PB-163

N. Husberg: Categorial Heterogeneous Algebraic Models

TRYK: DAIMI/RECAU

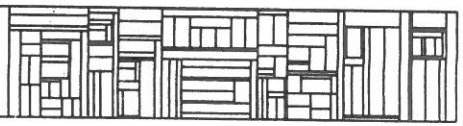
ISSN 0105-8517

Categorial Heterogeneous Algebraic Models of Programming Languages

Nisse Husberg

DAIMI PB-163
August 1983

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



CATEGORIAL HETEROGENEOUS ALGEBRAIC MODELS OF PROGRAMMING LANGUAGES

Nisse Husberg

1983

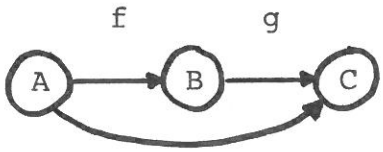
Abstract

Heterogeneous algebraic theories and algebras are treated in detail with examples showing how to model programming languages. All categorial concepts needed - and only those - are defined and explained assuming no earlier background in category theory. The presentation covers language grammars and syntax fairly well, but semantics is treated only in general terms.

CONTENTS

0 NOTATIONS

A mapping $f:X \to Y : x \mapsto xf$ has source (set) X and target (set) Y. Because categories and graphs are important in this work the argument is to the left of the function symbol, that is xf or (x)f instead of f(x). Composition of mappings (and morphisms in general) is written in the natural way



fg

and x(fg) = (xf)g like (GTWW 1973), (Manes 1976) and (Cohn 1965), but unlike (Schubert 1972) and (Herrlich-Strecker 1973).

Composition of f and g is usually written fg but if it must be stressed that fg is a composition it is sometimes written $f \circ g$.

If A denotes a family of sets $A=(A_s \mid s \in S)$ then the cartesian product is denoted

$$A^w = A_{s_1} \times \cdots \times A_{s_n}, \text{ where } w = s_1 \cdots s_n.$$

The elements of A^w are w-tuples, i.e.

$$A^w = \{(a_{s_1}, \cdots, a_{s_n}) \mid a_{s_i} \in A_{s_i}\}.$$

If S has one element only, the product is denoted A^n. If w=s (a string with length 1) the product is denoted A_s (a set of sort s). If $w=\lambda$ then $A^w = A^\lambda = \{()\}$, the set with one element which is the empty tuple.

A set is denoted $A = \{a_1, \cdots, a_n\}$ or $A = \{ a \mid p(a) \}$, where p(a) is a predicate, that is, if a has some property defined by the predicate, then a is in the set A.

The empty set $\{\}$ is denoted by $\emptyset$.

An alphabet A is a set of some symbols (usually a finite set) A string (in an alphabet A) is a sequence of symbols $a_1 \cdots a_n$ in A. The empty string is denoted by λ. The set of all strings (including the empty string) is denoted by A^* and (excluding the empty string) by A^+.

1. INTRODUCTION

Category theory is used quite much in computer science, but only by mathematicians and those - rather few - "engineers" who have had motivation enough to climb over the threshold into the categorial paradise. Those who have made it tend to go still deeper into the fascinating landscape thereby leaving the majority of the computer people out in the cold.

This is a modest attempt to give a hand to those still outside which are interested but do not know where to begin. It is also intended to clarify the basic framework which is often presented a bit too briefly because the stress is put on more complex and more interesting features. For the newcomer it is, however, important to have a good understanding in the basic ideas.

The ADJ group (Goguen, Thatcher, Wagner, Wright) has made important contributions to the introduction of category theory into computer science. In their report "A junction between computer science and category theory" (part I 1973, part II 1976) they presented a lot of interesting categorial means and showed how they could be used in computer science.

The most interesting phase was, however, the introduction of a heterogeneous algebraic model (HAM) of the syntax and the

semantics of a programming language. This was made by Letichevski already in 1968 and by Rus in 1972, who both used heterogeneous algebra to describe (rather abstract) translators. The notations used by the ADJ group and the idea to put the HAMs into a categorial framework (CHAM) made the approach concise and efficient.

A small but important step which made the HAM very easy to read was the introduction of the "mixfix" notation (Rus 1972 and Goguen 1978) thereby "distributing" the operator symbol among the arguments as in most programming languages (if p then A else B, where if _ then _ else _ is an operator symbol with "placeholders" _).

The initial, free and other algebras used here are close to those used in the ADJ papers (GTWW 1977) but they are constructed in a different way. Here the complete categorial construct is given explicitly in the general heterogeneous case. To hide the categories as in (GTWW 1977) is not a "soft" introduction but rather creates confusion.

All the basic concepts needed are presented here because no former knowledge of category theory is assumed. Some rather advanced concepts like algebraic theories are treated in great detail while many basic concepts of category theory found in any textbook are not included at all.

The choice of material is determined by the application - modelling programming languages - but the disposition is close to an introduction into category theory. The main ideas can be found in chapters 2 and 3 in (Husberg 1980a), but in a rather primitive form.

No parallels to other similar approaches are drawn but a lot of practical examples are given to help the intuitive understanding. A good bibliography can be found in (Wagner 1981).

Warning ! The algebraic theories used here are roughly dual (the arrows are "turned around") to those of ADJ. This has the advantage that arrows point in the same direction in the algebraic theory and in the algebra generated by it. The arrow turning of ADJ is confusing for beginners. Notice that (Goguen 1975) has a different definition than in most ADJ papers, the same as the one used here and in (Gallier 1978) and (Gallier 1981).

2. BASIC CATEGORIES OF CHAM

Here some basic categories used in the categorial heterogeneous algebraic model (CHAM) are presented. Unfortunately the literature of category theory uses many different definitions of a category. Although most differences are notational, it can, however, be rather difficult to read them "side by side". Thus all concepts needed here are defined here also.

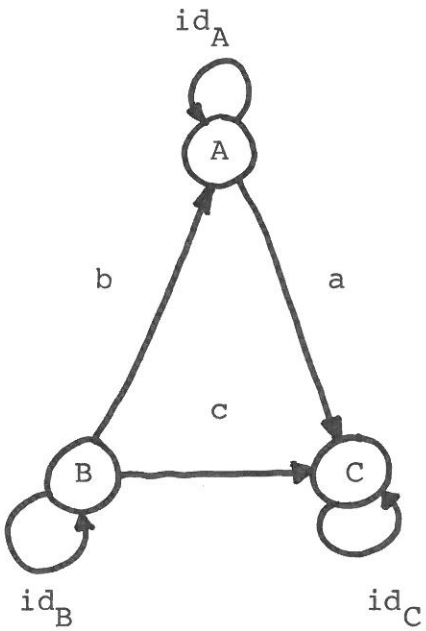
The basic concepts used are mainly from (Schubert 1972), (GTWW 1975) and (Cohn 1965), but also (Pareigis 1970), (Herrlich-Strecker 1973) and (MacLane 1971) are used as references. Those who want to go deeper into category theory could start with Herrlich-Strecker which has many examples from "normal" mathematics or Cohn if they like algebra. Schubert is not explaining much but is logical and exact. It also covers many topics. A very nice introduction (but not complete) can be found in (Goldblatt 1979) - do not get scared by the title.

The heterogeneous case is not treated at all in usual textbooks on category theory although it is possible to generalize the homogeneous case fairly easily. Explicit definitions of many-sorted (heterogeneous) algebraic

theories can be found in (WTW 1978) and (WWT 1979), but deprived of almost all category-theoretic background.

A category consists of objects and morphisms. An object can be seen as a generalization of the set concept and a morphism as a generalization of the mapping (function) concept. In general objects are no sets and morphisms no mappings.

A graph representation is sometimes used for a category where the nodes denote objects and the arrows denote morphisms:



In general a graph is no category because a category must have identities for each object in the category. Compositions of morphisms must also be defined and it must be unique and associative.

If in the example above we define composition so that $b \circ a$ (or simply ba - notice the order !) is the same as c then the graph above is a category with three elements. To sum it up we have the following definition:

A <u>category</u> <u>K</u> consists of a class of <u>objects</u> $Ob_{\underline{K}}$ and for each pair $A,B \in Ob_{\underline{K}}$ a class of <u>morphisms</u> $Mor_{\underline{K}}(A,B)$ which satisfy the following conditions:

1) each morphism belongs to <u>only</u> <u>one</u> morphism class, i.e. $Mor_{\underline{K}}(A,B) \cap Mor_{\underline{K}}(C,D) = \emptyset$ if $(A,B) \neq (C,D)$,

2) for each object $A \in Ob_{\underline{K}}$ there is a special morphism $id_A \in Mor_{\underline{K}}(A,A)$ called <u>identity</u> such that for each $Mor_{\underline{K}}(A,B)$ we have

$$(id_A)f = f = f(id_B),$$

3) if $f \in Mor_{\underline{K}}(A,B)$ and $g \in Mor_{\underline{K}}(B,C)$ then there is a <u>unique</u> morphism $fg = Mor_{\underline{K}}(A,C)$ called the <u>composition</u> of f and g,

4) composition is <u>associative</u>, i.e. $(fg)h = f(gh)$ for $f \in Mor_{\underline{K}}(A,B)$, $g \in Mor_{\underline{K}}(B,C)$ and $h \in Mor_{\underline{K}}(C,D)$.

A morphism $f \in Mor_{\underline{K}}(A,B)$ is usually denoted by $f: A \rightarrow B$, where A is the source and B the target, although f possibly is no mapping (or relation).

In general the objects do not form a set but a class (a more general concept than a set). If $Ob_{\underline{K}}$ is a set then the category is called a small category. If all morphisms from A to B, where (A,B) is any pair of objects in $Ob_{\underline{K}}$, form a set then $\underline{K}$ is locally small (or well-powered).

If a category is small it is also locally small, but not necessarily vice versa. The important category Set with sets as objects and all mappings between the sets as morphisms is locally small but not small (because the collection of all sets is not a set).

A set is a discrete category because a discrete category has no other morphisms than identities.

A product category $\underline{K}_1 \times \underline{K}_2 \times \cdots \times \underline{K}_n$ of categories $\underline{K}_1, \underline{K}_2, \cdots, \underline{K}_n$ is a category with ordered sequences of $\underline{K}_i$-objects $(K_1, K_2, \cdots, K_n)$ as objects and ordered sequences of $\underline{K}_i$-morphisms
$(f_1, f_2, \cdots, f_n): (K_1, K_2, \cdots, K_n) \rightarrow (K_1', K_2', \cdots, K_n')$
as morphisms, where $K_i, K_i' \in Ob_{\underline{K}_i}$ and $f_i: Mor_{\underline{K}_i}(K_i, K_i')$.

Composition is defined by:

$$(f_1,f_2,\cdots,f_n)\circ(g_1,g_2,\cdots,g_n) = (f_1\circ g_1, f_2\circ g_2,\cdots,f_n\circ g_n)$$

and identity is obviously

$$id_{(K_1,K_2,\cdots,K_n)} = (id_{K_1},id_{K_2},\cdots,id_{K_n}).$$

The dual or opposite category $\underline{K}^{op}$ of a category $\underline{K}$ has exactly the same objects as $\underline{K}$ and the "same" morphisms but with "reversed arrows", i.e.

$$Mor_{\underline{K}^{op}}(A,B) = Mor_{\underline{K}}(B,A).$$

There is a morphism $f^{op} : B \to A$ in $\underline{K}^{op}$ iff there is a corresponding morphism $f: A \to B$ in $\underline{K}$ and composition in $\underline{K}^{op}$ is defined by $f^{op}g^{op} = (fg)^{op}$.

A subcategory $\underline{L}$ of a category $\underline{K}$ is a category with

1) $Ob_{\underline{L}} \subseteq Ob_{\underline{K}}$,
2) $Mor_{\underline{L}}(A,B) \subseteq Mor_{\underline{K}}(A,B)$ for each pair $A,B \in Ob_{\underline{L}}$,
3) for each pair (f,g), where $f: A \to B$, $g: B \to C$, and for any $A,B,C \in Ob_{\underline{L}}$, the composition in $\underline{K}$ $fg \in Mor_{\underline{L}}(A,C)$,
4) for each $A \in Ob_{\underline{L}}$, the identity id_A in $\underline{K}$ is also the identity of A in $\underline{L}$.

A subcategory is <u>full</u> if for any two objects $A,B \in Ob_{\underline{L}}$ all $\underline{K}$-morphisms from A to B are also $\underline{L}$-morphisms, i.e.

$$Mor_{\underline{K}}(A,B) = Mor_{\underline{L}}(A,B).$$

2.1 Some special objects and morphisms

The notion of product is very important, e.g. in the string category to be defined in next section and used throughout the paper. For sets the (cartesian) product is defined for two sets A and B as

$$A \times B = \{(a,b): a \in A \text{ and } b \in B\}$$

with mappings

$$p_A: A \times B \to A: (a,b) \mapsto a$$

and

$$p_B: A \times B \to B: (a,b) \mapsto b.$$

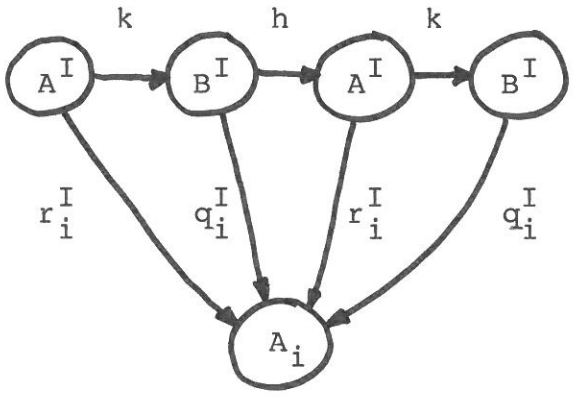
In a category the product is defined more generally. A product can be defined for an arbitrary family $(A_i | i \in I)$ of objects in a category $\underline{K}$, where I is an index set (Schubert 1972, p. 49).

A product is an object A^I (or $\Pi_{i \in I} A_i$) with morphisms $p_i^I: A^I \to A_i$ such that for each object $Y \in Ob_{\underline{K}}$ and for any family $(f_i: Y \to A_i | i \in I)$ there is exactly one morphism $h: Y \to A^I$ such that $f_i = hp_i^I$ for all $i \in I$ (the universal property). Usually h is denoted (f_i) or $(f_{i_1},\cdots,f_{i_n})$ if I is a finite set.

The morphisms p_i^I are called projections. A finite product has a finite index set I. The identity for a product A^I is the I-tuple of projections $(p_i \mid i \in I)$.

If $I = \emptyset$ then for each object $Y \in Ob_{\underline{K}}$ there must be exactly one morphism h from Y into the empty product E. Thus E is a terminal object (Pareigis 1970, p. 31).

Pareigis has also proved an important lemma: that products of the same family of objects are isomorphic, i.e.:
Let $(A^I, (r_i^I))$ and $(B^I, (q_i^I))$ be products of some collection of objects $(A_i \mid i \in I)$ in $\underline{K}$. Then there is a uniquely determined isomorphism $k: A^I \to B^I$ such that $r_i^I = kq_i^I$.



There must be unique morphisms k and h because A^I and B^I are products. For the same reason kh and hk must be unique and $kh = id_A I$ and $hk = id_B I$.

Initiality is very important in computer science because it ensures the existence and the uniqueness of a morphism from the initial object to any other object in the category.

An initial object X for a category K is an object such that $Mor_K(X,B)$ contains exactly one morphism for all K-objects B. All initial objects for K are isomorphic (Schubert 1972, p. 35).

In the category Set the empty set $\emptyset$ is an initial object because there is exactly one mapping from $\emptyset$ to any other set, the empty mapping:

$$\lambda : \emptyset \rightarrow B: \quad \rightarrow B$$

where $B \in Ob_{Set}$ is a set.

There is a dual concept (dual means roughly "turning arrows around") and it is terminal.

A terminal object Y for a category K is an object such that $Mor_K(B,Y)$ contains exactly one morphism for all K-objects B. All terminal objects are isomorphic.

In the category Set any set consisting of one element is a terminal object. For any set B there is only the mapping $s_B: B \rightarrow \{y\}: b \rightarrow y$, mapping every element $b \in B$ into the single element y of the terminal object (from $\emptyset$ there is the empty mapping). The isomorphisms are $i: Y \rightarrow Y': y \rightarrow y'$, where Y and Y' are one element sets.

Considering that a morphism is a generalization of a function, it is reasonable to expect that there are

morphisms having properties corresponding to "injective" and "surjective" functions. Because an object may be no set, these properties are defined "externally", i.e. without any reference to the structure of the objects (Goldblatt 1979, p.37).

A monomorphism (monic morphism) $f: A \to B$ in a category $\underline{K}$ has the property that for any pair of morphisms $g,h: C \to A$ in $\underline{K}$

$$gf = hf \text{ implies } g = h.$$

In Set a monomorphism is an injective mapping.

An epimorphism (epic morphism) $f: A \to B$ in a category $\underline{K}$ has the property for any pair of morphisms $m,n: B \to C$ in $\underline{K}$

$$fm = fn \text{ implies } m=n.$$

In Set an epimorphism is a surjective mapping.

A morphism $f: A \to B$ in a category $\underline{K}$ is an isomorphism if there is a morphism $g: B \to A$ in $\underline{K}$ such that

$$fg = id_A \text{ and } gf = id_B.$$

An isomorphism is both a monomorphism and an epimorphism but every monic and epic morphism is not an isomorphism (Pareigis, p.17).

2.2 The string category

The category StS with strings as objects is the basis of the algebraic theories used in this CHAM (categorial heterogeneous algebraic model). It models sort strings, where a "sort" can be interpreted as a nonterminal in a grammar or as a word in angle brackets <> in the BNF (Backus-Naur Form) of syntax. It is a good example that the objects in a category can have an "inner" structure which is "hidden" in the category.

Here the category $\underline{N}$ of natural numbers is needed (Herrlich-Strecker 1973, p. 20). The idea is that $\underline{n}$ denotes the set of all smaller numbers, i.e. $\underline{n} = \{\underline{0},\underline{1},\cdots,\underline{n-1}\}$, but because we want to index elements in the strings directly from this set it is better to use

$$[n] = \{1,2,\cdots,n\}$$

like (WTW 1978). Consequently $[0] = \{\} = \emptyset = \underline{0}$ and we take

$$[1] = \{1\} = \{\emptyset\} = \{\underline{0}\} = \underline{1}$$
$$[2] = \{1,2\} = \{\emptyset,\{\emptyset\}\}$$
$$\vdots$$

as objects in the category $\underline{N}$. This category could be defined as a discrete category if the order of the natural numbers is forgotten, but now the mappings between the sets $[0],[1],\cdots$ are taken as morphisms of $\underline{N}$.

Because there is no element in $[0] = \emptyset$ an unique mapping can always be defined from this into any other object $[n]$, i.e. the empty mapping $i^n_\lambda: [0] \to [n]$. Thus $[0]$ is the initial object of $\underline{N}$.

In $Mor_{\underline{N}}([1],[n])$ there are exactly n morphisms, i.e. the injections $i^n_j: [1] \to [j]: 1 \mapsto j$, where $j = 1, \cdots, n$. In the opposite morphism set $Mor_{\underline{N}}([n],[1])$ there is only one morphism, the surjection $s^n_1: \{1,2,\cdots,n\} \to \{1\}: n \mapsto 1$. Thus $[1]$ is the terminal object of $\underline{N}$.

Strings from a set S are defined as mappings

$$w: [n] \to S: i \mapsto s$$

but are usually written $w = s_1 \cdots s_n$, that is, the element iw in S is denoted by s_i. The set of all strings from S (including the empty string) is often denoted S^*.

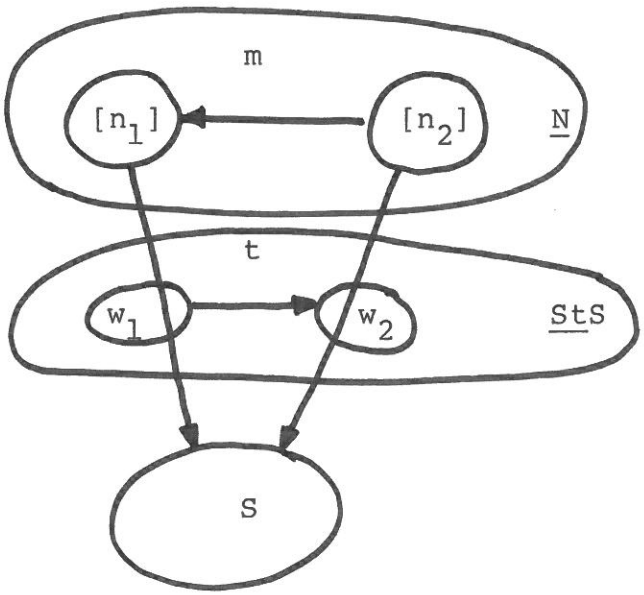
The length of a string w is $wlg = n$, where lg is a mapping from the strings into the maximum element n of their source set $[n]$, i.e. $lg: S^* \to N: w \mapsto n$ if $w: [n] \to S$ and N is the set of natural numbers.

The sort set of a string w is the subset S_w of S which can be defined as

$S_w = \{ s \mid iw = s \text{ for some } i \text{ with } 1<=i<=n \text{ and } n = wlg \}$

Taking strings $w \in S^*$ as objects in a category StS the object class Ob_{StS} is the set S^*. The morphisms of this category are defined in the following way:

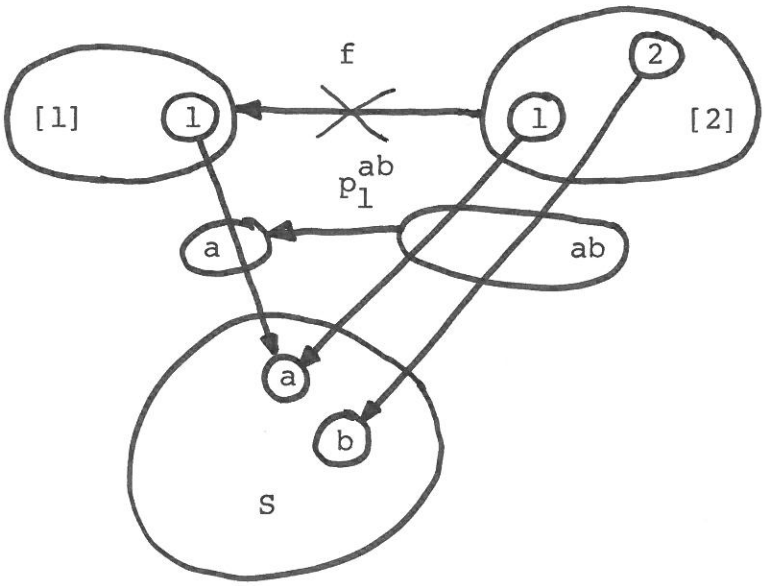
There is a morphism t: $w_1 \rightarrow w_2$ in $Mor_{StS}(w_1, w_2)$ iff there is a mapping m: $[n_2] \rightarrow [n_1]$ such that the following diagram commutes:



It may seem strange that t and m have opposite "directions" but it is perfectly possible to define the morphisms in such a way. The reason for this is that we need products (and thus projections) in StS. If the arrows are not turned around here, they have to be turned around when going from the algebraic theory to its algebras which is considered

more harmful because a lot of work will be done on that level while this is only for the basic construction of the string category.

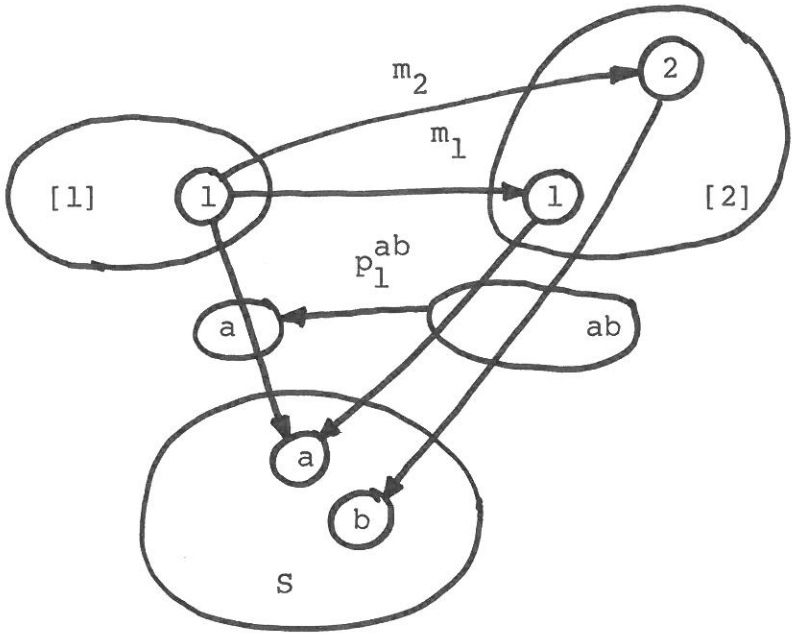
If t is turned around, the strings will not be products (but coproducts) because in the diagram below there can be no such mapping f that the diagram should commute. Therefore the projection p_1^{ab} is no morphism in such a category.



Remember that in order to commute the diagram must have $f \circ a = (ab)$, i.e. (if) $a = i(ab)$, that is, f applied to i (in [2]) composed with a gives the same element of S as ab applied to the same i.

The only mapping from [2] to [1] is the surjection defined earlier. It has $1f = 1$ and thus $(1f)a = a = 1(ab)$, but as $2f = 1$ also we have $(2f)a = a \neq b = 2(ab)$.

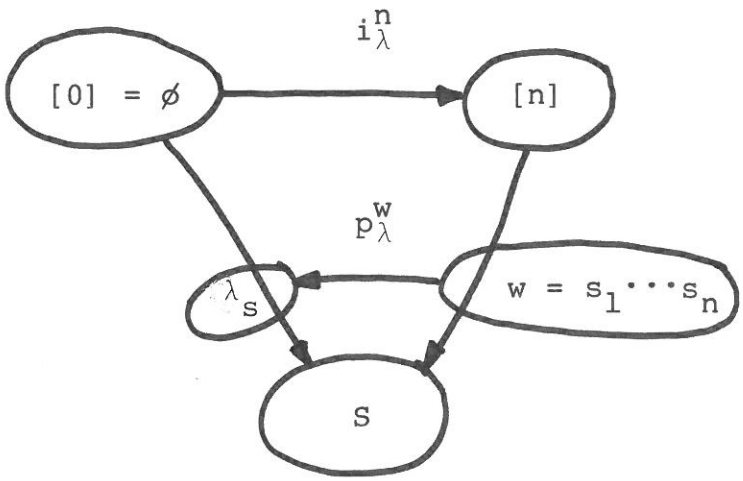
According to the definition of t we have



where there are <u>two</u> mappings in $Mor_{\underline{N}}([1],[2])$ as defined earlier. The diagram commutes for $m_1(ab) = a$ which means that $p_1^{ab} \in Mor_{\underline{StS}}(ab,a)$.

<u>Another</u> diagram could be given for $m_2(ab) = b$ which would give p_2^{ab} as the second projection. Thus ab has both projections needed. In order to prove that ab really is a product in <u>StS</u>, the universal property should also be asserted, but it is not within the scope of this report.

A special product is very interesting - the empty string (sometimes it is indexed λ_S to show that it is an element of the set S):



For every n <u>there is a unique mapping</u> i^n_λ: [0] → [n] - the <u>empty</u> mapping. The empty string λ_s: [0] → S is the empty mapping into S. Thus the diagram commutes for all w of length n and from <u>every</u> <u>string</u> w in the category <u>StS</u> there is a <u>unique</u> projection (the lambda projection) from w to λ_s (the empty sort string) which is a <u>terminal</u> <u>object</u> in <u>StS</u> (see 2.1).

A string w = abc is a product of the objects a,b and c. Thus using the notations in the definition of the product (section 2.1) we could have

$$I = \{1,2,3\} \qquad A_1 = a,\ A_2 = b,\ A_3 = c$$

and

$A^I = w = abc$ with

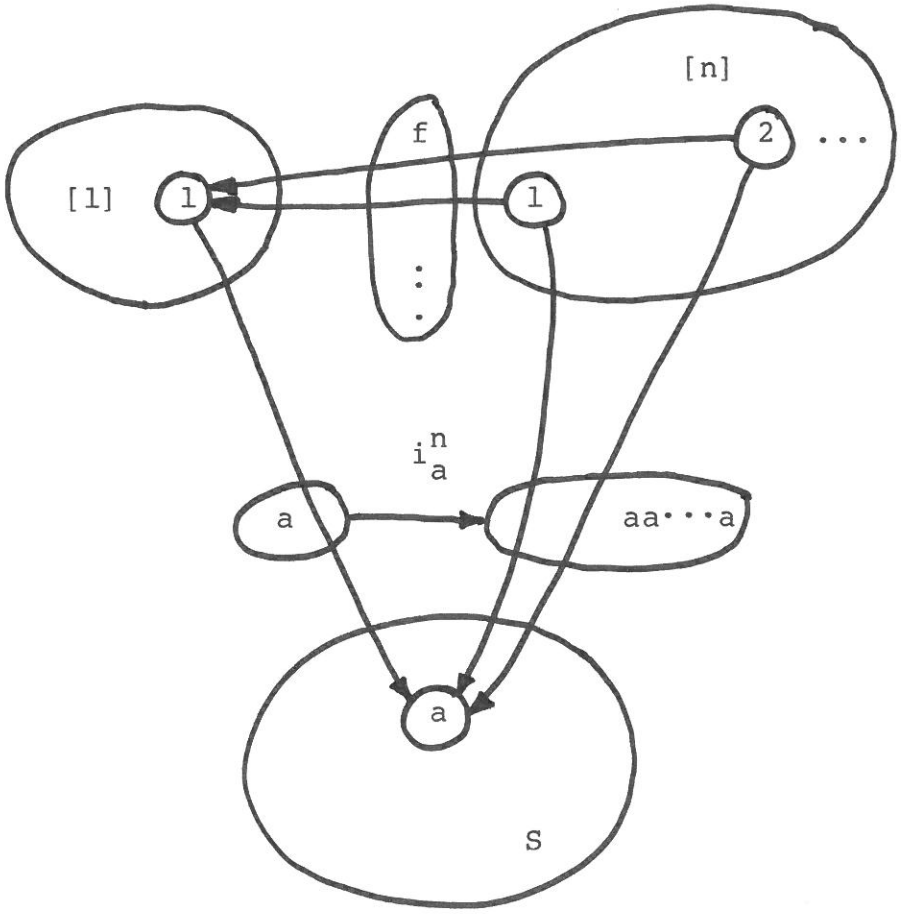
$p^I_1: A^I \to A_1$, i.e. $p^w_1: w \to a$

$p^I_2: A^I \to A_2$, i.e. $p^w_2: w \to b$

$p^I_3: A^I \to A_3$, i.e. $p^w_3: w \to c$

<u>StS</u> has injections i^n_s: s → ss···s which are unique because [1] is a terminal object in <u>N</u>, i.e. there is a unique

mapping from every object [n] to [1], that is, the surjection s_1^n: [n] $\rightarrow$1: n $\mapsto$1. But the single sort strings s are <u>not</u> initial objects in <u>St</u>S because there are no morphisms into an arbitrary string w from s.



By the universal property for products (see 2.1) there must be a <u>unique</u> morphism h: w' $\rightarrow$w for any object w' $\in$ $Ob_{\underline{St}S}$ with a family of morphisms $(f_i: w' \rightarrow A_i \mid i \in I)$.

Now there are many products for a,b and c. For example, if B^I = w' = bac with

$$A_1 = b,\ A_2 = a,\ A_3 = c$$

we have the product

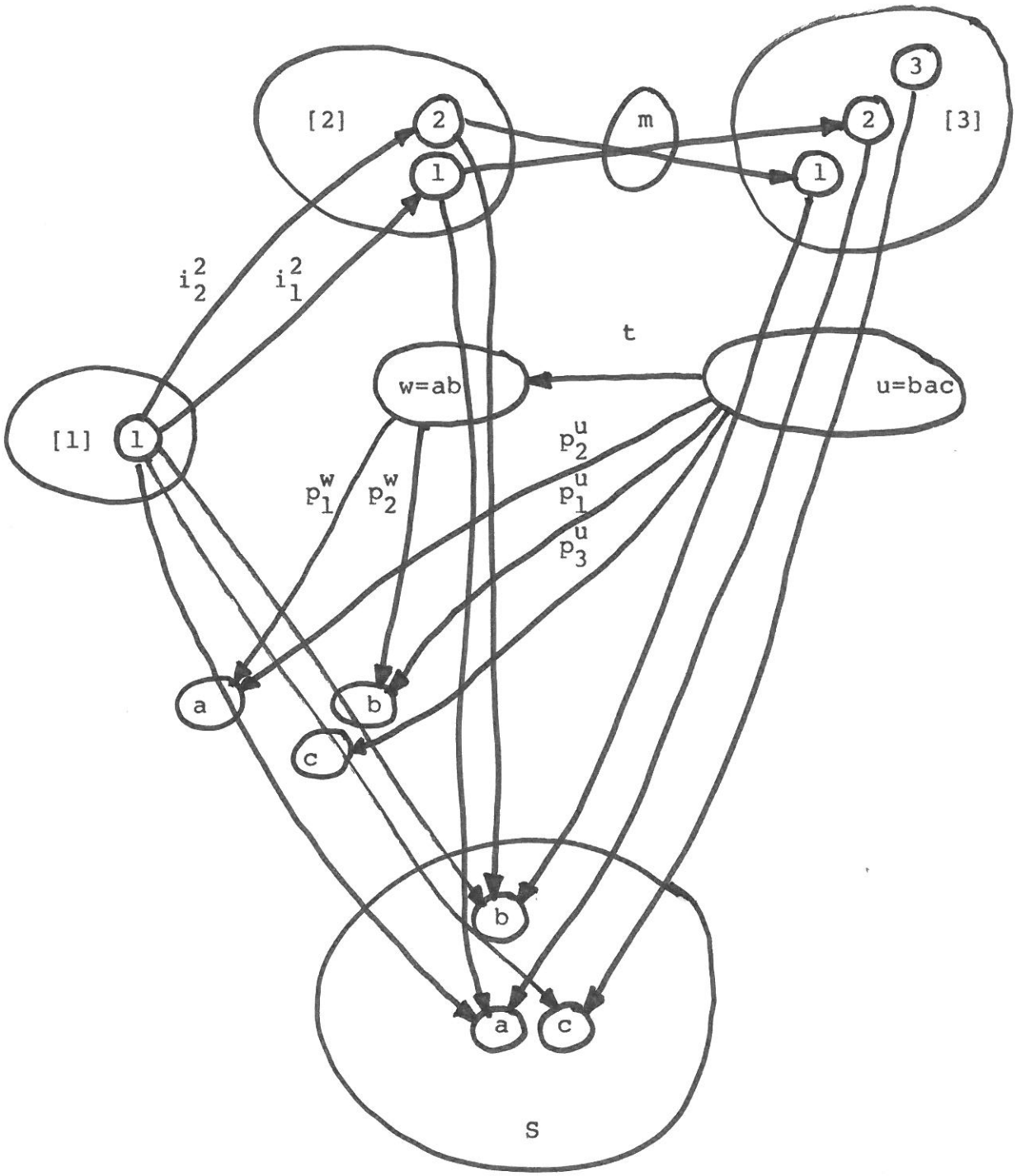
$$B^I = w' = bac \quad \text{with} \quad p^{w'}_1: w' \to b \quad p^{w'}_2: w' \to a \quad p^{w'}_3: w' \to c$$

Thus there is a unique h: $w' \to w$ which can be denoted $(p^{w'}_2, p^{w'}_1, p^{w'}_3)$, a w-tuple of morphisms from w' into the single sort strings. This morphism is an isomorphism (see 2.1). Thus all strings of the same length and with the same number of occurrences of each sort (all permutations of a string) are isomorphic.

The category constructed above has mappings as objects and some things (indirectly defined) as morphisms. Note that the morphisms are no mappings. Now we use the standard category theory trick: We forget about the nature of the objects and the morphisms and take them as plain strings and projections, tuples, etc.

The "hidden" structure is not lost. It can be remembered and used in case it is needed. Below it will be used to investigate a special class of morphisms in StS, that is, the morphisms from one string w to another string w' when the lengths of the strings are bigger or equal to 2.

Consider the following diagram



The morphism t:u →w exists (according to the definition) only if there is a mapping m: [2] →[3] such that the diagram mu = w commutes. Componentwise we can define such a mapping m: [2] →[3]: {1 ↦2, 2 ↦1}, but a corresponding

mapping from [3] to [2] cannot be defined (so that the diagram commutes) and thus there is no morphism from ab to bac.

This does not mean that there are no morphisms from some string to a longer string (there is one from ab to baa, for example). In general there is a morphism from w to u if $S_u \subseteq S_w$ because of the target tupling, i.e. all the target sorts must be found among the source sorts.

According to the universal property of the product w = ab there is a unique morphism h: u $\rightarrow$ w for each family of morphisms from u to a and b, i.e. for each pair of morphisms f_a: u $\rightarrow$ a and f_b: u $\rightarrow$ b.

Because there are only three mappings from [1] to [3], the injections, there are only three morphisms from bac to a, b and c, respectively. Thus the only family of morphisms from u to a and b are the projections $p_1^u = f_b$ and $p_2^u = f_a$.

As there is only one morphism $h = (p_2^u, p_1^u)$: u $\rightarrow$ w we must have h = t.

The morphisms $h = (f_{s_1}, \cdots, f_{s_n})$ are called <u>target tuples</u> because the indexes of the morphisms form a sequence $s_1 \cdots s_n$ which is exactly the sort string making up the target of the morphism h. Thus in the example above $h = (p_2^u, p_1^u)$ has target w = ab and the sort of p_2^u is 2u = a while p_1^u has sort 1u = b.

Because $i_1^2 m = i_2^3$ in $\underline{N}$ we must have $tp_1^w = p_2^u$ in $\underline{StS}$ according to the definition. In the same way $tp_2^w = p_1^u$. Considering that $t = (p_2^u, p_1^u)$ the composition of t and p_i^w can be seen as <u>substitution</u> of the ith element from t instead of the projection p_i^w.

From the definition of the product in a category the same fact can be seen. For a morphism $t = (t_1, \cdots, t_n): u \to w$ with $w = s_1 \cdots s_n$ we have (by the universal property of the product)

$$tp_j^w = t_j: u \to s_j.$$

Projection (to the right !) "picks" out the jth component of the morphism t (or the composed morphism is obtained by replacing p_j^w with the jth component).

In the general case composition t" = tt' of any two morphisms $t = (t_1, \cdots, t_n): u \to v$ and $t' = (t'_1, \cdots, t'_p): v \to w$ gives a morphism $t'' = (t''_1, \cdots, t''_p): u \to w$ such that each t''_i is obtained from t'_i by replacing each projection $p^v{}_j$ (in t_i) by t_j, that is

$$t''_i = t'_i[p_j^v <- t_j].$$

Example: Composition of morphisms

Let $t = (A=B, X:=5, p_1^u)$

$t' = (\text{if } p_1^v \text{ then } p_2^v \text{ else } p_3^v, A:=0;B:=0)$

u = <statement><anysort>

v = <condition><statement><statement>

w = <sequence><statement>

Now the composition $t'' = tt' = (t_1'', t_2'')$ is obtained componentwise by inserting $t_1 = A=B$ instead of p_1^v, $t_2 = X:=5$ instead of p_2^v and $t_3 = p_1^u$ instead of p_3^v. Thus the composed morphism will be

$$t'' = (\text{if } A=B \text{ then } X:=5 \text{ else } p_1^u, A:=0;B:=0).$$

* * *

The identity of an object $w \in Ob_{\underline{StS}}$ is $id_w = (p_1^w, \cdots, p_n^w)$ for $w = s_1 \cdots s_n$ because for any $t: w \to u$ where $u = s_1' \cdots s_m'$

$$t'' = id_w t = (p_1^w, \cdots, p_n^w)(t_1, \cdots, t_m) = (t_1'', \cdots, t_m'')$$

where

$$t_i'' = t_i[p_j^w <- p_j^w] \quad => \quad t'' = t$$

and (taking the identity id_u of the target)

$$t'' = tid_u = (t_1, \cdots, t_m)(p_1^u, \cdots, p_m^u) = (t_1'', \cdots, t_m'')$$

where

$$t_i'' = p_i^u[p_j^u <- t_j] = t_i \quad => \quad t'' = t.$$

Thus a w-tuple of w-projections is both right and left identity as required in 2) of the definition of a category. To make sure that StS is a category, composition should be proved associative, but that is omitted here.

The category defined above is an algebraic theory, in fact it is the initial algebraic theory in the category of all S-sorted algebraic theories.

2.3 Operators

In general an algebraic theory has some morphisms which are not projections (or injections) or derived from these by target tupling and composition. These morphisms are called operators. The string category StS is a very special algebraic theory because it has no operators at all.

From the category theoretic point of view an operator is simply a morphism but here some special notations are developed in order to make the connection between the algebraic theory and the programming language as natural as possible.

An operator symbol σ is a sequence $(u_0,\cdots,u_n)$ of strings $u_i \in U^*$, where U is a set of terminal symbols. The set of operator symbols is called Σ.

A (basic) operator is a typed operator symbol $(\sigma,(w,s)) \in R$, where $\sigma \in \Sigma$, $(w,s) \in S^* \times S$ and R is a type relation $R: \Sigma \times (S^* \times S)$. R is often divided into (w,s)-indexed subsets which are denoted $\Sigma_{w,s} = \{\sigma \mid \sigma_{w,s} \in R\}$ (operator sets).

The (basic) operator $\sigma_{w,s}$ has type w,s, arity w, sort s and rank n (where n = wlg with lg: StS $\rightarrow$ N: w $\mapsto$ [n] as the length function). No operator can have sort s = λ (because $\lambda \notin S$!), but w = λ is perfectly correct ($\lambda \in S^*$) and

defines a constant (operator) of sort s. The primitive elements (Ginsburg 1966) like letters and digits are constant operators.

In an algebraic theory there are also other operators than basic operators. Usually they are called derived operators. The arity and the source of a basic operator are the same, but the proper arity w' of a derived operator is often another sort string than the source w. Thus a more general definition of an operator is used in an algebraic theory.

An operator is a morphism $\sigma_{w,s}: w \rightarrow s$, where $w = s_1 \cdots s_n$ is a string in S^* and s is a single sort in S. Here an operator is denoted by a sequence

$$[u_0 p_1^w \cdots u_{n-1} p_n^w u_n]_{w,s}$$

where the parenthesis are in the metalanguage and p_i^w denotes a projection from the product $w = s_1 \cdots s_m$ into single sort $iw = s_i$. If m=n and the projections appear in the same order as in w, the operator can be denoted

$$[u_0 \_ u_1 \cdots u_{n-1} \_ u_n]_{w,s}.$$

In this case the first (leftmost) placeholder (_) denotes the projection p_1^w, the second p_2^w, etc. If the order of the projections in the operator differs form the order in the

string w or the number of placeholders is not equal to m (the length of the string w), the projections must be used explicitly.

The proper arity of an operator

$$[u_0 p^w_{i_1} \cdots u_{n-1} p^w_{i_n}]_{w,s}$$

is the string $i_1 w \cdots i_k w$ obtained by taking the leftmost occurrence of each index i_j in a left-to-right order.

For example, if $w = s_1 s_2 s_3 s_4$, the proper arity of the operator

$$[u_0 p^w_3 u_1 p^w_1 u_2 p^w_3 u_3]_{w,s}$$

is $w' = s_3 s_1$.

There are many problems connected to the notation for an operator, but these will be discussed in section 2.6.

Two special operators are very frequent: inclusion (coercion) and catenation. In the semantics of a programming language these operators may denote operations which are different from inclusion and catenation, but in the syntax they really denote inclusion and catenation if this notation is used.

The inclusion (incl) is an operator $[\lambda\_\lambda]_{s,s'}$ which means that the composition $t_{w,s}[\lambda\_\lambda]_{s,s'}$ (where $t_{w,s}$ is any morphism for some string w) is defined as $[\lambda t\lambda]_{w,s'}$, that is, only the sort of the morphism t is changed, not the "name". This corresponds to the intuitive meaning of inclusion in syntax.

The catenation (cat) is an operator $[\lambda\_\lambda\_\lambda]_{s_1s_2,s}$ which means that the composition $(a,b)_{w,s_1s_2}[\lambda\_\lambda\_\lambda]_{s_1s_2,s}$ (where a and b are morphisms from some string w to s_1 and s_2, respectively) is defined as $[\lambda a\lambda b\lambda]_{w,s}$, that is, the "names" of the components a and b are catenated (see example Derived constants 2 in 2.5).

Projections can be seen as special cases of operators using the more general notation for an operator. Consider a projection p_i^w: $w \rightarrow s_i$, where $s_i = iw$ and $w = s_1 \cdots s_n$, to be an operator

$$p_i^w = [\lambda p_i^w \lambda]_{w,s_i} \quad \text{(Note that } p_\lambda^w = [\lambda]_{w,\lambda} \text{ !)}$$

with proper arity s_i. Thus composition

$$t_{u,w}p_i^w = (t_1,\cdots,t_n)[\lambda p_i^w\lambda] = [\lambda t_i\lambda] = t_{u,s_i}$$

and the operator "picks" the ith element out of the w-tuple.

Another possibility is to introduce killers γ_γ which kill the argument. Thus $\gamma t_j \gamma = \lambda$, the empty string, for any t_j and the projection could be written

$$p_i^w = [\gamma p_1^w \gamma \cdots \lambda p_i^w \lambda \cdots \gamma p_n^w \gamma]_{w, s_i}.$$

In the proper arity the killed arguments are ignored

In the special case p_λ^w, the lambda projection, the "0th" element is picked out, i.e.

$$p_\lambda^w = [\gamma p_1^w \gamma \cdots \gamma p_i^w \gamma]$$

is killing all real elements in the w-tuple and

$$t_{u,w} p_\lambda^w = [\lambda]_{u, \lambda}$$

which is denoted p_λ^u, the unique morphism from any product u into the empty product λ.

This is only a preliminary definition of operators. Some problems are discussed in section 2.6.

2.4 Syntax and signatures

The operators defined in the previous section are used to model some "phrases" in programming languages. Here the connection to the well-known BNF syntax is established. The presentation is restricted to context-free grammars only. It is very possible that the same method can be used also for more general kinds of languages, but that would be too far from a short introduction. The programming languages considered are phrase-structure languages (Ginsburg 1966).

A phrase structure language L is a subset of U^* (where U is a set of terminal symbols - an alphabet), if L can be generated by a (phrase structure) grammar G = (N,U,P), where N is the set of nonterminals, P is the set of productions and $N \cap U = \emptyset$.

A context-free grammar has only productions of the form $s \rightarrow u_0 s_1 u_1 \cdots u_{n-1} s_n u_n$ with $u_i \in U^*$ and $s, s_i \in N$. A context-free phrase structure language is generated by such a grammar and is sometimes called "Algol-like" (Ginsburg 1966).

It has been shown that programming languages defined by BNF are equivalent to context-free phrase languages (Ginsburg 1966). The notation for a grammar is different here because it is based on an algebraic model. It can, however, easily

be made equivalent to the definition of a grammar used by Ginsburg in the following way:

Define the set $V = N \cup U$, the set $\Sigma_1 = U$, the set $P = P$ and $\sigma = s_p$, where $s_p \in N$ is the nonterminal denoting a "sentence" in the language (usually s_p = <program>). Then $G' = (V, \Sigma_1, P, \sigma)$ is a grammar as defined by (Ginsburg 1975).

The "words" in the language are generated in a different way here and the "start symbol" σ of Ginsburg is here an index of the "final" set of words. Note that σ is used in this report to denote an operator symbol and not a nonterminal (as in Ginsburgs grammar).

There are, unfortunately, many different BNF notations. Here an explicit BNF with angle brackets is used. Other forms can easily be transformed into this form.

An <u>explicit</u> BNF is here defined as having only "equations" of the type

$$<x>::=u_0<x_1>u_1\cdots<x_n>u_n,$$

that is, no alternatives (using |, { }, etc.) or options (using [] etc.) are allowed.

A syntax given in explicit BNF is transformed into a grammar G = (N,U,P) in the following way:

1) The set N of nonterminals is the union of all words within angle brackets <>,

2) the set U of terminals is the union of all symbols in strings outside the angle brackets (except ::=),

3) the set P of productions is obtained from the "equations" of BNF after replacing ::= with →.

Example: Transforming BNF into a grammar G

Consider the BNF syntax for language A in the Appendix.We obtain the following sets:

1) N = {<letter>,<digit>,<digit string>,···,<program>}

2) U = {A,B,C,···,a,b,c,···,0,1,···,9}

Before extracting productions, rewrite the equations in explicit form by removing alternatives and optional notations e.g.

<identifier>::=<letter>|<identifier>$\{^{\text{<letter>}}_{\text{<digit>}}\}$

is rewritten as

<identifier>::=<letter>

<identifier>::=<identifier><letter>

<identifier>::=<identifier><digit>.

Now the productions are obtained directly

3) P = {<letter>→A,···,<identifier> →<letter>,···,
<goto>→goto <label>,···,
<program>→program <identifier>;<sequence> end}

If somebody wants a "start symbol" in the grammar G, it would be <program> in this example.

***

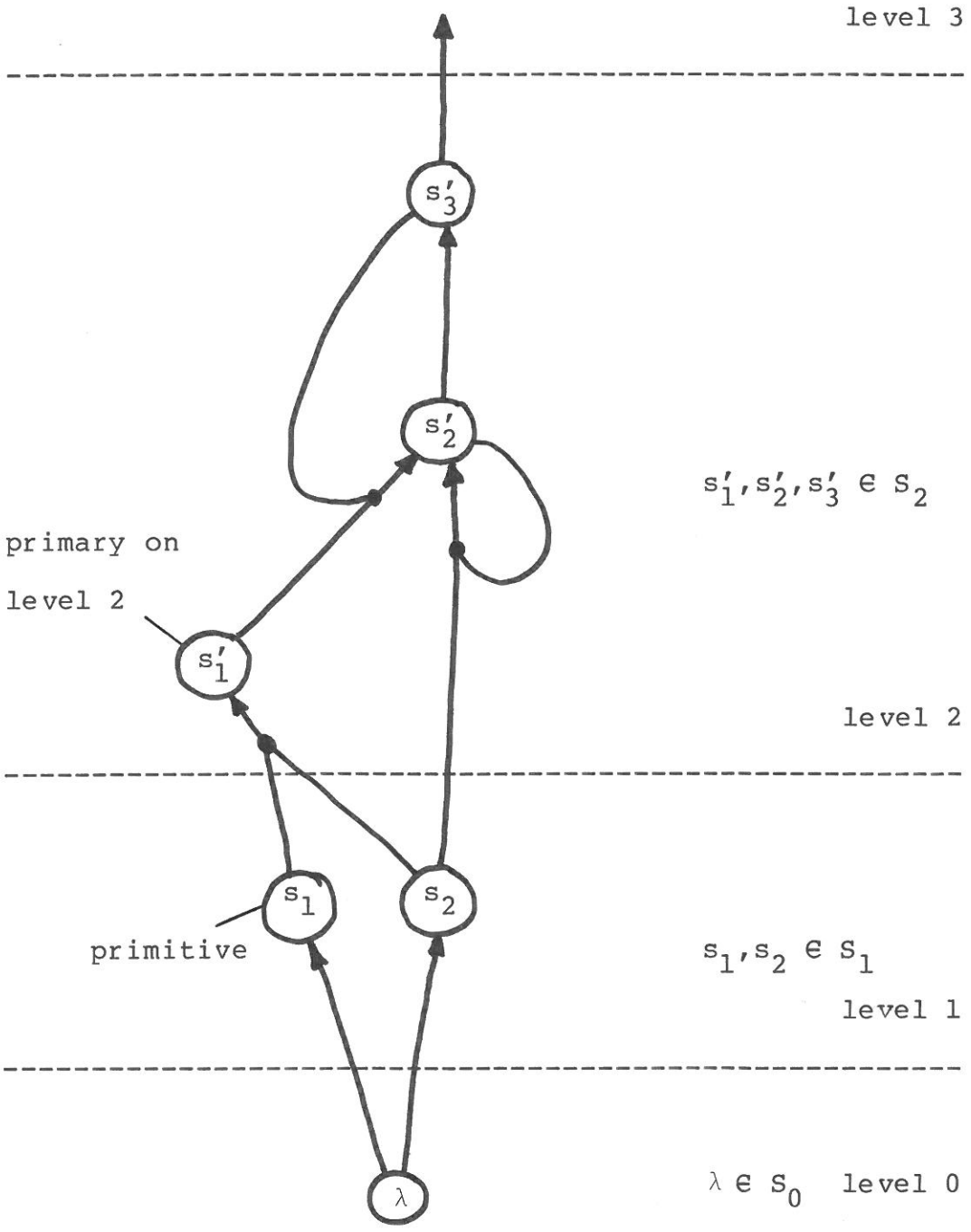
From a grammar G = (N,U,P) (or directly from BNF in explicit form) it is easy to construct a heterogeneous (or many-sorted) signature (or scheme of operators).

A S-sorted signature $S\Sigma$ (or simply Σ) consists of a sort set S, an operator symbol set Σ and a type relation $R: \Sigma \times (S^* \times S)$.

In fact a signature is a labelled directed graph (a diagram) with the nodes labelled from the sort set S and the arrows labelled from the set P of productions.

A sort hierarchy can be defined for the signature $S\Sigma$ so that a sort s' belongs to level j if the operator sets $\Sigma_{w,s'}$ are empty for all w with sorts from higher levels, that is,

$$s' \in S_i \text{ if } \Sigma_{w,s'} = \emptyset \text{ for all } w = s_1 \cdots s_n$$
$$\text{with some } s_j \in S_k \ (1<=j<=n) \text{ for } k>i.$$



In the graph above it is easy to see that elements of a sort s can be constructed only from sorts on the same or a lower level. If the sort is primary, then only elements from a lower level are used. The arrows in the graph can cross the level borders only from bottom up.

If $\Sigma_{w,s'}$ is empty also for all w which contain sorts from the same level (k=>i), then the sort s' is <u>primary</u> on the level i. The sort $\lambda \in S_0$ and every primary sort on level 1 is <u>primitive</u> in $S\Sigma$, that is, $s' \in S_1$ is primitive if all $\Sigma_{w,s'} = \emptyset$ for $w \neq \lambda$.

Thus <letter> and <digit> are primitive sorts in the graph in the Appendix. Notice the difference between primitive sorts and primitive elements (constants) and especially the difference between the empty terminal string $\lambda_u \in U^*$ and the empty sort string $\lambda_s \in S^*$.

A sort $s' \in S$ is <u>final</u> if $\Sigma_{vs'u,s} = \emptyset$ for all $s \in S$, $v,u \in S^*$. Usually there is only one final sort in the signature of a programming language (e.g. <program>) and it is equivalent to the "start symbol" σ of a grammar $G = (V, \Sigma_1, P, \sigma)$.

What about the case when $\Sigma_{w,s'} = \emptyset$ for <u>all</u> w without exceptions ? Then s' is not defined ! This is <u>not</u> equivalent to a grammar G with the production $s' \rightarrow \lambda$ (the empty string) for sort s' and no other productions where s' appears on the left side.

If a programming language has an "empty statement", e.g.

<statement>::=λ_u|<assignment>|<conditional>| etc.

where λ_u is the empty string in U^*, it can be rewritten as

<statement>::=<empty>|<assignment>|<conditional>| etc.

<empty>::=λ_u

and the production <empty>→ λ_u is the only production in the corresponding grammar with <empty> to the left.

This production defines an operator $[\lambda_u]_{\lambda_s,\text{<statement>}}$ which is a constant with the empty string as operator symbol. From the categorial point of view it does not matter if the operator symbol is the empty string or any other string like continue or skip.

Thus every sort s' defined exclusively by the production s'→ λ_u is a primitive sort (in S_1).

From a signature SΣ an algebraic theory $\underline{T} = \underline{StS}_\Sigma$ is defined using the string category StS of strings S* in the sort set S of that signature. It is constructed by simply adding the operator sets to the corresponding morphism sets of StS

$$Mor_{\underline{T}}(w,s) = Mor_{\underline{StS}}(w,s) \cup \Sigma_{w,s}$$

where the union should be disjoint.

The universal property of products (see 2.1) forces tuples to be constructed out of these additional operators. Thus target tupling and composition act like "constructors" creating (usually infinitely many) new morphisms out of the string category morphisms and operators added to StS.

2.5 The algebraic theory

An algebraic theory is a category constructed from the signature (via the string category) as explained in the previous section. Thus it has the same objects as $\underline{StS}$ (the sort strings in S). The morphism sets are, however, more interesting and will be investigated in detail in this section.

Generally speaking, there are four kinds of morphisms in the algebraic theory $\underline{T} = \underline{StS}_{\Sigma}$:

- projections
- (basic) operators (from Σ)
- tuples
- composed morphisms.

The projections are the same in both $\underline{StS}$ and $\underline{T}$ because they have the same products (sort strings), but $\underline{StS}$ has no real operators and only tuples of projections as morphisms.

Let us divide the objects in the algebraic theories into three different classes:

1) the <u>empty sort string</u> λ (or λ_s),
2) the <u>single</u> sorts (strings of length 1). This subset is isomorphic with the sort set S,
3) the <u>proper sort strings</u> w (length >1).

2.5.1 Morphisms in StS

Let us first consider the morphisms which are in Mor_{StS} and let us take a look on morphisms with target λ. As was shown in Section 2.2 there is a unique morphism from any object u to λ, the lambda projection p_λ^u because λ is terminal object in StS.

A morphism in StS with a single sort s´ as target can be a projection into itself, i.e. an identity p_1^s(if s=s´), or a projection p_i^w from some proper string w with iw = s´.

If a morphism in StS has a proper string w´ as target it can be a w´-tuple of projections p_i^u, often denoted $(p_i^u)^{w´}$. If u=s (a single sort) then it is a w´-tuple of identities p_1^s if w´ = ss···s, i.e. a string with occurrences of s only.

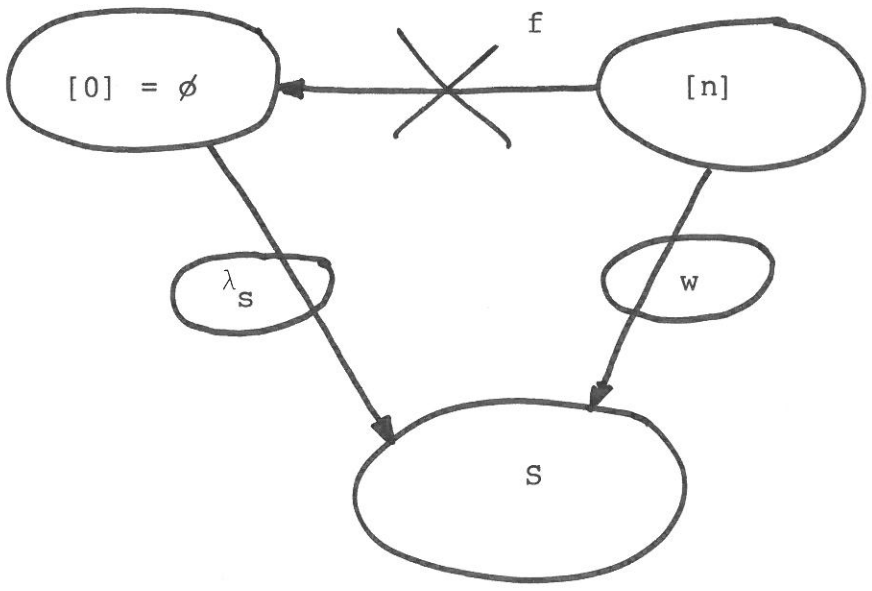
The morphism types of StS can be presented in the following way:

		target			
		λ	s´	w´	
	λ	p_λ^λ			
source	s	p_λ^s	p_1^s(s=s´!)	$(p_1^s)^{w´}$	(w´=ss···s!)
	w	p_λ^w	p_i^w(iw=s´!)	$(p_i^w)^{w´}$	

Notice that there are some conditions for the existence of certain morphisms: A morphism exists

- from s to s´ only if s = s´,
- from w to s´ only if iw = s´ for some i, where 1<=i<=n and n = wlg,
- from s to w´ only if w´ = ss···s
- from w to w´ only if $S_{w'} \subseteq S_w$.

The only morphism with source λ is the lambda identity. The reason is that there is no mapping from a set [n] to the empty set $\emptyset$ except if n = 0, that is we have the situation



2.5.2 Morphisms in a general algebraic theory T

In a general algebraic theory all the above mentioned morphisms can be found but also other morphisms "generated" by the operators in the signature. Because the signature can

have no operators with target λ, that is $\Sigma_{w,\lambda} = \emptyset$ for all w (see definition in 2.4), the only morphisms from any object w into the empty sort λ are the lambda projections p^w_λ and for all $w \in S^*$ we have

$$Mor_{\underline{T}}(w,\lambda) = \{p^w_\lambda\}.$$

Morphisms with target s (a single sort) can be operators in some set $\Sigma_{w,s}$. If $w = \lambda$ then it is a constant (operator), i.e. a terminal word. But there might also be compositions which have target s. An important case is the composition of an u-tuple of constants and an operator from u to s, i.e.

$$t_{\lambda,u}\sigma_{u,s}$$

which gives rise to a new "derived" constant (operator). Other compositions create derived operators in general (see next section).

Example: Derived constants 1

In language A (Appendix) there are constants A,B,C,... which are morphisms in $Mor_{\underline{T}}(\lambda,\langle\text{letter}\rangle)$. There is also an operator $[\lambda\_\lambda]_{\langle\text{letter}\rangle,\langle\text{identifier}\rangle}$. Thus by composition there is a morphism

$$A \circ \lambda\_\lambda = \lambda A\lambda = A \in Mor_{\underline{T}}(\lambda,\langle\text{identifier}\rangle)$$

(or strictly

$$A_{\lambda,\langle\text{letter}\rangle} \circ [\lambda p_1^{\langle\text{letter}\rangle}\lambda]_{\langle\text{letter}\rangle,\langle\text{identifier}\rangle} = A_{\lambda,\langle\text{identifier}\rangle})$$

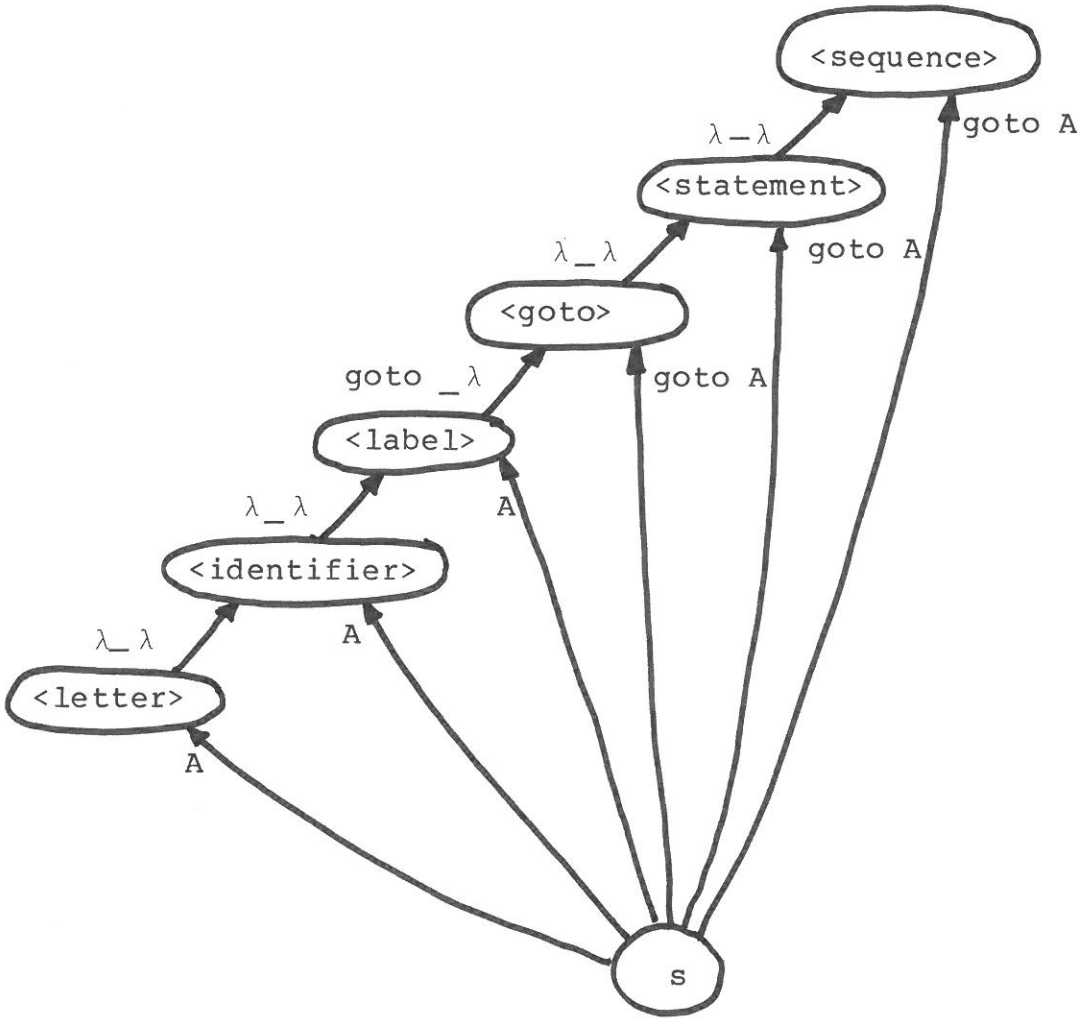
because composition with the mixfix notation λ_λ of the (syntactic) inclusion operator is simply substitution (see 2.3). Another operator $[\lambda\_\lambda]_{\langle\text{identifier}\rangle,\langle\text{label}\rangle}$ will take the

letter A into $Mor_{\underline{T}}(\lambda,\langle label\rangle)$ and if we apply the operator goto $_{\lambda}{}_{\langle label\rangle,\langle goto\rangle}$ we get the goto statement

goto A $\in Mor_{\underline{T}}(\lambda,\langle goto\rangle)$

which even can be included into the morphism sets $Mor_{\underline{T}}(\lambda,\langle sequence\rangle)$ and $Mor_{\underline{T}}(\lambda,\langle program\rangle)$ by the corresponding "inclusions".

Thus in a graphical notation for the constant L and the above mentioned operators we have



where goto A ∈ $Mor_{\underline{T}}(\lambda,\langle\text{sequence}\rangle)$ is the composition

$$A_{\lambda,\underline{le}}\circ[\lambda\_\lambda]_{le,\underline{id}}\circ[\lambda\_\lambda]_{\underline{id},\underline{la}}\circ[\text{goto }\_\lambda]_{\underline{la},g}\circ[\lambda\_\lambda]_{g,\underline{st}}\circ[\lambda\_\lambda]_{\underline{st},\underline{se}}$$

using the short notations

$\underline{le}$ = <letter>

$\underline{id}$ = <identifier>

$\underline{la}$ = <label>

g = <goto>

$\underline{st}$ = <statement>

$\underline{se}$ = <sequence> .

This graph naturally shows only a few morphisms and objects in the category $\underline{T}$ (there are infinitely many).

* * *

The idea with this example is to show that the terminal "words" of a single sort s in the language are morphisms from the empty string into that sort. Thus the set of all programs is the morphism set $Mor_{\underline{T}}(\lambda,\langle\text{program}\rangle)$ in the algebraic theory constructed from the syntax (in explicit BNF) of the language.

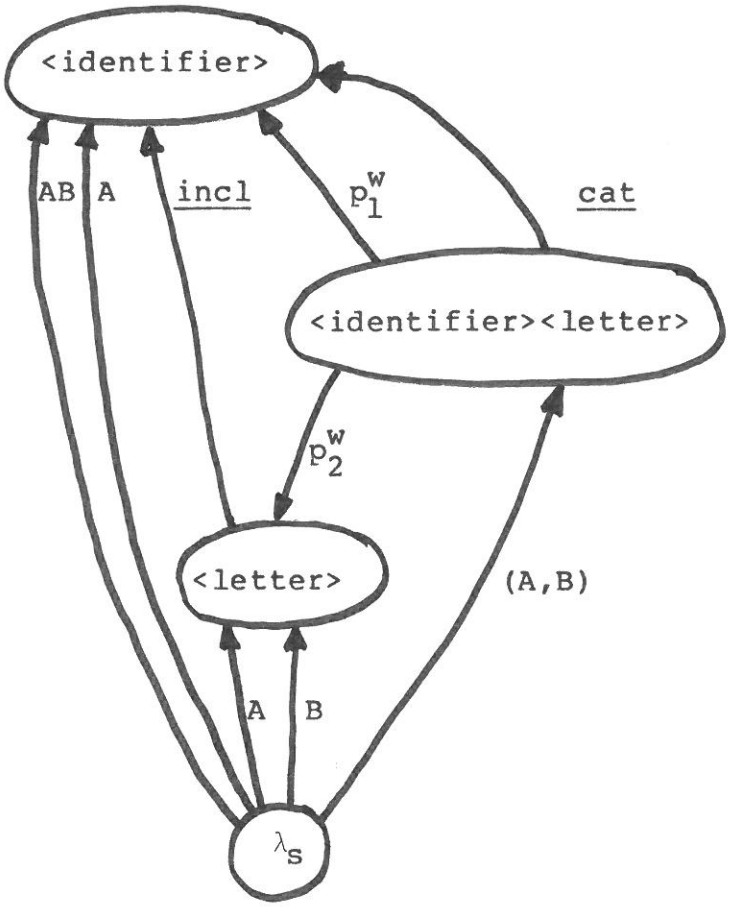
The morphism sets $Mor_{\underline{T}}(\lambda,w)$ where w is a proper string contain only w-tuples of morphisms in $Mor_{\underline{T}}(\lambda,s_i)$ if $w=s_1\cdots s_n$. Composition of a w-tuple with an operator $\sigma_{w,s'}$ will give a new morphism in $Mor_{\underline{T}}(\lambda,s')$. Thus composition means that the operator is "applied" to "arguments" (in a w-tuple) giving a new "value" of sort s'.

Example: Derived constants 2

In the language A (see Appendix) there is an operator $[\lambda\_\lambda\_\lambda]_{\text{<identifier><letter>,<identifier>}}$ and constant $B_{\lambda,\text{<letter>}}$. In the previous example it was shown that $A \in \text{Mor}_{\underline{T}}(\lambda,\text{<identifier>})$. Thus by target tupling there is

$$(A,B) \in \text{Mor}_{\underline{T}}(\lambda,\text{<identifier><letter>}).$$

The composition of the operator with this tuple gives



where AB denotes the composition

$$(A,B)_{\lambda,\underline{idle}} \circ [\lambda\_\lambda\_\lambda]_{\underline{idle},\underline{id}}.$$

* * *

Certainly there are infinitely many morphisms in $Mor_{\underline{T}}(\lambda,\langle identifier\rangle)$, in fact all non-empty strings in the constants of sorts <letter> and <digit> beginning with a letter. The empty string λ_u is not in this morphism set unless it is a constant of sort <letter>.

Because the morphisms are not mappings and the objects not sets there is no "empty mapping" from λ_s. If the empty string λ_u is included it has its own "arrow" and from the categorial point of view it does not matter if its "name" is the empty string or any other string (as long as it is disjoint from the other morphisms in the same set).

In general the morphisms with target s are operators, either basic or derived (for proper derived operators, i.e. not constants, see next section). They have a very rich structure which will be needed when modelling programming languages.

One interesting class of morphisms is defined by composition of lambda projections and constants (including derived constants), i.e.

$$p_\lambda^u t_{\lambda,s}$$

which "transfers the source of all constants from λ to u". Because p_λ^u is unique for every object $u \in Ob_{\underline{T}}$ the morphism set $Mor_{\underline{T}}(\lambda,s)$ is mapped injectively (one-to-one) into $Mor_{\underline{T}}(u,s)$. The same goes for morphisms from λ to any string w.

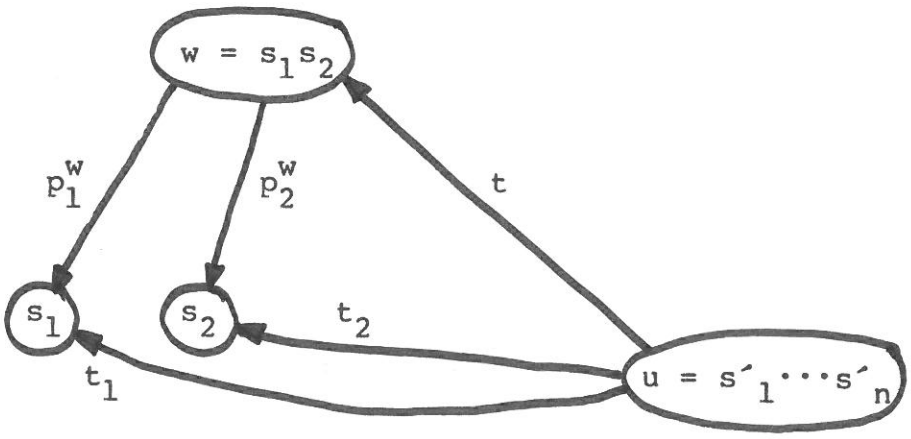
According to the definition of composition, each projection p_i^w in t should be replaced by the corresponding component t_i in the lambda projection. Because the only component in the lambda projection is the empty component, the morphism t will remain unchanged - only the source will be set to w instead of λ. Thus

$$\text{if } t \in Mor_{\underline{T}}(\lambda,u) \text{ then } t \in Mor_{\underline{T}}(w,u)$$
$$\text{for all } u,w \in Ob_{\underline{T}}.$$

Strictly every morphism is indexed by its source and target (the disjointifying trick), but to increase the readability the indexes are dropped if the source and target are clear from the context.

All the morphisms with target w (a proper string) are w-tuples of morphisms from some string u to the single sorts s_i in the string w. Every morphism can be seen as a w-tuple but if w is equal to λ or a single sort s the tupling is trivial.

A morphism from λ to w is a w-tuple of constants while morphisms from some string u can be w-tuples of basic or derived operators, projections, etc. In the general case there is a morphism (target tuple) $t: u \rightarrow w$ only if there are morphisms $t_i: u \rightarrow s_i$ for all s_i in Σ_w. Thus in the following example there must be $t_1: u \rightarrow s_1$ and $t_2: u \rightarrow s_2$.



The universal property of a product w in a category $\underline{T}$ requires that there is a unique morphism h: u $\rightarrow$w for each w-tuple of morphisms f: u $\rightarrow s_j$, where w = $s_1 \cdots s_n$ and 1<=j<=n. Thus we can write

$$Mor_{\underline{T}}(u,w) = (Mor_{\underline{T}}(u,s_j))^w$$

for h = $(f)^w$.

2.6 Derived operators in an algebraic theory

In order to show how a derived operator is "constructed" a simple example is presented. No "new" morphisms are added to the morphism sets of the algebraic theory $\underline{T} = \underline{St}S$. The derived operator is shown to <u>be</u> in the morphism set. In fact <u>all derived operators are in the algebraic theory</u> constructed as in Section 2.4.

<u>Example</u>: Derived operator

Consider a signature S :

$$\text{sort set } S = \{s,a,b,c,d,e,f,\cdots\}$$

$$\text{operators } \sigma = [A\_B\_C]_{ab,s}$$

$$\sigma_1 = [M\_N\_O]_{cd,a}$$

$$\sigma_2 = [R\_S\_T]_{ef,b}$$

In the algebraic theory $\underline{St}S_\Sigma$ there is an object w = cdef. Because it is a product we have projections

$$p_1^w: w \to c$$

$$p_2^w: w \to d$$

$$p_3^w: w \to e$$

$$p_4^w: w \to f$$

and target tuples of these projections

$$t_1 = (p_1^w, p_2^w): w \to cd$$

$$t_2 = (p_3^w, p_4^w): w \to ef.$$

These tuples can be composed with operators of corresponding arity

$$t_3 = t_1\sigma_1 = Mp_1^wNp_2^wO: w \to a$$

$$t_4 = t_2\sigma_2 = Rp_3^wSp_4^wT: w \to b$$

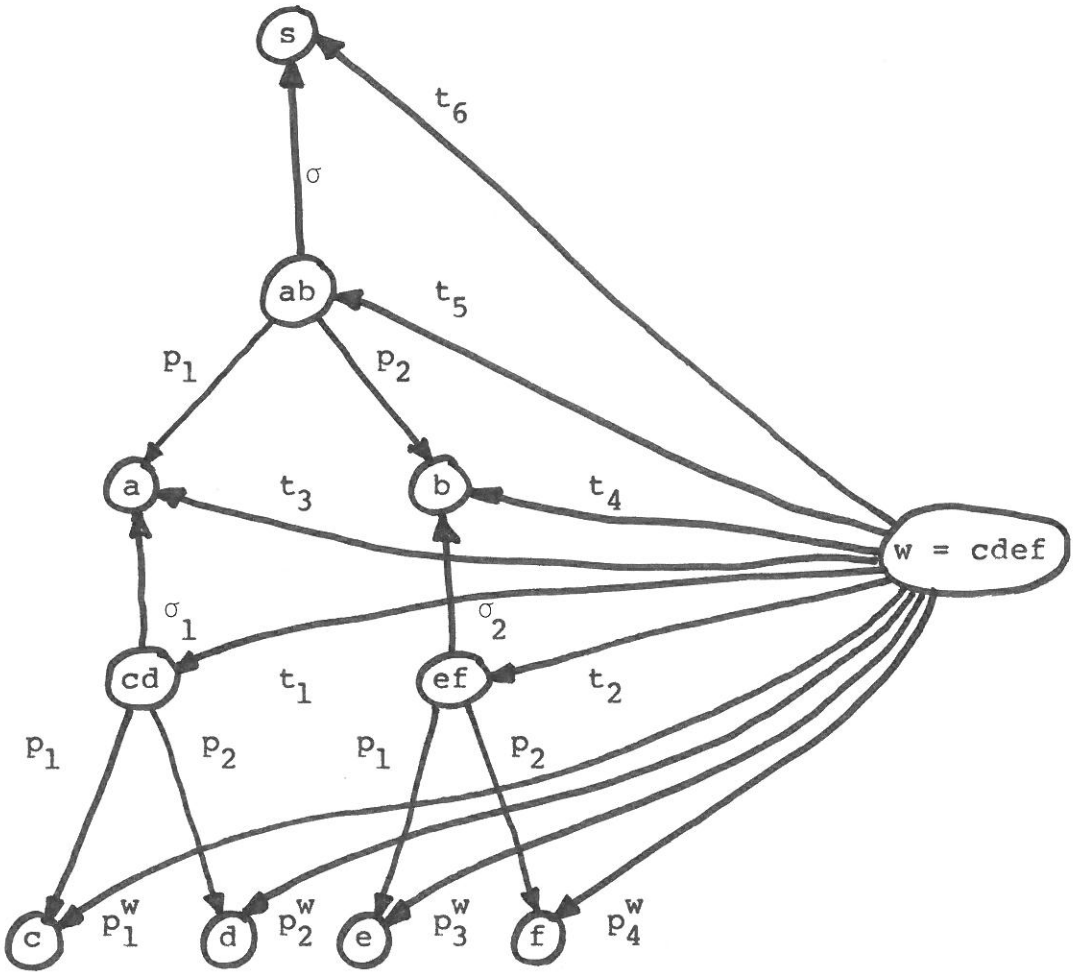
and target tupling of the compositions gives

$$t_5 = (t_3, t_4): w \to ab$$

which can be composed with σ giving

$$t_6 = t_5\sigma = (t_3, t_4)\sigma = A\underbrace{Mp_1^wNp_2^wO}_{t_3}B\underbrace{Rp_3^wSp_4^wT}_{t_4}C: w \to s$$

In graphical form these morphisms are inserted as follows:



If t_6 is rewritten with placeholders (_) instead of the projections (see 2.3) we get the derived operator

$$[AM\_N\_OBR\_S\_TC]_{w,s}.$$

Thus tupling is used to "climb" up to the source of the projections and composition to "go up" to the target of the operators.

* * *

A derived operator of type w,s is constructed from the projections of the product w using target tupling and composition until the sort s is reached (if it is possible).

In this work application of a function f to an argument x is written

$$xf$$

which looks like composition of morphisms

$$t_5\sigma$$

where $t_5 = (t_1\sigma_1, t_2\sigma_2)$ and t_1, t_2 are tuples of projections.

If the operator σ is "applied" to the "tuple" (σ_1, σ_2) of "type" cdef,ab we get

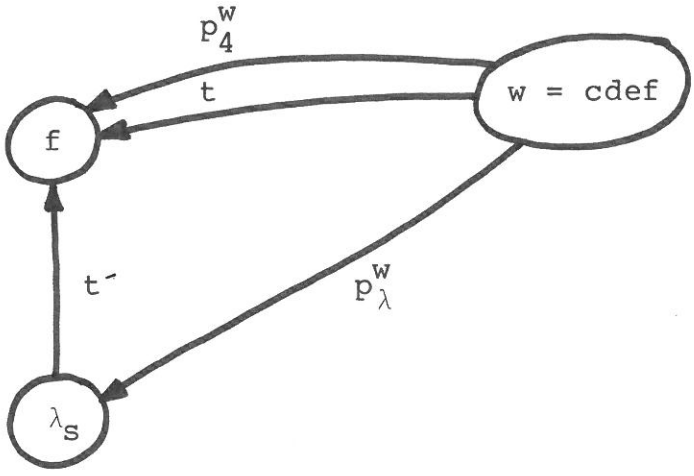
$$(\sigma_1,\sigma_2)\sigma = [\underbrace{AM\_N\_}_{\sigma_1}\underbrace{OBR\_S\_}_{\sigma_2}TC]_{w,s}$$

which is the same as the derived operator in the example. Thus it is rather easy to find derived operators.

The derived operator in the example above is, however, of a special kind because each projection appears only once in the derived operator. In general it should be possible to include multiple occurrences of the same projection and to have an arity which is different from the proper arity.

Example: Derived operator 2

Suppose that we want a derived operator like in example "Derived operator 1", but with a terminal string instead of p_4^w. A terminal string is a morphism with source λ_s (the empty sort), i.e. a constant (possibly derived) operator. Thus we have the following diagram:



where $t' = u'$ and $u' \in U^*$ and $t = p_\lambda^w t'$. If the derived operator is generated using t instead of p_4^w the result will be

$$[AMp_1^wNp_2^wOBRp_3^wSu'TC]_{w,s}$$

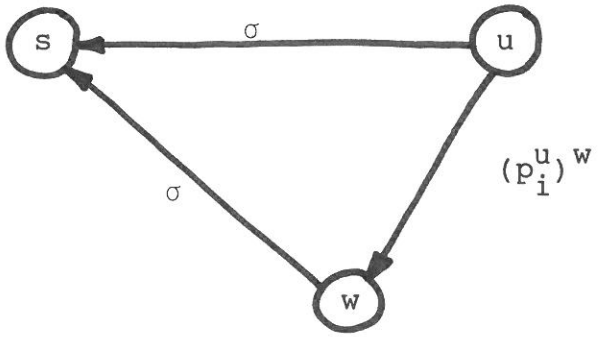
because t has the same "name", i.e. the same operator symbol (in this case a single terminal string u´) as t´ (see 2.5).

The derived operator has the arity w (according to the derivation) although it intuitively should be $w' = cde$. It is easy to derive an operator with the "right" (or proper) arity because

$$p^{w'}_{\lambda}\, t'_{\lambda,f} = t'_{w',f}$$

can be used instead of p^{w}_{4} if the whole derivation is performed with respect to w' instead of w.

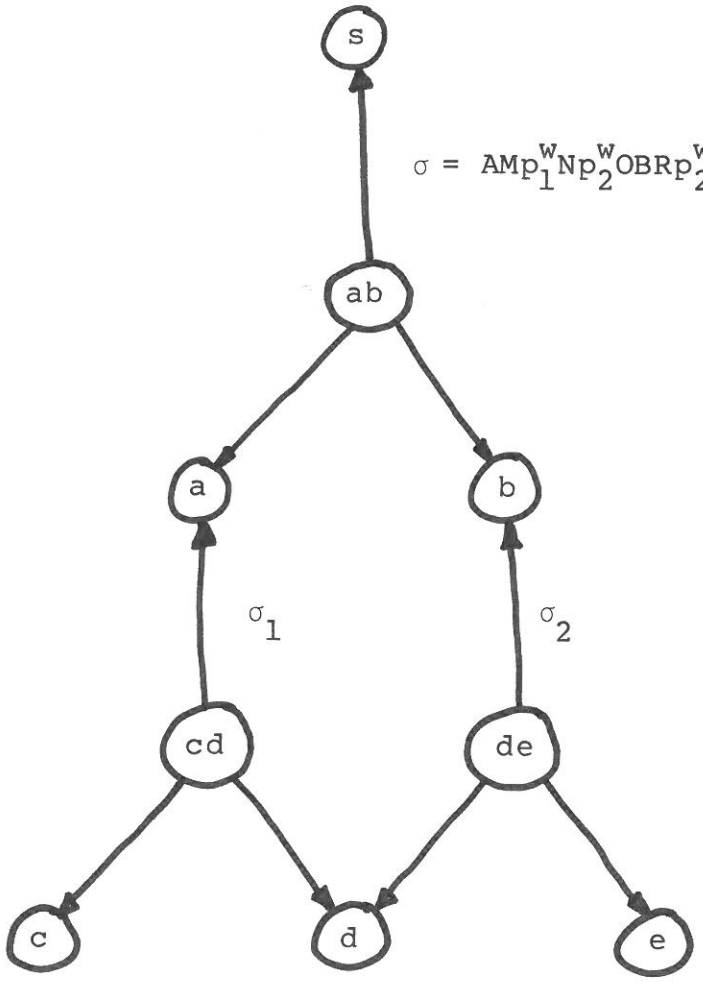
A derived operator with proper type w,s can be formed with respect to every string u which has a sort set S_u such that $S_w \subseteq S_u$ (see 2.2).



The composition $(p^u_i)^w\sigma$ gives a derived operator σ which has projections p^u_i instead of projections p^w_i in the original operator. Here both operators have the same name which is correct as long as the operator symbol can be used as the name (with placeholders _ instead of projection names). If this is not possible, e.g. when the order of the projections does not follow the order in the target string, the names of the operators are different (see 2.3).

Example: Derived operator with multiple occurrences

Consider the case when two operators have arities which are overlapping, i.e. for σ_1 of type w_1,a and σ_2 of type w_2,b there are s_i in w_1 and s_j in w_2 such that $s_i = s_j$. Then we have the situation in the following diagram.



The operator derived with respect to w = cde has two occurrences of the projection p_2^w.

* * *

The type of an operator with multiple occurrences of the same projection is defined from the sequence of the left-most occurrences of the projections in the operator.

This definition is a bit arbitrary because the "same" derived operator could be constructed from another string (e.g. u=dce), which is a permutation of the first. In the algebraic theory these morphisms are different, but there is an isomorphism between them. The isomorphism is the tuple $t = (p_2^w, p_1^w, p_3^w)$: w →u with "invers" (p_2^u, p_1^u, p_3^u): u →w and the derived operators have the following relation

$$\lceil AMp_2^u Np_1^u OBRp_1^u Sp_3^u TC\rceil_{u,s} = (p_2^u, p_1^u, p_3^u)\lceil AMp_1^w Np_2^w OBRp_2^w Sp_3^w TC\rceil_{w,s}$$

which is consistent with the definition of composition (see 2.2).

3. FUNCTORS AS MODELS OF ALGEBRAS

In order to express relations between categories a pair of mappings (a functor) is introduced. These mappings must satisfy two main conditions - they should preserve identities and composition. Thus the following definition can be used:

A (covariant) functor F from the category $\underline{K}$ to the category $\underline{L}$ is a pair of mappings $F_{Ob}: Ob_{\underline{K}} \rightarrow Ob_{\underline{L}}$ and $F_{Mor}: Mor_{\underline{K}} \rightarrow Mor_{\underline{K}}$ satisfying:

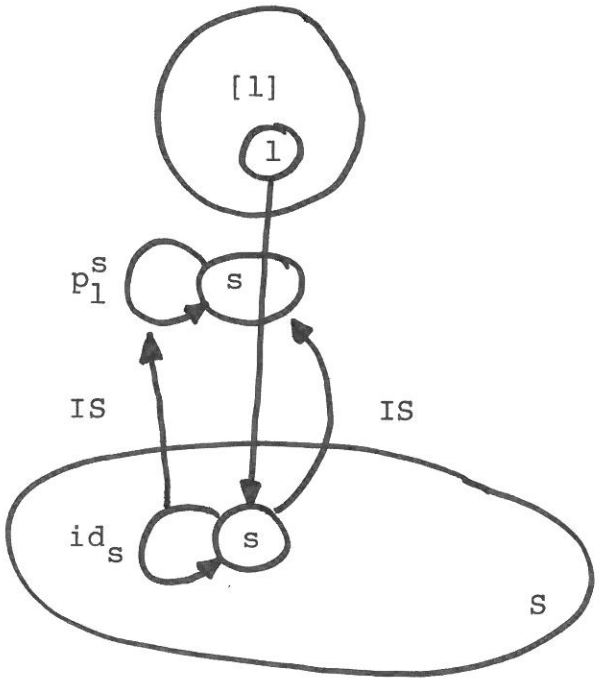
1) $(id_A)F_{Mor} = id_{AF_{Ob}}$ for every object $A \in Ob_{\underline{K}}$

2) if $f: A \rightarrow B$ and $g: B \rightarrow C$ in $\underline{K}$ then $(fg)F_{Mor} = (fF_{Mor})(gF_{Mor}): AF_{Ob} \rightarrow CF_{Ob}$ in $\underline{L}$.

Usually a functor is denoted $F: \underline{K} \rightarrow \underline{L}$. Because a functor preserves identities and composition it also preserves isomorphism. Normally a functor is covariant which means that the "direction of the arrows" is preserved but sometimes a contravariant functor is used which reverses the direction and composition is $(fg)F_{Mor} = (gF_{Mor})(fF_{Mor}): CF_{Ob} \rightarrow AF_{Ob}$ in $\underline{L}$ in that case.

Example: Injection of sorts

A very simple functor would be the injection from the sort set S (seen as a discrete category) to the string category StS. This functor IS: S → StS with IS_{Ob}: s ↦ ([1] → s) maps sorts on sort strings. It is not inclusion because s and [1] → s are different, although the unique mapping [1] → s: 1 ↦ s usually is denoted simply by s (see 2.2).



The morphism mapping is also an injection IS_{Mor}: id_s ↦ p_1^s, where s is a sort. Because there are no other morphisms than the identities in a discrete category the functor is completely defined.

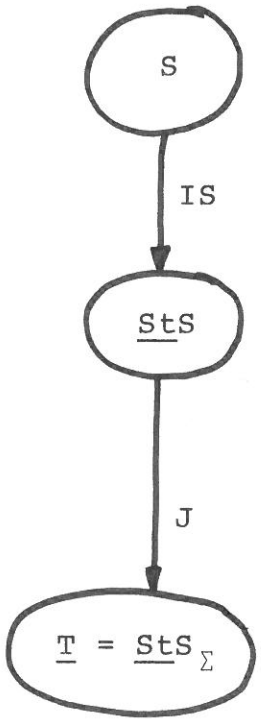
* * *

Example: Algebraic theory embedding

There is a (covariant) functor J: StS $\rightarrow$ T, which is an inclusion if T = StS_{Σ} defined earlier. Some authors like (Pareigis 1973) define an algebraic theory as a covariant functor which is bijective on objects and which preserves finite products.

The algebraic theory used here is, however, heterogeneous while Pareigis defined a homogeneous theory using N instead of StS. The ADJ group (GTWW 1975) defines a many-sorted theory as a functor but their functor preserves coproducts (a matter of taste).

Thus we have the following categories and functors:



sort set (discrete category)

injection of sorts

S-sorted string category
(with target tuples)

inclusion of strings

S-sorted algebraic theory
with operator set

A functor always preserves isomorphism but this does not mean that the functor itself always is isomorphic.

A functor F: $\underline{K} \rightarrow \underline{L}$ is isomorpic (bijective) if F_{Mor}: $Mor_{\underline{K}} \rightarrow Mor_{\underline{L}}$ is a bijection. Because it also maps identities bijectively, the object mapping F_{Ob}: $Ob_{\underline{K}} \rightarrow Ob_{\underline{L}}$ must be a bijection. Categories $\underline{K}$ and $\underline{L}$ are isomorphic if there is an isomorphic functor between them. The renaming of objects or morphisms induces isomorphic functors and categories.

The functor F is full iff F_{Mor} is surjective and faithful iff F_{Mor} is injective on (i.e. restricted to) each morphism set $Mor_{\underline{K}}(A,B)$ where $A,B \in Ob_{\underline{K}}$. This does not mean that F_{Mor} (unrestricted) is surjective or injective. Thus a full and faithful functor has F_{Mor} bijective on each morphism set, but possibly not F_{Mor} (unrestricted) bijective, as two different objects may be mapped into the same one by a full and faithful functor (Schubert 1972, p. 25).

The functor F is dense if for each $B \in Ob_{\underline{L}}$ there is an $A \in Ob_{\underline{K}}$ such that AF is isomorphic to B (Herrlich-Strecker 1973, p. 69).

The functor F is an embedding if F_{Mor} (unrestricted) is injective. A faithful functor is an embedding only if it also is bijective on objects (Herrlich-Strecker 1973, p. 69).

If F_{Ob} is an identity, the functor is called strict.

A forgetful functor U: K → Set maps every object in K to a (set) mapping. There might be other forgetful functors (into some other category than Set) but those are of little interest here.

3.1 Algebras

Given an algebraic theory $\underline{T}$, an algebra of type $\underline{T}$ or a $\underline{T}$-$\underline{\text{algebra}}$ is a functor $A: \underline{T} \to \underline{\text{Set}}$ such that:

1) $wA_{Ob} = (sA_{Ob})^w$, i.e. if $w = s_1 \cdots s_n$,
then $(sA_{Ob})^w = s_1A_{Ob} \times \cdots \times s_nA_{Ob}$ (canonical product)

2) if $p_i^w: w \to s_i$ in $\underline{T}$, where $w = s_1 \cdots s_n$,
then $p_i^wA_{Mor}: (sA_{Ob})^w \to s_iA_{Ob}$ (product-preserving).

Usually the set s_iA_{Ob} is called A_{s_i} (the $\underline{\text{carrier}}$ of sort s_i). The canonical condition is needed because there are many products for a family of objects $\{s_i\}_{i \in I}$, where $s_i \in S$, in the string category. It ensures that a product $w = s_1 \cdots s_n$ is mapped into a product $A^w = A_{s_1} \times \cdots \times A_{s_n}$ and not to the (permutation) $A_{s_2} \times A_{s_1} \times A_{s_3} \times \cdots \times A_{s_n}$ which contains the same carriers but in another order. In this work all algebras are supposed to be canonical.

If $w = \lambda$ (the empty product) then we have $\lambda A_{Ob} = (sA_{Ob})^\lambda$ and the terminal object λ in $\underline{T}$ is mapped into the one element set $\{()\}$. The projection p_λ^λ is mapped into the empty mapping $(sA_{Ob})^\lambda \to \lambda A_{Ob}$, that is $\{()\} \to \{()\}$.

The one element set $\{()\}$ (in fact $\underline{\text{any}}$ one element set) is $\underline{\text{terminal object}}$ in $\underline{\text{Set}}$ because there is a unique mapping from any set B to the one element set, i.e. the surjection $s: B \to \{()\}$ mapping every element of B into the single element of $\{()\}$.

The sets A_s (= sA_{Ob}) in the set family $A = (A_s: s \in S)$ are the <u>carriers</u> of the algebra A and the mappings $\sigma_{w,s}A$, usually denoted $\sigma_A: A^w \to A_s$, are the <u>operations</u> of the algebra A (remember that σ is an operator <u>symbol</u>, $\sigma_{w,s}$ is an operat<u>or</u> and $\sigma_{w,s}A: A^w \to A_s$ is an operat<u>ion</u>).

3.2 Hom-functors

The source of a functor can be a product of categories (the product is a category too). Especially interesting is a bifunctor (from a product of two categories) which is called the hom-functor

$$\mathrm{Hom}: \underline{K}^{op} \times \underline{K} \to \underline{Set}$$

defined by

$$\mathrm{Hom}_{Ob}: Ob_{\underline{K}}{}^{op} \times Ob_{\underline{K}} \to Mor_{\underline{K}}: (A,B) \mapsto Mor_{\underline{K}}(A,B)$$

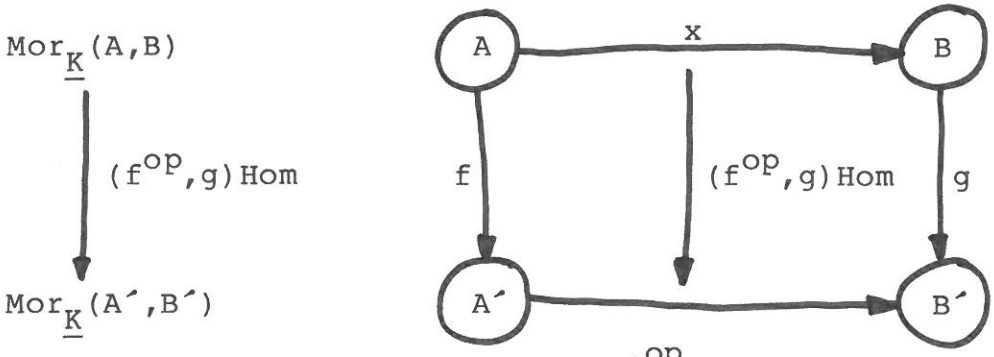
and

$$\mathrm{Hom}_{Mor}: Mor_{\underline{K}}{}^{op} \times Mor_{\underline{K}} \to Mor_{\underline{K}}: (f^{op},g) \mapsto (f^{op},g)\mathrm{Hom}$$

where $f: A \to A'$ and $g: B \to B'$ gives

$$(f^{op},g)\mathrm{Hom}: Mor_{\underline{K}}(A,B) \to Mor_{\underline{K}}(A',B'): x \mapsto f^{op}xg$$

which gives the commuting diagram



$$f^{op}xg = x(f^{op},g)\mathrm{Hom}$$

The hom-functor is defined only for locally small categories (the morphisms from an object A to an object B form a set for any $A,B \in Ob_{\underline{K}}$), but as a rule all interesting categories are locally small in this report.

If either argument is fixed, the hom-functor is reduced to two "usual" functors:

the covariant hom-functor (usually denoted Hom^A) for $\underline{K}$ with respect to an object A

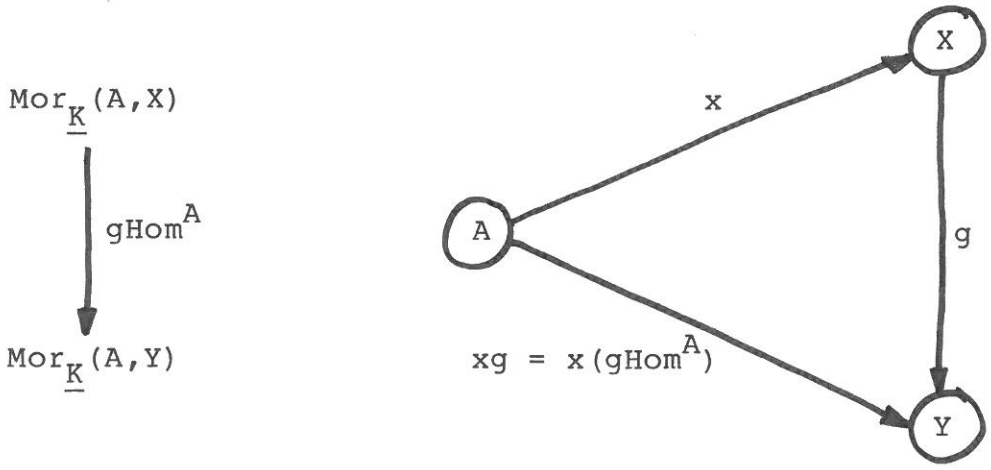
$$(A,\_)Hom: \underline{K} \rightarrow \underline{Set}$$

and the contravariant hom-functor (usually denoted Hom_B) for $\underline{K}$ with respect to an object B

$$(\_,B)Hom: \underline{K}^{op} \rightarrow \underline{Set}.$$

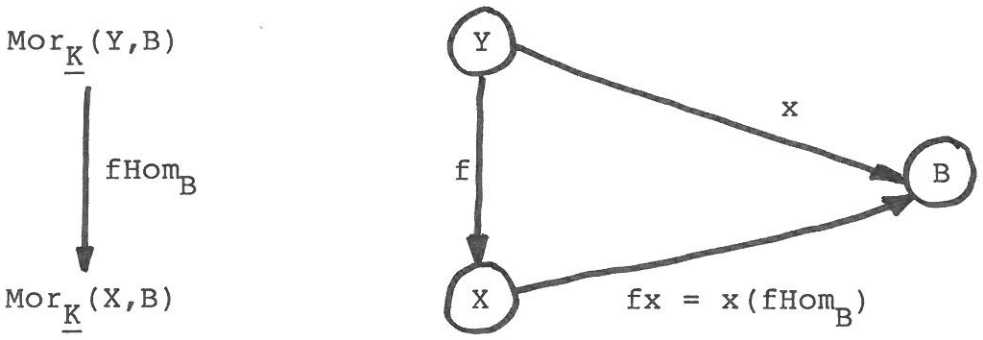
Note that Hom_B is defined as a <u>covariant</u> functor from $\underline{K}^{op}$ here. It is a contravariant functor from $\underline{K}$ (hence its name) but for a contravariant definiton of Hom_B the composition must be redefined (turned around). In this report, however, only the covariant functor definition will be used.

The <u>covariant hom-functor</u> $Hom^A: \underline{K} \rightarrow \underline{Set}$ assigns to each object X in $\underline{K}$ the morphism set from A to X, i.e. $XHom^A = Mor_{\underline{K}}(A,X)$ and to each morphism $g: X \rightarrow Y$ a mapping $gHom^A: Mor_{\underline{K}}(A,X) \rightarrow Mor_{\underline{K}}(A,Y): x \mapsto xg$ such that the following diagram commutes:



The contravariant hom-functor is not used here, but as an example of a contravariant functor (used by the ADJ group to define an algebra in (GTWW 1975)) the definition is given in the contravariant form.

A <u>contravariant hom-functor</u> Hom_B: $\underline{K} \rightarrow \underline{Set}$ assigns to each object X in $\underline{K}$ the morphism set from X to B, i.e. $XHom_B = Mor_{\underline{K}}(X,B)$ and to each morphism f: $Y \rightarrow X$ a mapping $fHom_B$: $Mor_{\underline{K}}(Y,B) \rightarrow Mor_{\underline{K}}(X,B)$: $x \mapsto fx$ such that the following diagram commutes:



Because the functor is defined in its contravariant form the definition of a (covariant) functor must be redefined for Hom_B: $\underline{K} \rightarrow \underline{Set}$ (but <u>not</u> for Hom_B: $\underline{K}^{op} \rightarrow \underline{Set}$!!):

2b) (for a contravariant functor $F: \underline{K} \to \underline{L}$)

if $f: A \to B$ and $g: B \to C$ in $\underline{K}$

then $(fg)F_{Mor} = (gF_{Mor})(fF_{Mor}): CF_{Ob} \to AF_{Ob}$

in the category $\underline{L}$, that is, the contravariant functor preserves but reverses composition.

3.3 Free algebras

For a given algebraic theory $\underline{T}$ there are usually many $\underline{T}$-algebras (even canonical). A certain group is especially interesting, the free algebras, because (by definition) there always is a unique morphism from the free algebra to any other algebra (with the same set of generators).

Such an unique morphism is a model of the semantics in this approach and thus it is very important to know that it is always possible to find a unique behaviour for any (syntactic) word in the free algebra.

Given an algebraic theory $\underline{T}$ it would be nice to have a method of constructing a free algebra, i.e. a functor $F: \underline{T} \rightarrow \underline{Set}$ which is product-preserving, canonical and has an unique morphism to any other algebra of the same "type".

In (GTWW 1975, p. 35) the contravariant hom-functor was shown to be a free algebra of a homogeneous algebraic theory (with one single sort). Because the algebraic theories in this report are dual to those of (GTWW 1975) the covariant hom-functor is a strong candidate.

Consider the hom-functor $Hom^{v}: \underline{T} \rightarrow \underline{Set}$ with respect to a string (an object) v in the algebraic theory $\underline{T}$. From the

definition of a (covariant) hom-functor (see 3.2) we have the object mapping

$$wHom^{V} = Mor_{\underline{T}}(v,w).$$

Because w is a product in $\underline{T}$, there is a unique morphism $h: v \to w$ for each w-tuple of morphisms $f: v \to s_j$, where $w=s_1 \cdots s_n$ and $1<=j<=n$. Thus we have

$$Mor_{\underline{T}}(v,w) = (Mor_{\underline{T}}(v,s_j))^{W}$$

for $h = (f)^{W}$ and

$$wHom^{V} = Mor_{\underline{T}}(v,w) = (Mor_{\underline{T}}(v,s_j))^{W} = (s_jHom^{V})^{W}$$

because $s_jHom^{V} = Mor_{\underline{T}}(v,s_j)$. Consequently Hom^{V} is <u>canonical</u> (see 3.1).

The morphism mapping of Hom^{V} maps each morphism $t: w \to u \in Mor_{\underline{T}}(w,u)$ into

$$tHom^{V}: Mor_{\underline{T}}(v,w) \to Mor_{\underline{T}}(v,u): x \mapsto xt$$

which for $t = p_j^{w}: w \to s_j$ and $x = (p_i^{v})^{w}$ gives

$$p_j^{w}Hom^{V}: Mor_{\underline{T}}(v,w) \to Mor_{\underline{T}}(v,s_j).$$

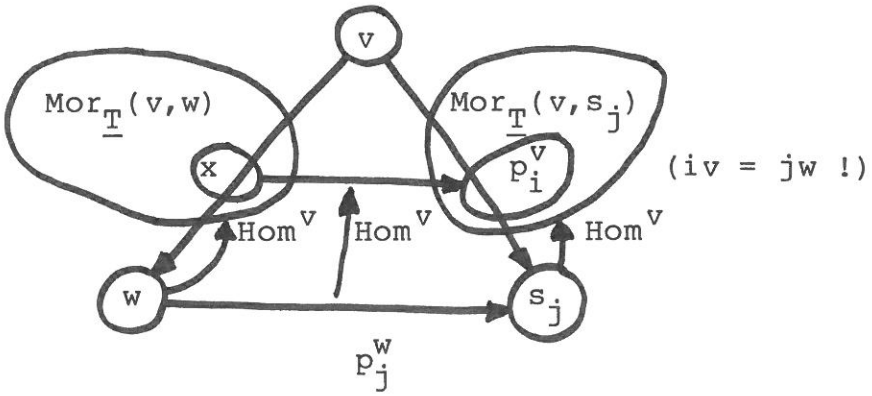
Considering the object mapping above we can write this

$$p_j^{w}Hom^{V}: (Mor_{\underline{T}}(v,s_j))^{W} \to Mor_{\underline{T}}(v,s_j)$$

or

$$p_j^{w}Hom^{V}: (s_iHom^{V})^{W} \to s_jHom^{V}$$

where $w = s_1 \cdots s_n$ and $1<=i<=n$. Thus Hom^{V} is <u>product-preserving</u> (see 3.1) and consequently a $\underline{T}$-algebra.



The objects of the algebraic theory $\underline{T}$ are mapped by Hom^v into the morphism sets of $\underline{T}$ from v to some other string w. This is the reason why they were investigated so thoroughly in Section 2.5.

A carrier of Hom^v is the set of morphisms from v to some single sort s (carrier of sort s). Because of the lambda projections p^v_λ, by composition all the (terminal) words of sort s belong to this set (see 2.5). If s=<program> then all well-formed (syntactically correct) programs of the language are in the carrier $T_{v,s} = Mor_{\underline{T}}(v,s)$.

There might be other morphisms too, which are not (terminal) words in the language because they contain projections p^v_i (sometimes the projections are called variables and denoted x_{i,s_i} (GTWW 1977), but they are metavariables - not variables in the programming language to be modelled). These morphisms are derived operators, in fact all derived operators which can be generated by projections from v. Of course these projections can be renamed using some (meta) variable set X_v.

The morphism sets from v to proper strings w consist of w-tuples of morphisms from v to single sorts. An example might best illustrate the nature of the free algebra.

Example: The carriers of a free algebra Hom^{v}

Let the signature have sort set $S = \{a,b\}$ and operator sets

$$\Sigma_{\lambda,a} = \{A\}$$
$$\Sigma_{\lambda,b} = \{B\}$$
$$\Sigma_{ab,a} = \{(\lambda\_+\_\lambda)\}$$
$$\Sigma_{b,b} = \{(-\_\lambda)\}$$
$$\Sigma_{w,u} = \emptyset \text{ for all other } w,u \in S^{*}.$$

It is considered that parenthesis ( ) and underline _ do not belong to the alphabet of the language to be modelled.

If v = abab, then (according to the definition of a theory T) the following morphisms are in the carriers of Hom^{v} (and their products):

p_1^v, p_3^v	$\in$	$T_{v,a}$	(projections)
$A\ (= p_\lambda^v A)$	$\in$	$T_{v,a}$	(constant)
p_2^v, p_4^v	$\in$	$T_{v.b}$	(projections)
$B\ (= p_\lambda^v B)$	$\in$	$T_{v,b}$	(constant)

By the definition of a product there must be target tuples in $T_{v,ab}$ for each pair of morphisms in $T_{v,a}$ and $T_{v,b}$:

$$(p_1^v, p_2^v), (p_1^v, B), \cdots, (A, p_2^v), (A, p_4^v), (A,B), \cdots \in T_{v,ab}.$$

Composition of these tuples with the operator in $T_{ab,a}$ gives $p_1^v+p_2^v, p_1^v+B, \cdots, A+p_2^v, A+p_4^v, A+B, \cdots \in T_{v,a}$.

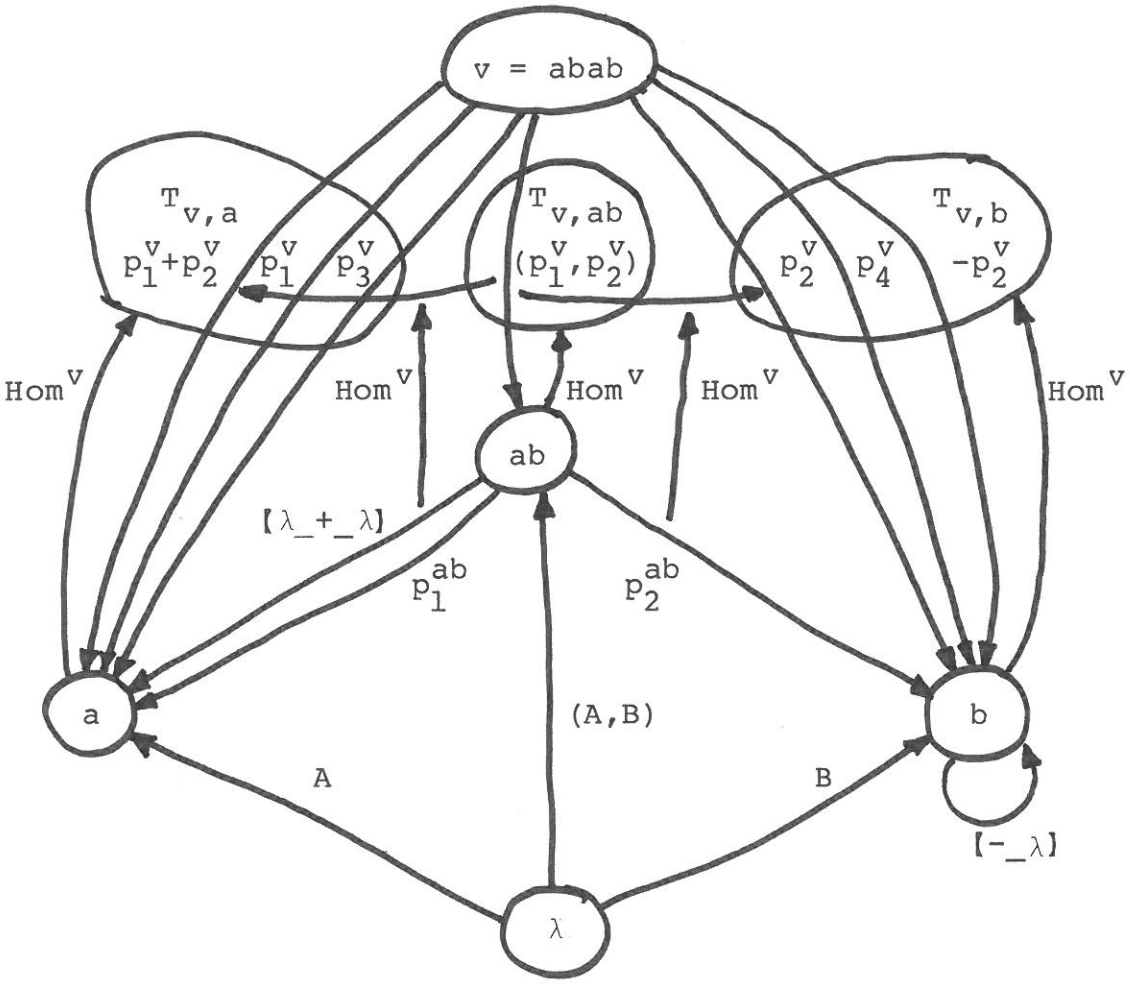
On the next level there must be target tuples

$$(p_1^v+p_2^v, p_2^v), \cdots, (A+B, B), \cdots \in T_{v,ab}$$

and by composition

$$p_1^v+p_2^v+p_2^v, \cdots, A+B+B, \cdots \in T_{v,a}$$

and so on infinitely. Thus the carriers contain "expressions" in + and - with constants A and B together with projections p_1^v, p_2^v, p_3^v and p_4^v. Some of the morphisms are shown in the following figure:



* * *

Once the algebraic theory $\underline{T}$ is constructed, it is easy to obtain the free algebras Hom^v by taking all morphism sets $Mor_{\underline{T}}(v,w)$, with the string v as source, as objects and define the obvious mappings between them so that the diagrams of the type above all commute.

The free algebra is useful when programs (or constructs of other sorts) are studied on an abstract level, that is, when it is irrelevant exactly which constructs of lower order are used.

For example, if the structure of identifiers is considered to be uninteresting, a free algebra with a suitable number of projections p_i^v, where iv = <identifier> for some 1<=i<=n) can be used to model the syntax of language A (see Appendix).

To get rid of the infinite set of identifiers generated by the operators

$$[\lambda\_\lambda]_{\underline{le},\underline{id}} \qquad [\lambda\_\lambda\_\lambda]_{\underline{id}\,\underline{le},\underline{id}} \qquad [\lambda\_\lambda\_\lambda]_{\underline{id}\,d,\underline{id}}$$

where $\underline{le}$ = <letter>, $\underline{id}$ = <identifier> and d = <digit>, these operators must be deleted from the signature generating the algebraic theory $\underline{T}$.

Thus the free algebra of a theory $\underline{T}' = \underline{StS}_{\Sigma'}$, where Σ' is the same as Σ except for the deleted operators, would

give a model with a finite set of identifiers. It would be convenient to rename the projections p_i^v with names which are identifiers in the language A - if it is unnecessary to stress that they really are metavariables.

Using a free algebra it is possible to get "words" in the carriers even if there are no constants at all in the signature and thus all morphism sets $Mor_{\underline{T}}(\lambda, w)$ are empty. In that case the "words" are not in the language because they consist of projections and operators (mostly derived operators) but using the renaming trick with X_v, a set of terminal words in the language, it is possible to get natural constructs anyway.

3.4 Initial algebras

A special case of free algebras, which is quite interesting, is the initial algebra (with $v = \lambda$). There are no projections at all in the carriers of the initial algebra (except for $p^{\lambda}_{\lambda} = id_{\lambda}$ in $T_{\lambda,\lambda}$). Thus all words in the carriers are terminal (well-formed expressions or terms) and the initial algebra is a nice model of the syntax. It is also called the absolutely free algebra, the anarchic algebra, the word algebra and the Herbrand universe (the carriers $T_{\lambda,s}$).

If there are no constants in the signature generating the algebraic theory $\underline{T}$, then the carriers of its initial algebra are all empty. In fact, for every primitive sort s there must be at least one constant (operator in $\Sigma_{\lambda,s}$) in order to have no empty carriers.

Example: The initial algebra Hom

Using the same signature as in section 3.3 we have the following morphisms in the carriers of the initial algebra Hom^{λ}:

$$A \in T_a \ (= Mor_{\underline{T}}(\lambda,a)) \qquad \text{(constant)}$$
$$B \in T_b \ (= Mor_{\underline{T}}(\lambda,b)) \qquad \text{(constant)}$$

By target tupling we also have

$$(A,B) \quad \in T_{ab}$$

and by composition with the operator in $T_{ab,a}$

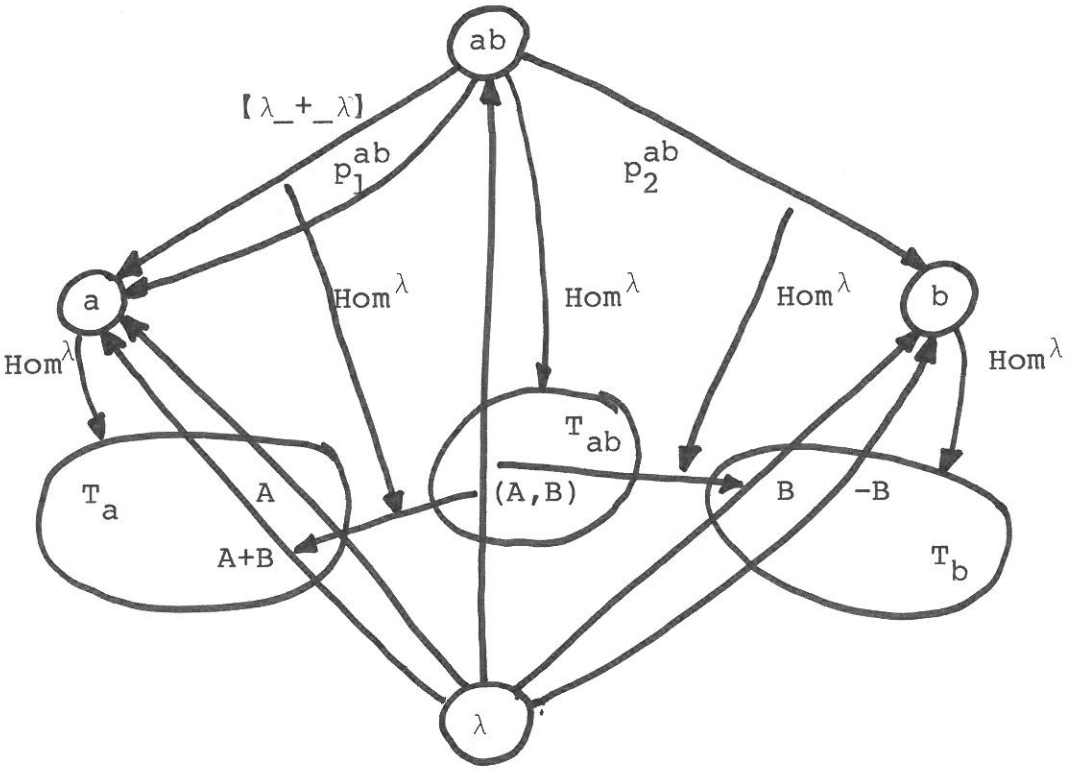
$$A+B \quad \in T_a.$$

Thus the carriers will be

$T_a = \{A, A+B, A+-B, A+A+B, \cdots\}$

$T_b = \{B, -B, --B, \cdots\}$

$T_{ab} = \{(A,B), (A,-B), (A+B,B), (A+B,-B), \cdots\}$



Note how v=abab in 3.3 is replaced by λ above. The Hom^v-mappings are the same in both examples.

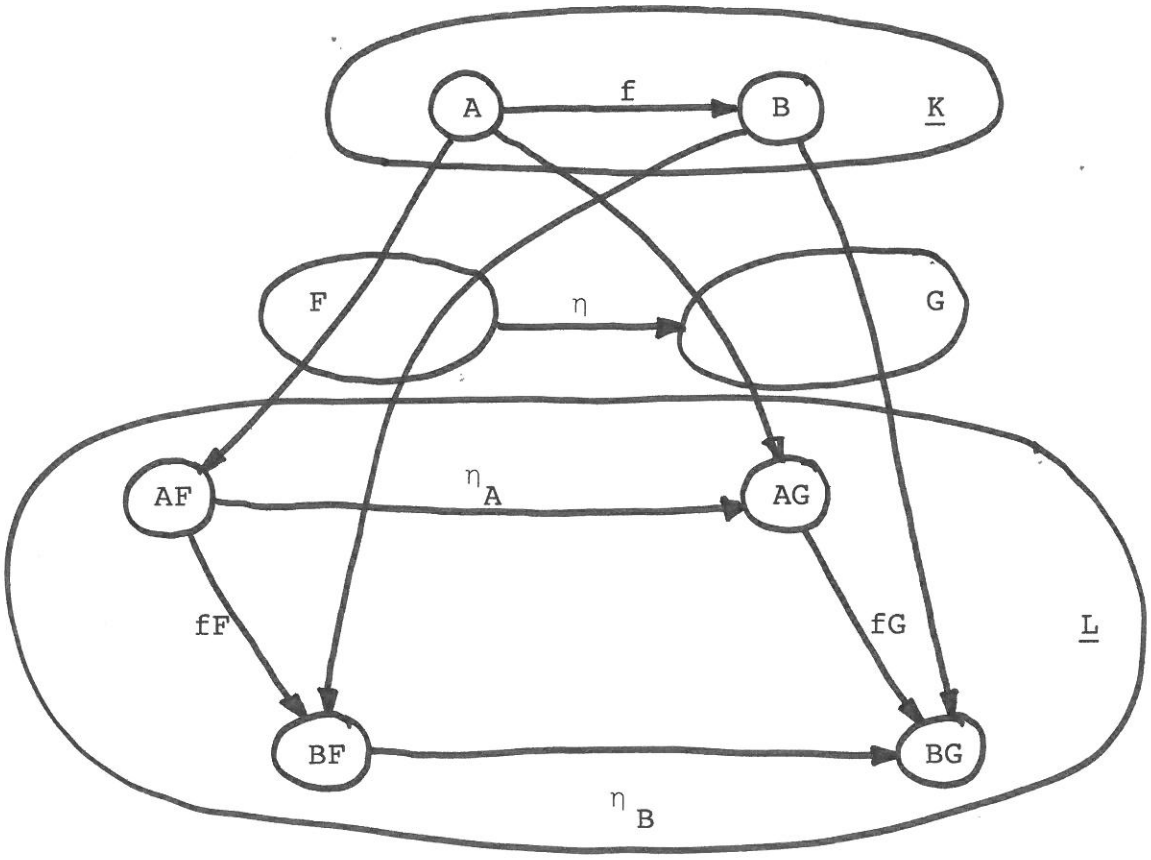
* * *

The Herbrand universe for the case with no constants is not Hom^{λ} but a free algebra Hom^{v}, where v is chosen so that there is one projection p_i^v to every primitive sort s. This projection corresponds to the arbitrary constant in the standard construction of a Herbrand universe.

4. NATURAL TRANSFORMATIONS DEFINE THE SEMANTICS

In this model both syntax and behaviour are algebras, i.e. functors. Thus it is very important to have a concept relating functors to each other. In category theory there is a fundamental construct which also can be used to construct categories of functors, among others categories of algebras.

A <u>natural transformation</u> η: F $\rightarrow$G (F: $\underline{K} \rightarrow \underline{L}$ and G: $\underline{K} \rightarrow \underline{L}$ are functors) is a mapping $\eta : Ob_{\underline{K}} \rightarrow Mor_{\underline{L}}$: A $\mapsto \eta_A$ such that the following diagram commutes for all f: A $\rightarrow$B $\in Mor_{\underline{K}}(A,B)$:



Thus a natural transformation is a family of $\underline{L}$-morphisms $(\eta_A: AF \to AG$ for all $A \in Ob_{\underline{K}})$ such that $\eta_A \circ fG = fF \circ \eta_B$. The morphisms η_A are called the components of η. A natural transformation is sometimes called a functorial (or functor) morphism.

If the functors $F,G: \underline{T} \to \underline{Set}$ are algebras then a natural transformation $\eta: F \to G$ is a family

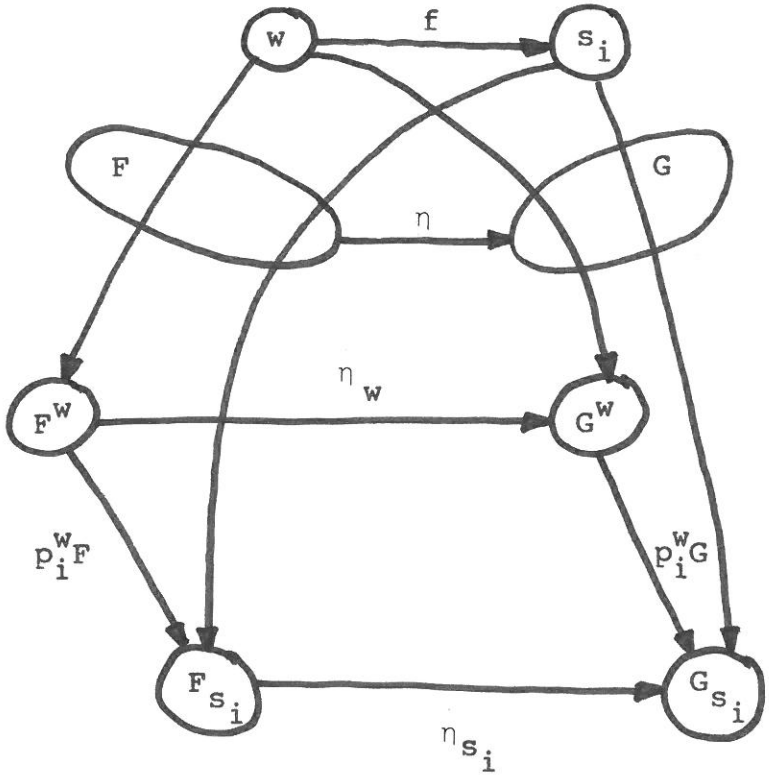
$$(\eta_w: F^w \to G^w \text{ for all } w \in Ob_{\underline{T}}),$$

where F^w is the product $(sF_{Ob})^w$ and G^w is $(sG_{Ob})^w$.

From the definition of a natural transformation we have

$$\eta_w \circ p_i^w G = p_i^w F \circ \eta_{s_i}$$

for $f = p_i^w: w \to s_i$.



The Set-object F^w is a canonical product with projections $p_i^w F: F^w \to F_{s_i}$. Thus for an arbitrary $(f_{s_1},\cdots,f_{s_n}) \in F^w$, where $w=s_1\cdots s_n$, we have $(f_{s_1},\cdots,f_{s_n})p_i^w F = f_{s_i}$ and $f_{s_i}\eta_{s_i} = g_{s_i}$.

The mapping η_w applied to an element of F^w gives

$$(f_{s_1},\cdots,f_{s_n})\eta_w = (g_{s_1},\cdots,g_{s_n}) = (f_{s_1}\eta_{s_1},\cdots,f_{s_n}\eta_{s_n})$$

because the diagram above must commute. Thus for $w=s_1\cdots s_n$ $\eta_w = (\eta_{s_1},\cdots,\eta_{s_n})$. This means that if $\eta_s = \rho_s$ for all $s \in S$ then $\eta = \rho$.

A (vertical) composition of natural transformations $\eta: F \to G$ and $\varepsilon: G \to H$ is defined as the mapping $\eta\varepsilon: Ob_{\underline{K}} \to Mor_{\underline{L}}: A \mapsto (\eta\varepsilon)_A$, where $(\eta\varepsilon)_A: AF \to AH$ is the usual composition of the $\underline{L}$-morphisms $\eta_A: AF \to AG$ and $\varepsilon_A: AG \to AH$. The composition $\eta\varepsilon$ is also a natural transformation.

Because the composition of morphisms η_A, ψ_A and ε_A is associative for all $A \in Ob_{\underline{L}}$ then we have

$$(\eta\psi)\varepsilon = \eta(\psi\varepsilon).$$

The morphism family $(id_{AF}: AF \to AF$ for all $A \in Ob_{\underline{K}})$ defines a natural transformation $id_F: F \to F$ such that for all natural transformations $\eta: F \to G$ and $\psi: G \to F$, we have

$$id_F\,\psi = \psi \text{ and } \eta id_F = \eta.$$

The functors from a category $\underline{K}$ to a category $\underline{L}$ form a new category, the functor category $[\underline{K},\underline{L}]$ (or $\underline{L}^{\underline{K}}$) with all functors $F: \underline{K} \rightarrow \underline{L}$ as objects and natural transformations between the functors as morphisms. But this is true only if $\underline{K}$ is a small category (Pareigis 1970, p. 10). This is usually the case in computer science applications.

If also $\underline{L}$ is a small category (as it usually is here) the functor category $[\underline{K},\underline{L}]$ is small (Schubert 1972, p. 19).

In general a morphism η in $[\underline{K},\underline{L}]$ (a natural transformation) "inherits" the categorial properties that are common to η_A for all $A \in Ob_{\underline{K}}$. The functor category $[\underline{K},\underline{L}]$ often inherits the properties of $\underline{L}$ (Herrlich-Strecker 1973, p. 95).

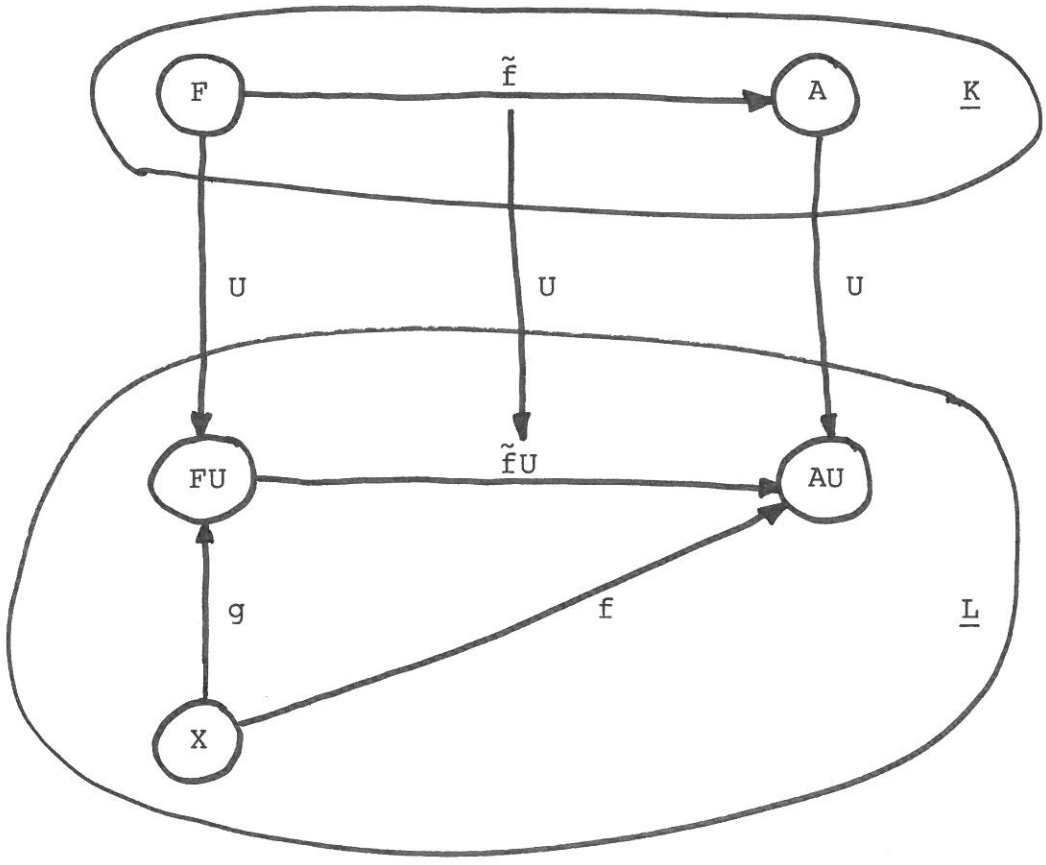
Thus, if the components η_A are monomorphisms in the category $\underline{L}$ (for all $A \in Ob_{\underline{K}}$), then the natural transformation η is a monomorphism in the functor category $[\underline{K},\underline{L}]$.

If a category $\underline{L}$ has a terminal (or initial) object then the functor category $[\underline{K},\underline{L}]$ will have a terminal (initial) object too for any category $\underline{K}$ (Herrlich-Strecker 1973, p. 95).

4.1 Generators and free objects

The theoretical background of free algebras is closely connected to "universal maps". Here only a special kind of universal maps is used, but first the general definition:

A universal map $g: X \to FU$ (a free object F) for X with respect to U, where $F \in Ob_{\underline{K}}$, $X \in Ob_{\underline{L}}$ and the functor $U: \underline{K} \to \underline{L}$, has the property that for each $A \in Ob_{\underline{K}}$ and each $f: X \to AU$ there is an unique $\underline{K}$-morphism $\tilde{f}: F \to AU$ such that the following diagram commutes:



Often the pair (g,F) is called a universal map (Herrlich-Strecker 1973, p. 178), or free over X with respect to U (Goldblatt 1979, p. 441).

The object X is also called generator, F is called the free object generated by X and g is inclusion of generators.

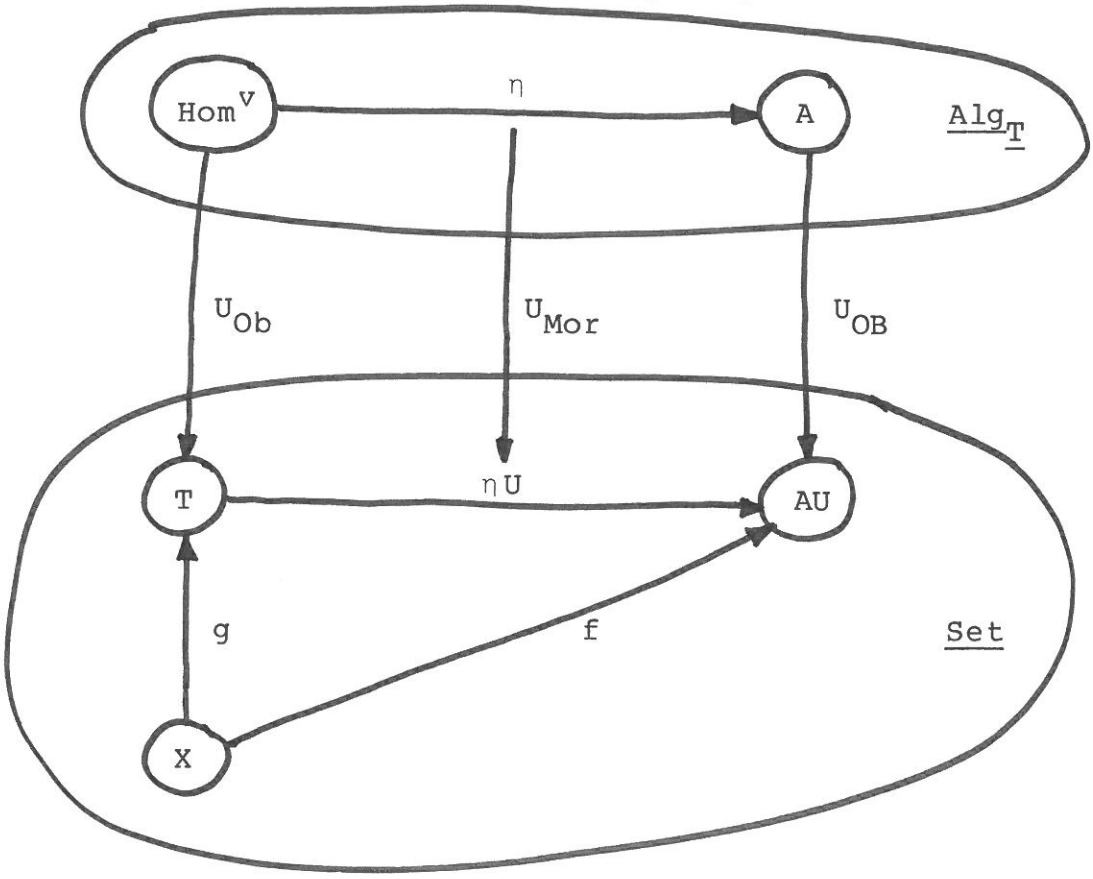
4.2 The category of $\underline{T}$-algebras

The category $\underline{Alg}_{\underline{T}}$ of algebras has $\underline{T}$-algebras (functors) as objects and natural transformations as morphisms. In this category all algebras are of the same type because they are canonical product-preserving functors from $\underline{T}$ to $\underline{Set}$.

To show that Hom^V is a free object in $\underline{Alg}_{\underline{T}}$, i.e. a free algebra, it is necessary to show that for some set X (generators) and some mapping $g: X \rightarrow Hom^V U$, where $U: \underline{Alg}_{\underline{T}} \rightarrow \underline{Set}$ is a forgetful functor, there is an <u>unique</u> $\underline{Alg}_{\underline{T}}$-morphism (a natural transformation)

$$\eta: Hom^V \rightarrow A$$

for each algebra (functor) $A: \underline{T} \rightarrow \underline{Set}$ and each mapping $f: X \rightarrow AU$ such that the following diagram commutes



and T is the set family $T = (T_{v,w}: w \in S^*)$. This set family consists of the carriers $T_{v,s}$ of Hom^v and their products (see 3.3). The same goes for $AU = (A^w: w \in S^*)$.

The generator $X = (X_{v,s})$ has as many elements as there are sorts in the string $v = s_1 \cdots s_n$ and moreover it is divided into subsets $X_{v,s} = \{x_{v,i}: iv = s\}$. Thus the generator set $X = \{x_{v,1}, \cdots, x_{v,n}\}$, where each element (metavariable) $x_{v,i}$ is of sort iv.

Now the inclusion of generators (rather an injection) is a family $g = (g_s: X_{v,s} \to T_{v,s}: x_{v,i} \mapsto p_i^v$ for all $s \in S_v$ and i such that iv=s), where $S_v = \{s_j: 1<=j<=n\}$.

If A is an algebra such that there is a mapping $f = (f_s: X_{v,s} \to A_s: x_{v,i} \mapsto a_{s,i}$ with iv = s) there must be

$$g \circ \eta U = f.$$

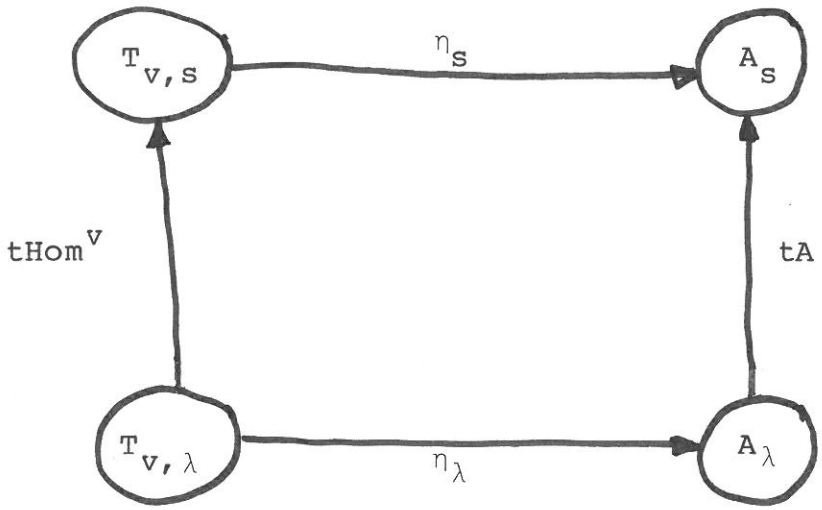
As the components of η are mappings in Set we simply consider the natural transformation η to be a family of morphisms in Set (like Pareigis 1970, p. 9) and thus $\eta = \eta U$ turning the condition above into

$$g_s \circ \eta_s = f_s \text{ for all } s \in S_v.$$

According to the definition of a natural transformation

$$\eta_\lambda \circ tA = tHom^v \circ \eta_s$$

for any morphism $t: \lambda \to s$ and any $s \in S$, i.e. the following diagram commutes



From section 3.1 we know that $A_\lambda = \{()\}$ and from section 2.5 that $T_{v,\lambda} = \{p_\lambda^v\}$. Thus $\eta_\lambda: T_{v,\lambda} \to A_\lambda: p_\lambda^v \mapsto ()$ is an isomorphism.

The morphisms $tA: A_\lambda \to A_s: () \mapsto ()tA$ are injections from a singleton. Thus we can identify the image with the mapping, i.e. denote $()tA$ with tA.

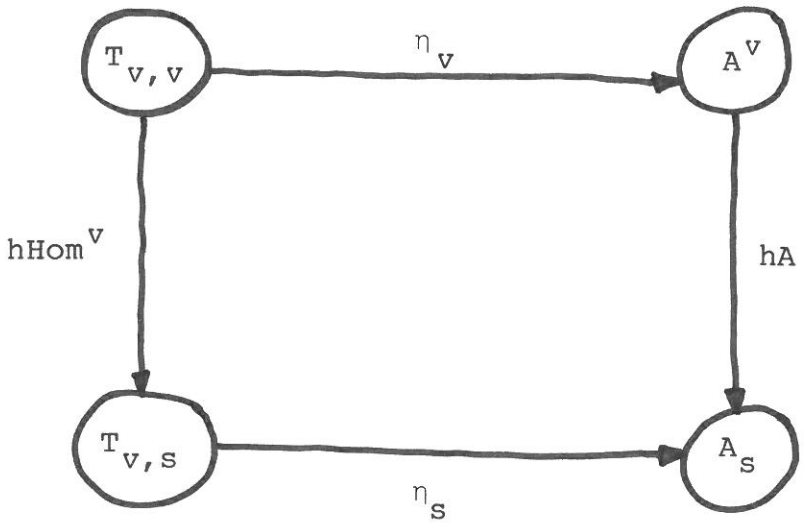
From the definition of Hom^v (section 3.2) we know that $tHom^v: T_{v,\lambda} \to T_{v,s}: p_\lambda^v \mapsto p_\lambda^v \circ t$ and because $T_{v,\lambda}$ is a singleton too, we must have

$$(p_\lambda^v \circ t)\,\eta_s = tA \quad (= ()tA).$$

The natural transformation η must also satisfy

$$\eta_v \circ hA = hHom^v \circ \eta_s$$

for any morphism h: $v \to s$ and any $s \in S$, i.e. the following diagram must commute



Above the mapping η_s was determined for morphisms $p_\lambda^v \circ t_{\lambda,s} \in T_{v,s}$, but there might be other elements in $T_{v,s}$ too. According to section 2.5 it is possible that $T_{v,s}$ contains <u>projections</u> p_i^v (if $iv = s$ for some i) and/or <u>operators</u> $\sigma_{v,s}$.

If there is a projection p_i^v such that $iv = s$, then for this i we have

$$x_{v,i} g_s = p_i^v$$

and in order to have $g_s \circ \eta_s = f_s$ we must have

$$(p_i^v)\eta_s = x_{v,i} f_s$$

where $x_{v,i} f_s \in A_s$.

If there is an operator $\sigma_{v,s} \in T_{v,s}$ then commutativity gives (for $id_v \in T_{v,v}$)

$$(id_v)\sigma_{v,s}\mathrm{Hom}^v \circ \eta_s = (id_v)\eta_v \circ \sigma_{v,s}A$$

where

$$(id_v)\sigma_{v,s}\mathrm{Hom}^v = id_v \circ \sigma_{v,s} = \sigma_{v,s}$$
$$(id_v)\eta_v = ((p_i^v)\eta_{s_i})^v = (x_{v,i}f_s)^v,$$

and thus

$$(\sigma_{v,s})\eta_s = (x_{v,i}f_s)^v\sigma_{v,s}A.$$

Thus η_s is uniquely determined for all morphisms in $T_{v,s}$ for all $s \in S$ and for a given set X and a given mapping $g: X \to \mathrm{Hom}^v U$ there is a unique η for each algebra A and mapping $f: X \to AU$.

Example: The semantics of a free algebra

If d = <digit> and n = <natural number> are sorts in a signature $S\Sigma$ then for v = nd we have a free algebra $\mathrm{Hom}^v: \underline{T} \to \underline{\mathrm{Set}}$ (where $\underline{T} = \underline{\mathrm{St}}S$), which is a free object in $\underline{\mathrm{Alg}}_{\underline{T}}$ for X = {X1,X2} with respect to $U: \underline{\mathrm{Alg}}_{\underline{T}} \to \underline{\mathrm{Set}}$.

The universal map $g: X \to \mathrm{Hom}^v U$ is defined by $X1g = p_1^v \in T_{v,n}$ and $X2g = p_2^v \in T_{v,d}$.

If $A: \underline{T} \to \underline{\mathrm{Set}}$ is another algebra (the behaviour) with A_d = Dig, the set of digits 0,···,9, and A_n = Nat, the set of natural numbers, then for a mapping $f: X \to AU$ there is a unique natural transformation $\eta: \mathrm{Hom}^v \to A$. Here f is an

assignment of values to the (meta)variables and η defines the "semantics" of Hom^V.

Thus if X1f = 37 and X2f = 5 then we must have

$$(p_1^V)\eta_n = X1f = 37$$
$$(p_2^V)\eta_d = X2f = 5$$

and

$$(\underline{cat})\eta_n = (X1f, X2f)\underline{cat}A = 375 \in Nat$$

where

$$\underline{cat}_{V,n} = [\lambda p_1^V \lambda p_2^V \lambda]_{V,n}.$$

Of course, for all $t_{\lambda,n} \in T_{\lambda,n}$ the elements $p_\lambda^V \circ t_{\lambda,n} \in T_{V,n}$ are mapped by η_n into Nat, especially we have

$$(p_\lambda^V \circ 37_{\lambda,n})\eta_n = 37.$$

Thus η_n is mapping both p_1^V and $p_\lambda^V 37_{\lambda,n}$ into the same element of Nat if X1g = p_1^V and X1f = 37.

Because this is true for all maps f: X $\rightarrow$AU, we know that the semantics η is "correct" (intuitively). But notice that the "semantics" is <u>different</u> for different f-mappings. This is not very satisfactory, but fortunately there is a free algebra with only one semantics for each behaviour A, the initial algebra Hom^λ.

* * *

From the reasoning above it is clear that the initial algebra Hom^{λ} always will have a unique natural transformation η: $\mathrm{Hom}^{\lambda} \rightarrow A$ (for any algebra A: $\underline{T} \rightarrow \underline{\mathrm{Set}}$). The generator X will be the empty set $\emptyset$ in this case and the universal map g: $\emptyset \rightarrow \mathrm{Hom}^{\lambda}U$ is the empty mapping.

The assignment of values f: $\emptyset \rightarrow AU$ is an empty mapping too and the components of the natural transformation are

$$\eta_s : T_{\lambda,s} \rightarrow A_s .$$

The only elements in $T_{\lambda,s}$ are constants and η_s is completely determined by the condition

$$(t_{\lambda,s})\eta_s = t_{\lambda,s}{}^{A} \quad (= ()\ t_{\lambda,s}{}^{A}) .$$

Thus Hom^{λ} is an initial object in $\underline{\mathrm{Alg}}_{\underline{T}}$. The unique natural transformation η: $\mathrm{Hom}^{\lambda} \rightarrow A$ is the semantics of the language defined by the signature S . Of course the semantics is completely defined by the behaviour (the algebra) A. Some prefer to call A the semantics.

5. CONCLUSIONS

The main aim of this work is to make the theoretical background clear and develop notations which are suitable for this application. The key concept in this work is the notation of an operator (section 2.3). Compared with the corresponding notation in (Husberg 1980a) it has been extended considerably, but unfortunately there are still certain problems.

Multiple occurrences in derived operators are rather clear (section 2.6), but the notation of an (derived) operator with "internal variables" is still not satisfactory. Another problem is the "splitting of sorts" which seem to be necessary in modelling translators (Husberg 1980b).

This report is, however, already too long and these remaining notational problems are left to another work.

This presentation contains no formal proofs because it is mainly written for practical computer people with an interest in theoretical computer science, but without expert knowledge in that area. I hope, however, that some mathematician would take the time to write down the proofs in order to have a more solid mathematical foundation.

Possible directions for future research in CHAMs are:
1) theoretical background, i.e. problems described above and introduction of more category theory concepts (like adjunction).
2) ways of defining the semantics, which in one way is an independent problem. In (Husberg 1980a and 1980b) a "structural" semantics is described, which uses graphs (a graph is the basis of a category).

It seems necessary to move in both directions because they interact. Some of the theoretical problems have their origin in attempts to construct a practical model of semantics.

6. ACKNOWLEDGEMENTS

I would like to thank the Danish Government for a scholarship which made it possible to work at Aarhus University for four months during the winter 1982-83. The Computer Science Department was an excellent environment for this work and I am especially grateful to Peter Mosses and Frank Oles for interesting discussions. Luc Bougé and Neil Jones also helped me with many good comments and questions.

7. LITERATURE

(Cohn 1965) Cohn, P.M.: Universal Algebra, Harper and Row, New York, 1965, 333 p.

(Gallier 1978) Gallier, J. H.: Recursion schemes and generalized interpretations, in "Automata, Languages and Programming", Springer LNCS 71, 1978, pp. 256-269.

(Gallier 1981) Gallier, J.H.: Recursion-closed algebraic theories, JCSS 23 (1981), p. 69-105.

(Ginsburg 1966) Ginsburg, S.: The Mathematical Theory of Context Free Languages, McGraw-Hill, 1966, 232 p.

(Ginsburg 1975) Ginsburg, S.: Algebraic and Automata-Theoretic Properties of Formal Languages, North-Holland, 1975.

(Goguen 1975) Goguen, J.A.: Correctness and Equivalence of Data Types, Int. Symp. Mathematical Systems Theory, Udine 1975, Springer Lecture Notes in Economics and Math. Systems No. 131, 1976, pp. 352-358.

(Goguen 1978) Goguen, J.A.: Abstract Errors for Abstract Data Types, Formal Descriptions of Programming Concepts, E.J. Neuhold (ed.), North-Holland, 1978, pp. 491-522.

(Goldblatt 1979) Goldblatt, R.: Topoi, the categorial analysis of logic, Studies in logic and the foundations of mathematics 98, North-Holland, 1979, 486 p.

(GTWW 1973) Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: A junction between computer science and category theory, IBM Research Reports RC-4526 and RC-5908, Yorktown Heights, New York, 1973 and 1976.

(GTWW 1975) Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: An Introduction to Categories, Algebraic Theories and Algebras, IBM Report RC-5369, Yorktown Heights, New York 1975.

(GTWW 1977) Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial Algebra Semantics and Continous Algebras, JACM, 1977, No. 1, pp. 68-95.

(Herrlich-Strecker 1973) Herrlich, H., Strecker, G.E.: Category Theory, Allyn and Bacon, Boston, 1973, 400 p.

(Husberg 1980a) Husberg, N.: Algebraic Description of High Level Languages Using Category Theory, Helsinki University of Technology, Computing Centre, Research Report No. 16, 1980, 63 p.

(Husberg 1980b) Husberg, N.: Exact description of computing systems, Licentiate thesis, Helsinki University of Technology, Digital Systems Laboratory, 1980, 95 p. (In Swedish)

(Letichevski 1968) Letichevski, A.A.: Syntax and semantics of formal languages, Kibernetika, 1968, No. 4, pp. 1-9. (In Russian, English translation in Cybernetics, Plenum Publishing corp., New York)

(Manes 1976) Manes, E. G.: Algebraic Theories, Springer-verlag, 1976, 356 p.

(Pareigis 1970) Pareigis, B.: Categories and Functors, Academic Press, 1970, 268 p.

(Rus 1972) Rus, T.: S-algebra of formal languages, Bull. Math. de la R. S. Roumanie, 1971 (1972), No. 2, pp. 227-235.

(Schubert 1972) Schubert, H.: Categories, Springer-Verlag, 1972, 385 p.

(Wagner 1981) Wagner, E.: Lecture Notes on the Algebraic Specification of Data Types, IBM Research Report RC-9203, Yorktown Heights, New York, 21 p.

(WTW 1978) Wagner, E., Thatcher, J.W., Wright, J.B.: Programming Languages as Mathematical Objects, Mathematical Foundations of Computer Science, Zakopane 1978, Springer Lecture Notes in Computer Science 64, 1978, p. 84-101.

(WWT 1979) Wagner, E., Wright, J.B., Thatcher J.W.: Many-Sorted and Ordered Algebraic Theories, IBM Research Report RC 7595, Yorktown Heights, N.Y., 1979, 15 p.

APPENDIX

Language A (syntax)

<letter>::=A|B| ··· |Z|a|b| ··· |z

<digit>::=1|2|3|4|5|6|7|8|9|0

<digit string>::=<digit>|<digit string><digit>

<integer>::=<digit string>|$\{^{+}_{-}\}$<digit string>

<identifier>::=<letter>|<identifier>$\{^{\text{<letter>}}_{\text{<digit>}}\}$

<variable>::=<identifier>

<label>::=<identifier>

<expression>::=<integer>|<variable>|<expression>$\{^{+}_{-}\}$<variable>

<assignment>::=<variable>=<expression>

<goto>::=goto <label>

<condition>::=<variable>$\left\{\begin{matrix}\neq\\=\\<\end{matrix}\right\}$<expression>|<condition>$\{^{\text{or}}_{\text{and}}\}$<condition>

<conditional>::=if <condition> then <statement>[else <statement>]

<loop>::=for <variable>=<expression>[step <expression>]
until <expression> do <statement>

<statement>::=<assignment>|<goto>|<conditional>|<loop>|
(<sequence>)|<labelled statement>

<sequence>::=<statement>|<sequence>;<statement>

<labelled statement>::=<label>.<statement>

<program>::=program <identifier>;<sequence> end

INDEX

(Underlined pages contain definition)