq$.

Let $\mathbf{N}$ be as in Example 3.15, then for n and p in $|\mathbf{N}|$, their coproduct is precisely $n+p$, the arithmetic sum of the natural numbers n and p.

Coproducts provide an abstract model for the concept of a tagged VARIANT, that is a type, somewhat similar to the variant records of PASCAL or the union types of ALGOL68, which can, in effect, hold an object of any one of a number of designated types, and where the object is "tagged" to indicate its origin. The coproduct injections provide means for inserting an object into the VARIANT, and the mediating morphisms correspond to case-statements for accessing the VARIANT. The coproduct concept of VARIANT is more general than the union concept of ALGOL68 and it is type-safe in contrast to the variant record concept of PASCAL. □

Definition 9.118 A *coequalizer* is a colimit for a diagram of the form

$$\begin{tikzcd} A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \end{tikzcd}$$

Thus a coequalizer is a morphism $e : B \to C$ such that $e \bullet f = e \bullet g$, and for any $h : B \to H$ such that $h \bullet f = h \bullet g$, there exists a unique $\overline{h} : C \to H$ such that $\overline{h} \bullet e = h$.

$$\begin{tikzcd} A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \arrow[r, "e"] \arrow[dr, "h"'] & E \\ & & H \arrow[u, "\overline{h}"'] \end{tikzcd}$$

□

Example 9.119 Given two functions $f, g : A \to B$ in **Set**, their coequalizer object is the set $B/\equiv$ of $\equiv$-equivalence classes of B, where $\equiv$ is the smallest equivalence relation on B containing the relation $\{\langle f(a), g(a)\rangle \mid a \in A\}$, and the coequalizer morphism $e : B \to B/\equiv$ takes each $b \in B$ to its $\equiv$-equivalence class.

Coequalizers can be used to capture the essense of such quotient constructions in cases where the usual equivalence class and congruence class constructions are not available. For example, coequalizers are needed to carry out various constructions on the continuous posets and algebras used in fixed point semantics, where the familiar equivalence class constructions are not enough to ensure the existence of the required least upper bounds. $\square$

Definition 9.120 A *pushout* is a colimit for a diagram of the form

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "g"'] & C \\ B & \end{tikzcd}$$

Thus the pushout is an object $P \in |\mathbf{C}|$ together with morphisms $\overline{f} : B \to P$ and $\overline{g} : C \to P$ such that $\overline{g} \bullet f = \overline{f} \bullet g$ and for each $Q \in |\mathbf{C}|$ and $f' : B \to Q$ and $g' : C \to Q$ such that $g' \bullet f = f' \bullet g$, there exists a unique $q : P \to Q$ such that $q \bullet \overline{g} = g'$ and $q \bullet \overline{f} = f'$. $\square$

Example 9.121 Many authors have made use of pushouts in treating the actualization of parameters in studies of the algebraic specification of parameterized data types.

Pushouts also provide an appropriately abstract approach to the declaration of variables and the creation of pointers, see [20]. $\square$

Proposition 9.122 *Let $D : \mathbf{S} \to \mathbf{C}$ be a diagram in $\mathbf{C}$, then the colimit of D is unique up to isomorphism. (See Proposition 9.109.)*

Definition 9.123 A category $\mathbf{C}$ is said to be *cocomplete* if every diagram $D : \mathbf{S} \to \mathbf{C}$, where $\mathbf{S}$ is small, has a colimit; *finitely cocomplete* if for every $\mathbf{S}$ with $|\mathbf{S}|$ a finite set every diagram $D : \mathbf{S} \to \mathbf{C}$ has a colimit. $\square$

Proposition 9.124 *If $\mathbf{C}$ has binary coproducts and coequalizers then $\mathbf{C}$ is finitely cocomplete.*

9.3 Some Properties of Limits and Colimits

Proposition 9.125 *Let $f : X \to Y$ and $g : X \to Z$ where f is an epimorphism. If $\langle f' : Y \to W,\ g' : Z \to W\rangle$ be a pushout for $\langle f, g\rangle$ then f' is an epimorphism.*

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[d, "g"'] & Y \arrow[d, "g'"] \\ Z \arrow[r, "f'"] & W \end{tikzcd}$$

10 Records, Variants, and Recursive Types

We will use products and coproducts in order to build up another approach to data types. The mathematical framework will also give us an approach to basic control constructs such as If-Then-Else and While-Do. We begin by looking at the programming constructs: Records and Variants. We show that their mathematical counterparts are products and coproducts. When we formulate these ideas precisely we see that we can easily extend them to include recursive types such as Trees, Natural Numbers, Stacks, etc. These types are all definable in terms of recursion equations in $\times$ and $+$ (product and coproduct). Furthermore the product projections and coproduct injections correspond to the key operations associated with the types (e.g., for Stacks they provide the operations Pop, Top, Push and Empty). Going on step further we find that the mediating morphisms for the products and coproducts can be exploited to give the basic control constructs.

10.1 Records and Variants

We begin with an informal look at records and variants.

Consider the concept of a record in a programming language. The syntax of the declaration of a record type RITEM would look something like the following

```
RITEM = RECORD
          field_1 :  type_1
          field_2 :  type_2
                 ⋮
          field_n :  type_n
END.
```

The field_names are all different, the types may be the same.

If we declare a record X of type RITEM then we can access and, say, print its ith field, using code resembling the following

```
Print X.field_i;
```

Given n functions (expressions) $f_1, \ldots, f_n$ all of arity w and with $f_i : w \rightarrow$type_i, then there is an insert operation

$$\langle f_1, \ldots, f_n \rangle : w \rightarrow \mathtt{RITEM}$$

that uses the f_i to compute "new values for the fields of the record".

Conventionally, we only use the special case where f_i=field_i for all but one $i \in \{1, \ldots, n\}$, say j, and we write the insertion as

$$\mathtt{A.field\_j} := f_j$$

As the above description of records may suggest, we can look at records as products. In terms of the above declaration, if A_i is the set of all objects of type_i then $A_{RITEM} = A_1 \times \ldots \times A_n$, the ith access operation is the ith projection function, and the insert operations correspond to the product mediators.

Now for a look at variants. Variants may not be a programming concept with which you are familiar. The variant-records of PASCAL, and the union types of ALGOL'68 are related to variants but have significant differences. The syntax for the declaration of a variant type VITEM would look something like the following

```
VITEM = VARIANT
          tag_1 :  type_1
          tag_2 :  type_2
            ⋮
          tag_n :  type_n
END.
```

Where the tag_i's are all different but the types may be the same. The intuition is that variable X of type VITEM can hold a value of any of the types type_i and that its contents will always be "tagged" so one "can tell which variant it is". In keeping with this, there is an operation for each $i = 1, \ldots, n$ for inserting an object of type_i into an object VITEM and giving it tag tag_i. Typical programming notation for the inserting the value of an expression exp into a variable X of sort VITEM would be

```
X.tag_i := exp
```

The accessing of an object of type VITEM is an n-way case statement where the case is determined by the tags. Such an access operation might have the following form

CASE VITEM by tag =
        tag_1 DO f_1
        tag_2 DO f_2
            ⋮
        tag_n DO f_n
END.

where each f_i is a function (expression) of arity type_i and returning a value of some type w. This kind of access operation is type safe, in contrast to the accessing of variant_records in PASCAL.

As the above informal description may suggest, variants can be represented as coproducts. In terms of the above declaration, if A_i is the set of all objects of type_i then $A_{VITEM} = A_1 + \ldots + A_n$, the coproduct injections are insert operations, and the access operations correspond directly to the coproduct mediators.

10.2 Recursive Types

We now want to expand the above to deal with recursive types, in particular recursive records and variants, the informal idea is given by the following recursive definition of the type TREES_OF_INT.

```
TREES_OF_INT = VARIANT
        leaf :  INT
        join :  RECORD
```

```
                    lft :  TREES-OF-INT
                    rgt :  TREES-OF-INT
                    END;
    END.
```

We now give the desired formal treatment

Definition 10.126 An *(R, V)-presentation*, $\mathbf{P} = \langle G, R, V, \Sigma, E, \mu\rangle$ consists of

G, a set (of *ground sorts*)

R, a set (of *record sorts*)

V, a set (of *variant sorts*)

where G, R, and V are pairwise disjoint, and writing S for $G \cup R \cup V$,

$\mu : (R \cup V) \to S^*$

Σ, an S-sorted signature

Ω, an S-sorted signature, where, for all $w \in S^*$, $s \in S$, $\Omega_{w,s} \subseteq \Sigma_{w,s}$, and

$$\begin{gathered}\Omega_{w,s} = \{\iota_{v,i} : \mu(v)_i \to v \mid v \in V, w = \mu(v)_i, s = v\} \\ \cup\{\pi_{r,i} : r \to \mu(r)_i \mid r \in R, w = r, s = \mu(r)_i\}\end{gathered}$$

E, a set of Σ-axioms

Then a $\mathbf{P}$*-(R, V)-algebra*, A, is an Σ-algebra where

1. For all $v \in V$, if $\mu(v) = s_1 \cdots s_n$ then

   $$A_v = A_{s_1} + \cdots + A_{s_n}$$

   and, for all $i = 1, \ldots, n$,

   $$(\iota_{v,i})_A : A_{s_i} \to A_v$$

   is the corresponding coproduct injection.

2. For all $r \in R$, if $\mu(r) = s_1 \cdots s_n$ then

   $$A_r = A_{s_1} \times \cdots \times A_{s_n}$$

   and, for all $i = 1, \ldots, n$,

   $$(\pi_{r,i})_A : A_r \to A_{s_i}$$

   is the corresponding product projection.

We say that an (R,V)-presentation $\mathbf{P} = \langle G, R, V, \Sigma, E, \mu\rangle$ is *pure* if $\Sigma_{w,s} = \emptyset$ for all $w \in S^*$ and $s \in S$.

separated if, for all $w \in S^*$, and $s \in S$, $\Sigma_{w,s} - \Omega_{w,s} \neq \emptyset$ only if $w \in G^*$, $s \in G$.

recursive if it has recursive types, where, defining $s \prec t$ for $s, t \in S$ iff $t \in \{\mu(s)_i \mid i \in [| \mu(s) |]\}$, and letting $\prec^*$ be the transitive closure of $\prec$, we define $s \in S$ to be a *recursive type* iff $s \prec^* s$.

elementary if there are no recursive types.

□

Theorem 10.127 *If* $\mathbf{P} = \langle G, R, V, \Sigma, E, \mu\rangle$ *is separated then for every* $\mathbf{P}' = \langle G, \Sigma - \Omega, E\rangle$*-algebra* A *there exists an initial* $\mathbf{P}$*-(R, V)-algebra* $\overline{A}$ *with* $\mathbf{P}'$*-reduct* A*. (* $\overline{A}$ initial *means that for every other* $\mathbf{P}$*-algebr* B *there is exactly on homomorphism* $h : \overline{A} \to B$*.) (The* $\mathbf{P}'$-reduct A of a $\mathbf{P}$-(R,V)-algebra B is the $\langle G, \Sigma - \Omega, E\rangle$-algebra you get by forgetting all the Ω-indexed operations in B.) That is, one can extend A to form a $\mathbf{P}$-algebra
$\overline{A}$ which is initial

Proof Define $\overline{A}_0, \overline{A}_1, \overline{A}_2, \ldots$ where $\overline{A}_0$ is such that $(A_0)_g = A_g$, and $(\overline{A}_0)_r = \emptyset = (\overline{A})_v$ for all $r \in R$, $v \in V$, and where, given $\overline{A}_n$ we define $\overline{A}_{n+1}$ to be such that

$$(A_{n+1})_g = A_g$$

$$(A_{n+1})_v = (\overline{A}_n)_{\mu(v)_1} + \ldots + (\overline{A}_n)_{\mu(v)_{|v|}}$$

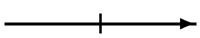
$$(A_{n+1})_r = (\overline{A}_n)_{\mu(r)_1} \times \ldots \times (\overline{A}_n)_{\mu(r)_{|r|}}$$

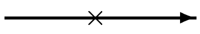
Now, for each $s \in G \cup R \cup V$, let $\overline{A}_s = \bigcup \langle (\overline{A}_n)_s \mid n \in \mathbf{N}\rangle$. It remains to show that the operations in Ω are well-defined. □

The general cases are more complicated. The requirements of Σ and E can conflict with those of R, V and μ. For example, if $G = \{g_1, g_2\}$, $R = \emptyset$, $V = \{v\}$, $\Sigma_{\lambda,v} = \{\sigma\}$ then there is no initial (R,V)-algebra, A, as there is no canonical choice for σ_A. That is, σ_A must be in $A_{g_1} + A_{g_2}$ and so must come from either A_{g_1} or from A_{g_2}, but not from both. Indeed we can construct (R,V)-algebras A_1 and A_2 with this presentation where $\sigma_{A_1} \in (A_1)_{g_1}$ and $\sigma_{A_2} \in (A_2)_{g_2}$ respectively. But if there existed an initial (R,V)-algebra, I, with this presentation, then σ_I would, say, come from I_{g_1} and there would be no (R,V)-homomorphism $h : I \to A_2$, contradicting the definition of initiality. We can also show that there are problems with elements of R, in the general case.

Later we will show examples of (R,V)-presentations that are not separated but have well-defined initial algebras. We now look at a number of examples of pure, recursive (R,V)-presentations.

Notation: In the diagrams accompanying the following examples we will use special arrows to denote injections and projections:

 for injections

 for projections

Example 10.128 The natural numbers may be specified as follows:

$G = \emptyset$

$R = \{\mathbf{1}\}$

$V = \{NAT\}$

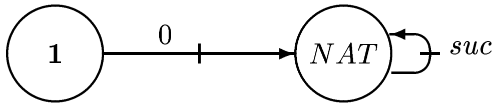
$\mu(\mathbf{1}) = \lambda$ (the empty string)

$\mu(NAT) = \mathbf{1} \cdot NAT$

The product of no sets is always a one point set - the terminal object in the category **Set** of sets and total functions. The above presentation corresponds to the recursive definition

$$NAT \cong \mathbf{1} + NAT$$

with injections $0 = \iota_{NAT,1} : \mathbf{1} \to NAT$, $suc = \iota_{NAT,2} : NAT \to NAT$.



□

Example 10.129 Let D be any set, then binary trees with leaves labeled by elements of D, can be specified by the following (R,V)-presentation

$G = \{D\}$

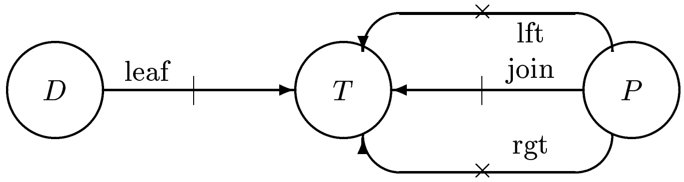
$R = \{P\}$ (P for pairs of trees)

$V = \{T\}$

$\mu(T) = D \cdot P$

$\mu(P) = T \cdot T$.

This corresponds to the specification, $T = D + (T \times T)$, the projections and injections are indicated in the following diagram.



□

Example 10.130 Here is a presentation for stack (or lists) of D:

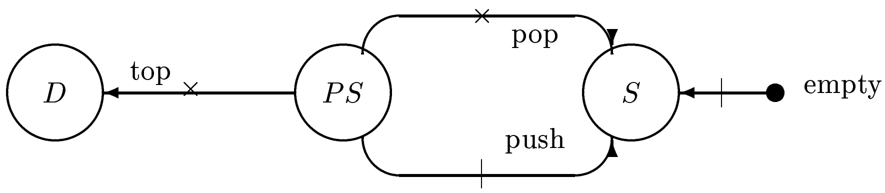
$G = \{D\}$

$R = \{PS, \mathbf{1}\}$ (PS for positive-stack = non-empty stack)

$V = \{S\}$

$\mu(\mathbf{1}) = \lambda,\ \mu(PS) = D \cdot S,\ \mu(S) = \mathbf{1} \cdot PS$

Or, equationally, $S = \mathbf{1} + (D \times S)$. The projections and injections are as indicated in the following diagram

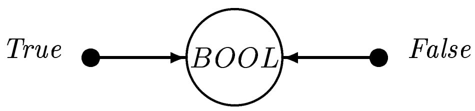


Note that we have not put in a circle for $\mathbf{1}$, but have just represented it as a dot. We will use this convention in the remaining examples. □

Example 10.131 Any finite set with named elements can be specified as a variant. Two useful examples are BOOL and CHAR. BOOL has presentation

$$BOOL = \mathbf{1} + \mathbf{1}$$

with injections *True* and *False*.



The definition of CHAR depends on what set of CHARacters is desired. If what is wanted is only capital Roman letters, then

$$CHAR = \underbrace{\mathbf{1} + \mathbf{1} + \ldots + \mathbf{1}}_{26}$$

with injections `A` through `Z`. □

11 Control Constructs via Products and Coproducts

11.1 Control Constructs

We start with a Lemma.

Lemma 11.132 *In* **Set**, *for all* $A, B, C \in |\,\mathbf{Set}\,|$,

$$(A \times C) + (B \times C) \cong (A + B) \times C$$

where, if $A + B$ *has coproduct injections*

$$A \xrightarrow{\iota_1} A + B \xleftarrow{\iota_2} B$$

then $(A + B) \times C$ *is the coproduct with injections*

$$A \times C \xrightarrow{\iota_1 \times 1_C} (A + B) \times C \xleftarrow{\iota_2 \times 1_C} B \times C$$

and the claimed isomorphism is:

$$[\iota_1 \times 1_C,\ \iota_2 \times 1_C] : (A \times C) + (B \times C) \cong (A + B) \times C$$

We present the basic ideas here by means of a series of examples:

Example 11.133 Conditionals correspond to coproduct mediators. For example, we can define the negation operation on $BOOL$ as the coproduct mediator $neg = [False, True]$

$$\begin{tikzcd}
\mathbf{1} \arrow[d, "True"'] \arrow[dr, "False"] & \\
BOOL \arrow[r, "neg"] & BOOL \\
\mathbf{1} \arrow[u, "False"] \arrow[ur, "True"'] &
\end{tikzcd}$$

A slightly more complex example,

that makes use of the lemma given above, is the definition of the and operation, and : $BOOL \times BOOL \to BOOL$ by means of the following

$$\begin{tikzcd}
\mathbf{1} \times BOOL \arrow[d, "True \times 1_{BOOL}"'] \arrow[dr, "\pi_2"] & \\
BOOL \times BOOL \arrow[r, "and"] & BOOL \\
\mathbf{1} \times BOOL \arrow[u, "False \times 1_{BOOL}"] \arrow[ur, "False \bullet \pi_1"'] &
\end{tikzcd}$$

What we have done above is define

$$and = [\pi_2,\ False \bullet \pi_1]\ ,$$

making use of the fact that $BOOL \cong \mathbf{1}+\mathbf{1}$, and so, by the above lemma, $BOOL \times BOOL \cong (\mathbf{1} \times BOOL) + (\mathbf{1} \times BOOL)$. The π_1 and π_2 are the projections for the product $\mathbf{1} \times BOOL$.

$$\mathbf{1} \xleftarrow{\ \pi_1\ } \mathbf{1} \times BOOL \xrightarrow{\ \pi_2\ } BOOL$$

We close this example by showing how to define the predecessor function, $pred : NAT \to NAT$ using a coproduct mediator.

$$\begin{tikzcd}
\mathbf{1} \arrow[d, "0"'] \arrow[dr, "0"] & \\
NAT \arrow[r, "pred"] & NAT \\
NAT \arrow[u, "suc"] \arrow[ur, "1_{NAT}"'] &
\end{tikzcd}$$

That is, $pred = [0, 1_{NAT}]$ □

In order to get the full power of a programming language we need some kind of iteration operation. The next example shows how to get WHILE-DO, and is followed by fundamental theorem on the WHILE-DO construct. The example after that indicates how to do primitive recursion on recursive data types.

Example 11.134 We capture WHILE-DO as a fixpoint of a coproduct diagram. The fixpoint is actually in the category **Pfn** of sets of partial functions. This causes some problems since, for starters, the above lemma does not hold in **Pfn** using the categorical product from **Pfn**. However, this difficulty can be overcome by using the Cartesian product (the product from **Set**), then the lemma holds and we can proceed as before (except that if we attempt to form a mediator for functions$f_1, \ldots, f_n$ using the Cartesian product then the resulting "quasi-mediator" will be undefined for any argument for which any of the f_i are undefined).

To give the idea of what we have in mind we first give a corresponding program in a PASCAL-like language:

```
Funct Add(A, B:NAT):NAT
      WHILE A≠ 0 DO
           A := PRED(A) /* A := A-1 */
           B := SUCC(B) /* B := B+1 */
      OD;
      RETURN B;
END.
```

Now, making using of the fact that $NAT \cong \mathbf{1}+NAT$ implies $NAT\times NAT \cong (\mathbf{1}\times NAT)+(NAT \times NAT)$ we get the following fixpoint diagram for $add : NAT \times NAT \to NAT$. It is a fixpoint diagram because the function W occurs in its own definition, that is, we are asserting that

$$W = [0 \times 1_{NAT},\ W \bullet (1_{NAT} \times Succ)]$$

and that

$$add = \pi_2 \bullet W$$

Here is the diagram:

$$\begin{tikzcd}
\mathbf{1} \times NAT \arrow[d, "0 \times 1_{NAT}"'] \arrow[dr, "0 \times 1_{NAT}"] & & \\
NAT \times NAT \arrow[r, "W"] & NAT \times NAT \arrow[r, "\pi_2"] & NAT \\
NAT \times NAT \arrow[u, "Suc \times 1_{NAT}"] \arrow[ur, "W \bullet (1_{NAT} \times Suc)"'] & &
\end{tikzcd}$$

$\square$

Theorem 11.135 *Let $C = A + B$ in* **Pfn***, and let $f : B \to C$, then there exists a (possibly partial) function $W : C \to C$ such that*

$$W = [\iota_A, W \bullet f]$$

$$\begin{tikzcd}
A \arrow[d, "\iota_A"'] \arrow[dr, "\iota_A"] & \\
C \arrow[r, "W"] & C \\
B \arrow[u, "\iota_B"] \arrow[ur, "W \bullet f"'] &
\end{tikzcd}$$

Indeed, for all $c \in C$, if there exists $n \geq 0$ such that $[\iota_A, f]^n(c) \in \iota_A(A)$ and for all $j < n$, $[\iota_A, f]^j(c) \in \iota_B(B)$, then $W(c) = [\iota_A, f]^n(c)$. Furthermore, if $f = g \bullet \iota_A$, $g : C \to C$, then $W(c) = g^n(c)$. (Where, for any $h : C \to C$ we define $h^0(c) = c$, and $h^{n+1}(c) = h(h^n(c))$.)

Proof If $W = [\iota_A, W \bullet f]$, then $W \bullet \iota_A = \iota_A$, and $W \bullet \iota_B = W \bullet f$, so

$$W = [W \bullet \iota_A, W \bullet f] = W \bullet [\iota_A, f] = W \bullet [\iota_A, f]^k$$

for all $k \geq 0$. Now if $[\iota_A, f]^n(c) \in \iota_A(A)$ then there exists $a \in A$ such that $\iota_A(a) = [\iota_A, f]^n(c)$ so

$$W(c) = (W \bullet [\iota_A, f]^n)(c) = W(\iota_A(a)) = (W \bullet \iota_A)(a) = \iota_A(a) = [\iota_A, f]^n(c)$$

Now assume $f = g \bullet \iota_A,\ g : C \to C$. Then we prove the desired result by induction on n. If $n = 0$ then, by definition

$$[\iota_A, f]^0 = c = g^0(c).$$

Now say that we have shown that for all $j \le k < n$ that $[\iota_A, f]^j(c) = g^k(c)$. Then $k < n$ implies that $g^k(c) \in \iota_B(B)$ so $g^k(c) = \iota_B(b)$ for some $b \in B$. But then

$$[\iota_A, f]^{k+1}(c) = [\iota_A, f](g^k(c)) = [\iota_A, f](\iota_B(b)) = f(b) = g(\iota_B(b)) = g(g^k(c)) = g^{k+1}(c).$$

□

We can also use fixpoint constructions on coproduct mediators to define a form of primitive recursion for arbitrary recursive types. We will not give the general construction, but we hope that the following examples will suggest what we have in mind.

Example 11.136 Addition of natural numbers is commonly defined by the following primitive recursive definition

add(0, y) = y

add(suc(x), y) = suc(add(x, y))

We can express this as the fixpoint

$$add = [\pi_2, suc \bullet add]$$

which corresponds to the following diagram

$$\begin{tikzcd}
\mathbf{1} \times NAT \arrow[d, "0 \times 1_{NAT}"'] \arrow[dr, "\pi_2"] & \\
NAT \times NAT \arrow[r, "add"] & NAT \\
NAT \times NAT \arrow[u, "suc \times 1_{NAT}"] \arrow[ur, "suc \bullet add"'] &
\end{tikzcd}$$

Here is a second example, a primitive recursive definition of the operation that takes a tree with D-labled leaves (see Example 10.129) and reverses the order of the leaves. The fixpoint expression is

$$rev = [leaf, join \bullet \langle rev \bullet rgt, rev \bullet lft\rangle]$$

the diagram is

$$
\begin{tikzcd}
D \arrow[d, "leaf"'] \arrow[dr, "leaf"] & \\
T \arrow[r, "rev"] & T \\
P \arrow[u, "join"] \arrow[ur, "join \bullet \langle rev \bullet rgt,\ rev \bullet lft \rangle"'] &
\end{tikzcd}
$$

Note that this example makes use of the product mediator $\langle \ldots \rangle$ as well as the coproduct mediator $[\ \ldots\ ]$. $\square$

11.2 Simple Memory Model

We have used coproducts to capture the concepts of conditionals and iteration operations, now we will show how to use products to get a simple memory model with and assignment operation. This model is adequate to model a language such as BASIC but will not suffice for a somewhat more complex language such as PASCAL. For an approach to memory that is adequate for PASCAL-like languages see the next section.

The intuitive idea is simply to view the memory as a "large record" with a field for every possible memory variable. This means that the "variables" are pre-typed. That is, there are no declarations of variables because every variable, as in BASIC, comes with an associated type. This is done in BASIC by having different sets of symbols for different types of variables. Of course we view the memory as a large product with a component of appropriate type for each variable. Thus if $\{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}\}$ is the set of `INTEGER` variables, and $\{\mathtt{A,\ B}, \ldots, \mathtt{Z}\}$ is the set of `STRING` variables, then the corresponding product (record) would have 52 components (fields), 26 of type `INTEGER` and 26 of type `STRING`. We can think of the variables as names for the corresponding projection operations. Thus if `A` is a variable of type `INTEGER` and x is an element of the product then the value of the variable `A` in state x is $\mathtt{A}(x)$. An expression, such as the arithmetic expression

$$\mathtt{B} * (\mathtt{A} + 3)$$

is then interpreted, in state x, as

$$\mathtt{B}(x) * (\mathtt{A}(x) + 3)$$

Of course the usual programming convention is not to put in the state, but to have it be implicit. An assignment statement of the form

```
A := B*(A+3)
```

can be regarded as an abbreviation for a mediating morphism from the product to itself in which for every variable $\mathtt{X} \neq \mathtt{A}$ the corresponding component of the mediating morphism is the projection `X` itself, so that `A` is the only component that is actually modified.

11.3 Exception Handling

The framework provided by (R,V)-presentations and (R,V)-algebras can be used to describe certain kinds of exception handling. Intuitively, the concept of an exception that want to investigate is that of an exception being a way to handle problems such as division-by-zero, array-index-out-of-bounds, etc., in a way that somehow terminates the computation, or stops the computation after taking certain actions, or that even "undoes the last few steps of the computation". The approach is to equip each data type with an additional element for each possible kind of exception that can arise with an operation that is normally supposed to return a value of that data type. This can be done by using the coproduct (the variant), that is, given a type INTeger then to take care of division-by-zero we change INT to

$$\mathrm{INT} \xrightarrow{\text{normal}} \mathrm{INT}+\mathbf{1} \xleftarrow{\text{div\_by\_0}} \mathbf{1}$$

All the operations on INT can be transformed into given $add : INT \times INT \to INT$, we get $add! : (INT + \mathbf{1}) \times (INT \times \mathbf{1}) \to (INT \times \mathbf{1})$ by exploiting the isomorphism

$$(INT + \mathbf{1}) \times (INT \times \mathbf{1}) \cong (INT \times INT) + (INT \times \mathbf{1}) \times (\mathbf{1} \times INT) + (\mathbf{1} \times \mathbf{1})$$

and taking $add!$ to be

$$add! = [add, div\_by\_0 \bullet t_{INT\times\mathbf{1}}, div\_by\_0 \bullet t_{\mathbf{1}\times INT}, div\_by\_0 \bullet t_{\mathbf{1}\times\mathbf{1}}]$$

where, for any set X, $t_X : X \to \mathbf{1}$ is the mediating morphism from X to $\mathbf{1}$. Then we can define a division operation on INT with the property that division by zero has value *div_by_0*. For example, using the equational specification of INT given (Goguen, Thatcher, Wagner *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*), we can extend it to give the following conditional specification of division as an operation

$$div : (INT \times INT) \to (INT + \mathbf{1})$$
$$\text{“}div(n,p) = n/p\text{”}$$

$$eq(p,0) = True \Rightarrow div(n,p) = div\_by\_0$$

$$(eq(p,0) = False \wedge (p \leq n) = True) \Rightarrow div(n,p) = 0$$

$$(eq(p,0) = False \wedge (p \leq n) = False) \Rightarrow div(n,p) = suc(div(n-p,p))$$

Given this definition we can immediatedly transform it into a function

$$div! : (INT + \mathbf{1}) \times (INT \times \mathbf{1}) \to (INT \times \mathbf{1})$$

So far we have only introduced the "exception" – we have not shown how to exploit it. The point is that we can, in effect, use the coproduct property to "jump" over, or out-of, the rest of the computation.

12 Adjoints

We come now to the final concept of "elementary category theory", the notion of an adjoint. We start by giving a definition which is somewhat abstruse. The reader may find the version given in Proposition 12.139 somewhat easier to comprehend. The advantage of the definition given here is that all the data is presented explicitly.

Definition 12.137 Let $U : \mathbf{D} \to \mathbf{C}$ be a functor, and let $\mathbf{1_C} : \mathbf{C} \to \mathbf{C}$ be the identity functor on $\mathbf{C}$. We say that a functor $F : \mathbf{C} \to \mathbf{D}$ is *a left adjoint for* U if there exists a natural transformation $\eta : \mathbf{1_C} \Rightarrow U \bullet F$, such that, for each choice of objects $X \in |\mathbf{C}|$, and $Y \in |\mathbf{D}|$, and morphism $\alpha \in \mathbf{C}(X, U(Y))$ there is a unique morphism $\overline{\alpha} \in \mathbf{D}(F(X), Y)$ such that

$$U(\overline{\alpha}) \bullet \eta_X = \alpha.$$

$$\begin{tikzcd}
X \arrow[r, "\eta_X"] \arrow[dr, "\alpha"'] & U(F(X)) \arrow[d, "U(\overline{\alpha})"] & F(X) \arrow[d, "\overline{\alpha}"] \\
 & U(Y) & Y
\end{tikzcd}$$

The natural transformation η is called the *unit* of the adjunction. We also say, under these conditions, that *U is a right adjoint for F*. As a shorthand, we write $F \dashv U$ for F is a left adjoint for U (or U is a right adjoint for F). See Proposition 12.141, below, for an alternative definition of right adjoint. □

We can establish the existence of an adjoint under somewhat weaker conditions than the above definition might suggest – in particular, as the following result makes precise, we don't need to "know in advance" that F is a functor or that η is a natural transformation. We first need a definition.

Definition 12.138 Let $U : \mathbf{C} \to \mathbf{D}$ be a functor, and let $D \in |\mathbf{D}|$, then a *universal arrow from D to U* consists of an object $C \in |\mathbf{C}|$ and a morphism $\eta_D : D \to U(C)$ with the property that, for each $X \in |\mathbf{C}|$, and $f : D \to U(X)$ there is a unique $\overline{f} : C \to X$ such that $U(\overline{f}) \bullet \eta_D = f$.

$$\begin{tikzcd}
D \arrow[r, "\eta_D"] \arrow[dr, "f"'] & U(C) \arrow[d, "U(\overline{f})"] & C \arrow[d, "\overline{f}"] \\
 & U(X) & X
\end{tikzcd}$$

Dually, a *universal arrow from U to D* consists of an object $C \in |\mathbf{C}|$ and a morphism $\varepsilon_D : U(C) \to D$ with the property that, for each $X \in |\mathbf{C}|$ and $g : U(X) \to D$ there is a unique $\overline{g} : X \to C$ such that $\varepsilon_D \bullet U(\overline{g}) = g$.

$$
\begin{tikzcd}
D & & U(C) & C \\
& & U(X) \arrow[llu, "g"] \arrow[u, "U(\overline{g})"'] & X \arrow[u, "\overline{g}"']
\arrow[from=1-3, to=1-1, "\varepsilon_D"']
\end{tikzcd}
$$

□

Proposition 12.139 *Let $U : \mathbf{C} \to \mathbf{D}$ be a functor. If for each $X \in |\mathbf{D}|$, there exists an object $\overline{X} \in |\mathbf{C}|$, and a morphism $\eta_X : X \to U(\overline{X}))$, such that $\langle \overline{X},\ \eta_X \rangle$ is a universal arrow from X to U, then there exists a functor $F : \mathbf{D} \to \mathbf{C}$ which is left adjoint to U and such that, for each $X \in |\mathbf{D}|$, $|F|(X) = \overline{X}$, and, for each $\alpha : X \to Y$ in $\mathbf{D}$, $F(\alpha)$ is the unique arrow $\overline{\alpha} : \overline{X} \to \overline{Y}$ such that $U(\overline{\alpha}) \bullet \eta_X = \eta_Y \bullet \alpha$.*

$$
\begin{tikzcd}
X \arrow[r, "\eta_X"] \arrow[d, "\alpha"'] & U(\overline{X}) = U(F(X)) \arrow[d, "U(\overline{\alpha}) = U(F(\alpha))"] & \overline{X} = F(X) \arrow[d, "\overline{\alpha} = F(\alpha)"] \\
Y \arrow[r, "\eta_Y"'] & U(\overline{Y}) = U(F(Y)) & \overline{Y} = F(Y)
\end{tikzcd}
$$

Note that the above square establishes exactly the condition needed for $\eta : \mathbf{1} \Rightarrow U \bullet F$, to be a natural transformation.

Example 12.140 Given a category $\mathbf{C}$ we make it into a graph by just forgetting the composition operation and the identity morphisms. It is easy to see that such forgetting gives us a functor $U : \mathbf{Cat} \to \mathbf{Graph}$. Conversely, given a graph G we can construct a category from it by forming its path category $\mathbf{Path}_G$. This construction gives us a functor $F : \mathbf{Graph} \to \mathbf{Cat}$. We claim that F is a left adjoint for U. To see this, let G be a graph, let $F(G)$ be the corresponding path category, and define $\eta_G : G \to U(F(G))$ be the obvious inclusion mapping (recall that $F(G)$ is formed from G by adding "additional arrows" corresponding to the paths in G, so $F(G)$, and thus $U(F(G))$, has the same vertices as G, and contains all the edges of G as paths of length one). Now given any small category $\mathbf{C}$ and any graph morphism $f : G \to U(C)$ we need to show that there is a unique extension of f to a functor $\overline{f} : F(G) \to \mathbf{C}$ such that $U(\overline{f}) \bullet \eta_G = f$. But that means only that we have to extend f to the paths on G, and that is completely straight forward: given a path $p : [n] \to E$, take $\overline{f}(p) = f(p(n)) \bullet \ldots \bullet f(p(1))$. □

The above definitions of adjoints can be restated in a dual version that emphasizes the right, rather than the left, adjoint.

Proposition 12.141 *Let $U : \mathbf{D} \to \mathbf{C}$, and let $F : \mathbf{C} \to \mathbf{D}$, then U is a right adjoint for F iff, where $\mathbf{1}_D : \mathbf{D} \to \mathbf{D}$ is the identity functor on $\mathbf{D}$, we have a natural transformation $\varepsilon : F \bullet U \Rightarrow 1_D$ (the* counit of the adjunction*) such that for any $Y \in |\mathbf{D}|$, $X \in |\mathbf{C}|$, and morphism $\beta \in \mathbf{D}(F(X), Y)$, there is a unique morphism $\overline{\beta} \in \mathbf{C}(X, U(Y))$ such that*

$$\varepsilon_Y \bullet F(\overline{\beta}) = \beta.$$

$$\begin{tikzcd}
Y & & F(U(Y)) \arrow[ll, "\varepsilon_Y"'] & U(Y) \\
& & F(X) \arrow[ull, "\beta"] \arrow[u, "F(\overline{\beta})"'] & X \arrow[u, "\overline{\beta}"']
\end{tikzcd}$$

Proof We give the "only-if" part of the proof. Assume $F \dashv U : \mathbf{D} \to \mathbf{C}$ with unit η. For $Y \in |\mathbf{D}|$ define $\varepsilon_Y : F(U(Y)) \to Y$ to be the unique morphism given by the adjunction such that $U(\varepsilon_Y) \bullet \eta_{U(Y)} = 1_{U(Y)}$. Given $g : F(X) \to Y$, let $\overline{g} = U(g) \bullet \eta_X : X \to U(Y)$. We will show that $\varepsilon_Y \bullet F(\overline{g}) = g$. Now

$$\begin{array}{rcll}
U(\varepsilon_Y \bullet F(\overline{g}) \bullet \eta_X & = & U(\varepsilon_Y) \bullet U(F(\overline{g}) \bullet \eta_X & \text{def. functor} \\
& = & U(\varepsilon_Y) \bullet \eta_{U(Y)} \bullet \overline{g} & \text{def. } \eta \text{ nat. trans.} \\
& = & \overline{g} & \text{def. } \varepsilon_Y
\end{array}$$

but this, together with $U(g) \bullet \eta_X = \overline{g}$, implies $\varepsilon_Y \bullet F(\overline{g}) = g$, by the universal property. Finally, $\overline{g}$ is unique , for if $\varepsilon_Y \bullet F(\overline{h}) = g$ then

$$\overline{g} = U(g) \bullet \eta_X = U(\varepsilon_Y \bullet F(\overline{h})) \bullet \eta_X = \overline{h}.$$

It remains to show that ε is a natural transformation, i.e., that for all $X, Y \in |\mathbf{D}|$ and $g : X \to Y$ the diagram

$$\begin{tikzcd}
F(U(X)) \arrow[r, "\varepsilon_X"] \arrow[d, "F(U(g))"'] & X \arrow[d, "g"] \\
F(U(Y)) \arrow[r, "\varepsilon_Y"'] & Y
\end{tikzcd}$$

commutes. Let $\hat{g} = \varepsilon_Y \bullet F(U(g))$. Then, $\hat{g}$ is the unique morphism such that $U(\hat{g}) \bullet \eta_{U(X)} = U(g)$, since

$$\begin{array}{rcll}
U(\hat{g}) \bullet \eta_{U(X)} & = & U(\varepsilon_Y \bullet F(U(g))) \bullet \eta_{U(X)} & \text{def. functor} \\
& = & U(\varepsilon_Y) \bullet U(F(U(G)))) \bullet \eta_{U(X)} & \text{def. } \eta \text{ nat trans} \\
& = & U(\varepsilon_Y) \bullet \eta_{U(Y)} \bullet U(g) = U(g) & \text{def. } \varepsilon_Y
\end{array}$$

and, the uniqueness follows from the universal property. So to show that the diagram commutes it suffices to show that $U(g \bullet \varepsilon_X) \bullet \eta_{U(X)} = U(g)$. But

$$\begin{array}{rcll}
U(g \bullet \varepsilon_X) \bullet \eta_{U(X)} & = & U(g) \bullet U(\varepsilon_X) \bullet \eta_{U(X)} & \text{def. functor} \\
& = & U(g) \bullet 1_{U(X)} & \text{def. } \varepsilon_X \\
& = & U(g) & \text{def. } 1_{U(X))}.
\end{array}$$

We leave the "if" part of the proof to the reader – the argument is essentially the dual of the above. $\square$

The above definition of right adjoint can also be rewritten in a weaker form analogous to that used in Proposition 12.139 – but using universal arrows from F to Y.

Example 12.142 (See Examples 6.69 and 6.70 for definitions.) Given an automaton A described by the data $\langle X, S, Y, \delta, \lambda, \sigma\rangle$, a state $s \in S$ will be said to be *reachable (from σ)* if their exists $\overline{x} \in X^*$ such that $\delta^*(\overline{x}, \sigma) = s$. We say *an automaton is reachable* if all of its states are reachable. Given an automaton A described by $\langle X, S, Y, \delta, \lambda, \sigma\rangle$, we can construct from it a reachable automaton $R(A)$ with the same behavior. To see this, let S' be the set of all reachable states in S. Observe, that if $s \in S'$, then for all $x \in X$, $\delta(x, s) \in S'$, thus there exists a total mapping $\delta' : X \times S' \to S'$, which agrees with δ on S'. Let λ' be the restriction of λ to S'. Then the desired automaton $R(A)$ is given by the data $\langle X, Y, S', \delta', \lambda', \sigma\rangle$. Clearly the reachable X-automata form a subcategory, $\mathbf{Reach}_X$, of the category, $\mathbf{Auto}_X$, of X-automata.

It is easy to show that the functor $R : \mathbf{Auto}_X \to \mathbf{Reach}_X$ taking each automaton A to its corresponding reachable automaton, $R(A)$, as defined above, is right adjoint to the inclusion functor $I : \mathbf{Reach}_X \to \mathbf{Auto}_X$. For an automaton A, $\varepsilon_A : R(A) \to A$ is just the obvious inclusion morphism. □

Example 12.143 Let $beh_X : \mathbf{Reach}_X \to \mathbf{Beh}_X$ be the behavior functor for reachable X-automata. Then we claim that beh_X has both a left adjoint $max : \mathbf{Beh}_X \to \mathbf{Reach}_X$ and a right adjoint $min : \mathbf{Beh}_X \to \mathbf{Reach}_X$. Intuitively, max constructs a maximal reachable automaton with the given behavior, and min constructs the minimal reachable automaton with the given behavior.

Proof (sketch) Given an X-behavior $b : X^* \to Y$, let $max(b)$ be the automaton specified by the data $\langle X, X^*, Y, \delta_{max}, b, \lambda\rangle$ where, for any $\overline{x} \in X^*$, and $x \in X$, $\delta_{max}(x, \overline{x}) = \overline{x} + x$ (the concatenation of $\overline{x}$ and x, where x is viewed as a string of length one). It is easy to see that $beh_X(max(b)) = b$, and we claim therefore that the unit of the adjunction is 1_b. To establish the existence of the adjunction let A be an X-automaton specified by data $\langle X, S_A, Y_A, \delta, \lambda, \sigma\rangle$, and let $h : b \to beh_X(A)$ (so h is given by a mapping $h : Y \to Y_A$ such that $h \bullet b = beh_X(A)$). Then we need to show that there is a unique automaton morphism $\overline{h}max(b) \to A$ such that $beh_X(\overline{h}) = h$. But clearly it suffices to take $\overline{h} = \langle \overline{h}_X, \overline{h}_S, \overline{h}_Y\rangle$ where $h_X = 1_X$ (this is forced by the definition of X-automaton),

$$\begin{gathered}\overline{h}_S : X^* \to S_A \\ \overline{x} \mapsto \delta_A^*(\overline{x}, \sigma)\end{gathered}$$

and $\overline{h}_Y = h$.

As for the right adjoint, given an X-behavior $b : X^* \to Y$ let $min(b)$ be the X-automaton given by the data $\langle X, X^*/Q, Y, \delta_{min}, \lambda_{mn.}, \lambda\rangle$ where Q is the equivalence relation on^{X*} given by $\langle \overline{x}, \overline{y}\rangle \in Q$ iff for all $\overline{z} \in X^*$, $b(\overline{x} + \overline{z}) = b(\overline{y} + \overline{z})$. Given $\overline{x} \in X^*$ let $\lceil \overline{x} \rceil$ denote its Q-equivalence class. Then define $\delta_{min}(x, \lceil \overline{x} \rceil) = \lceil \overline{x} + x \rceil$, and $\lambda_{min}(\lceil \overline{x} \rceil) = b(\overline{x})$. Again we get $beh_x(min(b)) = b$. We claim now that $beh_X \dashv min$ with counit 1_b. To see that this is so, let B be a reachable X-automaton given by the data $\langle X, S_B, Y_B, \delta_B, \lambda_B, \sigma\rangle$, and let $h : beh_X(B) \to b$. Then we must show that there exists a unique $\overline{h} : B \to min(b)$ such that $beh_X(\overline{h}) = h$. We claim that it suffices to take $\overline{h} = \langle 1_X, h_S, h\rangle$ where

$$\begin{gathered}h_S : S_B \to X^*/Q \\ s \mapsto \lceil \overline{x} \rceil\end{gathered}$$

where $\overline{x}$ is any element of X^* such that $\delta_B^*(\overline{x}, \sigma) = s$. □

$\square$

Here is another useful definition of adjunction.

Proposition 12.144 *Let $U : \mathbf{C} \to \mathbf{D}$, and let $F : \mathbf{D} \to \mathbf{C}$, then F is a left adjoint to U if, there is a natural isomorphism*

$$\psi : \mathbf{C}(F(\_), \_) \Rightarrow \mathbf{D}(\_, U(\_))$$

between the indicated hom functors (see Definition 6.88). That is, for all $X_1, X_2 \in |\mathbf{D}|$, and $Y_1, Y_2 \in |\mathbf{C}|$,

$$\psi_{X_1,\ Y_1} : \mathbf{C}(F(X_1),\ Y_1) \to \mathbf{D}(X_1, U(Y_1))$$

is an isomorphism natural in X_1 and Y_1, meaning that, for all $f : X_2 \to X_1$ and $g : Y_1 \to Y_2$, the following diagrams commute:

$$\begin{tikzcd}
\mathbf{C}(F(X_1,\ Y_1) \arrow[r, "\psi_{X_1,\ Y_1}"] \arrow[d, "\mathbf{C}(F(X_1),\ g)"'] & \mathbf{D}(X_1,\ U(Y_1)) \arrow[d, "\mathbf{D}(X_1,\ U(g))"] \\
\mathbf{C}(F(X_1),\ Y_2) \arrow[r, "\psi_{X_1,\ Y_2}"'] & \mathbf{D}(X_1,\ U(Y_2))
\end{tikzcd}$$

i.e., for any $\alpha \in \mathbf{C}(F(X_1),\ Y_1)$, $U(g) \bullet \psi_{X_1,Y_1}(\alpha) = \psi_{X_1,Y_2}(g \bullet \alpha)$.

$$\begin{tikzcd}
\mathbf{C}(F(X_1,\ Y_1) \arrow[r, "\psi_{X_1,\ Y_1}"] \arrow[d, "\mathbf{C}(F(f),\ Y_1)"'] & \mathbf{D}(X_1,\ U(Y_1)) \arrow[d, "\mathbf{D}(f,\ U(Y_1))"] \\
\mathbf{C}(F(X_2),\ Y_1) \arrow[r, "\psi_{X_2,\ Y_1}"'] & \mathbf{D}(X_2,\ U(Y_1))
\end{tikzcd}$$

i.e., for any $\alpha \in \mathbf{C}(F(X_1),\ Y_1)$, $\psi_{X_1,Y_1}(\alpha) \bullet f = \psi_{X_2,Y_1}(\alpha \bullet F(f))$.

Proof (very bare sketch) Referring back to Proposition 12.141, the intuition is that ψ should be the mapping that takes β to $\overline{\beta}$ This is captured precisely by taking

$$\psi_{X,Y} : \mathbf{C}(F(X),\ Y) \to \mathbf{D}(X,\ U(Y))$$
$$\beta \mapsto U(\beta) \bullet \eta_X$$

Correspondingly,

$$(\psi_{X,Y})^{-1} : \mathbf{D}(X,\ U(Y)) \to \mathbf{C}(F(X),\ Y)$$
$$\alpha \mapsto \varepsilon_Y \bullet F(\alpha)$$

Conversely, given the above definition with ψ, define η by $\eta_X = \psi_{X,F(X)}(1_{F(X)})$, and define ε by $\varepsilon_Y = (\psi_{U(Y),Y})^{-1}(1_{U(Y)})$. $\square$

Proposition 12.145 *Any two left adjoints to $U : \mathbf{C} \to \mathbf{D}$ are naturally isomorphic.* $\square$

Proposition 12.146 *If $F_1 \dashv U_1 : \mathbf{C}_1 \to \mathbf{C}_2$, and $F_2 \dashv U_2 : \mathbf{C}_2 \to \mathbf{C}_3$, then $F_1 \bullet F_2 \dashv U_2 \bullet U_1 : \mathbf{C}_1 \to \mathbf{C}_3$*

Proposition 12.147 *Left adjoints preserve colimits, and right adjoints preserve limits. Thus, for example, if $F \dashv U : \mathbf{C} \to \mathbf{D}$, and $G : \mathbf{S} \to \mathbf{D}$ is a diagram in $\mathbf{D}$ with colimiting cone $\langle \eta_X : G(X) \to L \mid X \in |\mathbf{S}| \rangle$ in $\mathbf{D}$, then $F(\eta) = \langle (\eta_X) : F(G(X)) \to F(L) \mid X \in |\mathbf{S}| \rangle$ is a colimiting cone in $\mathbf{C}$.*

The doctrine (see Introduction) that any natural construction is represented by an adjoint functor, is well illustrated by the example of limits and colimits.

Proposition 12.148 *Given a category $\mathbf{C}$ let*

$U_{\mathbf{1}} : \mathbf{C} \to \mathbf{1}$ be the unique functor from $\mathbf{C}$ to $\mathbf{1}$.

$\Delta : \mathbf{C} \to \mathbf{C} \times \mathbf{C}$ be the diagonal functor

$$\Delta : X \mapsto \langle X, X \rangle$$

and, for $\alpha : X \to Y in \mathbf{C}$,

$$\Delta : \alpha \mapsto \langle \alpha, \alpha \rangle : \langle X, X \rangle \to \langle Y, Y \rangle.$$

Then $\mathbf{C}$ has

1. *initial objects iff $U_{\mathbf{1}}$ has a left adjoint.*
2. *terminal objects iff $U_{\mathbf{1}}$ has a right adjoint.*
3. *binary coproducts iff Δ has a left adjoint.*
4. *binary products iff Δ has a right adjoint.*

See [11] for further examples including the general case.

Proof (sketch) To give the idea we will sketch parts of the proofs for (1) and (4) First we show that any left adjoint F of $U_{\mathbf{1}}$ yields an initial object. Assume $U_{\mathbf{1}} : \mathbf{C} \to \mathbf{1}$ has a left adjoint $F : \mathbf{1} \to \mathbf{C}$. Then, letting X denote the sole object of $\mathbf{1}$, we must have $U_{\mathbf{1}}(F(X)) = X$, and $\eta_X = 1_X$ since these are the only objects and morphisms in $\mathbf{1}$. Similarly, given any $Y \in |\mathbf{C}|$, we must have $U_{\mathbf{1}}(Y) = X$, and 1_X is the only morphism from X to $U_{\mathbf{1}}(Y)$. But $F \dashv U_{\mathbf{1}}$, says then that there is a unique morphism $f : F(X) \to Y$ such that $U(f) \bullet \eta_X = 1_X$, i.e. such that $U(f) = 1_X$, which thus implies that there is exactly one morphism from $F(X)$ to Y, i.e., that $F(X)$ is initial.

We will now show that, if $\mathbf{C}$ has a product functor

$$P : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$$
$$\langle A, B \rangle \mapsto A \times B$$

then P is right-adjoint to Δ with counit

$$\varepsilon_{\langle A,B \rangle} : \Delta(P(\langle A,\ B \rangle)) \to \langle A,\ B \rangle$$
$$\varepsilon_{\langle A,B \rangle} = \langle \pi_A, \pi_B \rangle : \langle A \times B,\ A \times B \rangle \to \langle A, B \rangle.$$

To see that the product is a right adjoint for Δ, consider that for any pair $\langle A, B\rangle \in |\mathbf{C}\times\mathbf{C}|$, and any $D \in |\mathbf{C}|$, we have $\Delta(D) = \langle D, D\rangle$. A morphism $h : \Delta(D) \to \langle A, B\rangle$ is then a pair of morphisms, $\langle f, g\rangle$ with $f : D \to A$ and $g : D \to B$. To show that the product gives a right adjoint for Δ we must show that, for each $h : \Delta(D) \to \langle A, B\rangle$, there is a unique $\overline{h} : D \to P(\langle A, B\rangle) = A \times B$ such that the following diagram commutes.

$$
\begin{tikzcd}
\langle A,B\rangle & \Delta(P(\langle A,B\rangle)) \arrow[l] & = & \langle A\times B,\ A\times B\rangle & A\times B \\
& \Delta(D) \arrow[u, "\Delta(\overline{h})"'] \arrow[ul, "h\, =\, \langle f,g\rangle"] \arrow[rr, phantom, "="] & & \langle D,\ D\rangle \arrow[u, "\langle\overline{h},\ \overline{h}\rangle"] & D \arrow[u, "\overline{h}"']
\end{tikzcd}
$$

But clearly the only choice is to take $\overline{h} : D \to A \times B$ to be the mediating morphism for the product (see Definition 9.103). $\square$

References

[1] J. A. Bergstra and J. V. Tucker. Algebraic specifications of computable and semicomputable data structures. *TCS*, 50:137–181, 1987.

[2] S. Eilenberg and J. B. Wright. Automata in general algebras. *Information and Control*, 11:52–70, 1967.

[3] J. A. Goguen. Minimal realization of machines in closed categories. *Bulletin of the AMS*, 78:777–183, 1972.

[4] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. A junction between computer science and category theory: I, basic definitions and examples, part 1. IBM Research Report RC 4526, IBM, September 1973.

[5] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology, IV, Data Structuring*, pages 80–149. Prentice-Hall, 1978.

[6] J.A. Goguen and R. M. Burstall. Introducing institutions. In *Proc. Logics of Programming Workshop*, pages 221–256. Carnegie-Mellon, Springer-Verlag, LNCS 154, 19??

[7] J.A. Goguen, J.W. Thatcher, E. G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *JACM*, 24:68–95, 1977.

[8] H. Herrlich and G. E. Strecker. *Category Theory.* Heldermann-Verlag, Berlin, second edition, 1979.

[9] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, Cambridge, 1986.

[10] F. W. Lawvere. Functorial semantics of algebraic theories. *Proc. National Academy of Science*, 50:869–872, 1963.

[11] Saunders MacLane. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

[12] E. G. Manes. Weakest preconditions: Categorical insights. In D. Pitt, S. Abramsky, A. Poigne, and D. Rydeheard, editors, *Category Theory and Computer Programming, Tutorial and Workshop, Guildford, U.K., September 1985*, pages 182–197. Springer-Verlag, 1985.

[13] A. Pasztor. Epis of some categories of z-continuous partial algebras. *Acta Cybernetica*, 6:111–123, 1983.

[14] J.W. Thatcher, E. G. Wagner, and J.B. Wright. More advice on structuring compilers and proving them correct. *TCS*, 24:223–249, 1981.

[15] J.W. Thatcher, J.B. Wright, and E. G. Wagner. Data type specification: Parameterization and the power of specification techniques. *TOPLAS*, 4:711–732, 1982.

[16] E. G. Wagner. An algebraic theory of recursive definitions and recursive languages. In *Proceedings Third IEEE Symp. on Switching and Automata Theory,E. Lansing, Mich.*, pages 12–23, 1971.

[17] E. G. Wagner. Algebras, polynomials, and programs. *Theoretical Computer Science*, 70:3–34, 1990.

[18] E. G. Wagner and H. Ehrig. Canonical constraints for parameterized data types. *Theoretical Computer Science*, 50:323–349, 1987.

[19] Eric G. Wagner. A categorical treatment of pre- and post-conditions. *Theoretical Computer Science*, 53:3–24, 1987.

[20] Eric G. Wagner. Semantics of block structured languages with pointers. In *Proceedings of the 3rd Annual Workshop on the Mathematical Foundations of Programming Language Semantics.* LNCS 298, Springer Verlag, 1987.

[21] M. Wand. Final algebra semantics and data type extension. *J. Comp. Sys. Sciences*, 19:27–44, 1979.

[22] J.B. Wright, J.W. Thatcher, and E. G. Wagner. Rational algebraic theories and fixed-point solutions. In *Proceedings 17th IEEE Symposium on Foundations of Computing*, pages 147–158, 1976.