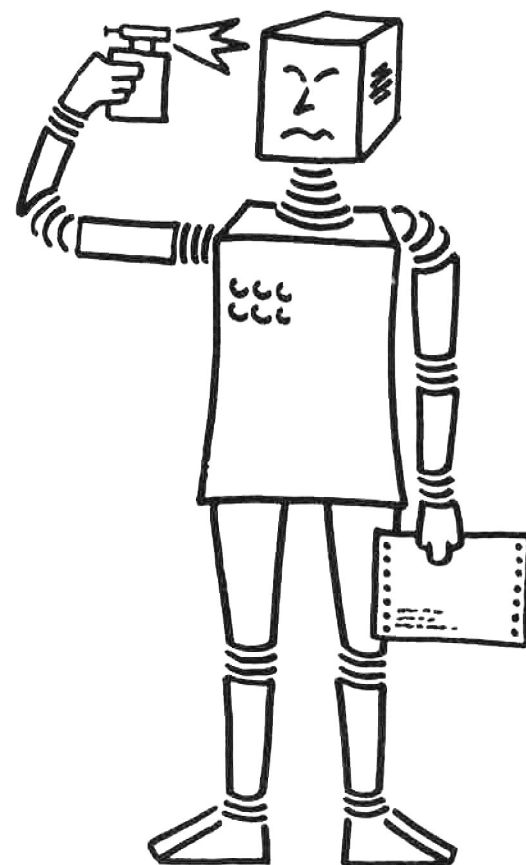
SEKI-PROJEKT

SEKI MEMO

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



A Treatment of Collection Data as Constructor Algebras

Harold Boley

MEMO SEKI-84-06 October 1984

A TREATMENT OF COLLECTION DATA AS CONSTRUCTOR ALGEBRAS

Harold Boley, Universitaet Kaiserslautern
Fachbereich Informatik, Postfach 3049, D-6750 Kaiserslautern

Abstract

This paper gives algebraic definitions of various types of nested variable-length "collections" of elements, usable as data structures. First of all, however, the paper introduces constructor algebras through an integer sequences data type. It then begins with the fundamental CONS algebra of N-tuples and a variant with "negative" elements, complementing it and subsequent homogeneous algebras by heterogeneous ones. For such list algebras, the paper postulates axioms embodying the three "basic properties" [Commutativity, Idempotence, Associativity], and uses these to define the remaining seven "basic collections" [strings, communes, acommunes, bags, abags, sets, heaps]. Proceeding to "non-basic collections", it then introduces "adsorption" properties ["complementary" to absorption], which are characteristic for graphs, and postulates them as axioms in algebras of ordinary graphs and of directed recursive labelnode hypergraphs [DRLHs]. Finally, it defines the property of "Similpotence" ["weaker" than Idempotence] and postulates it for DRLHs with contact labelnodes, as applied in knowledge representation.

Contents

1 INTRODUCTION 1
2 A CONSTRUCTOR ALGEBRA OF INTEGER SEQUENCES 8
3 FUNDAMENTAL CONSTRUCTOR ALGEBRAS FOR LIST PROCESSING 19
3.1 The CONS Algebra of N-tuples 19
3.2 A List Algebra with Negated Elements 31
4 THE BASIC PROPERTIES AND THE BASIC COLLECTIONS 43
4.1 Commutativity 43
4.2 Idempotence 46
4.3 Associativity 53
4.4 The Eight Basic Collections 54
5 THE ADSORPTION PROPERTIES AND GRAPH COLLECTIONS 59
5.1 Binary/N-ary Adsorption and the Ordinary Graph Collections 59
5.2 N-ary/N-ary ADsorption and DRLHs without Contact Labelnodes 71
5.3 Similpotence and DRLHs with Contact Labelnodes 82
6 REFERENCES 106

1 INTRODUCTION

The term "collection data" in the title refers to variable-length compound data characterized solely by the way they combine their arbitrary-type elements [bags, for example, disregard element order], i.e. collections are not prestructured by an underlying "field pattern" [as typical for records and arrays]. The other technical term, "constructor algebras", should be understood as "algebras whose operations are data constructors rather than constructors together with selectors and/or predicates". Since the latter concept may be less

obvious, let us first discuss the motivation for using [such] algebras as data types.

As starting point we take the well-known "types are not sets" (Morris 1973) argument, paraphrasing it positively as "types are more than sets". That is, we are asking what the "minimum extension" of sets might be for making them a useful type concept. The answer to be explored here can be summarized informally by the following "equation":

DATA TYPE = SET OF CONSTRUCTOR NESTINGS + SYSTEM OF EQUALITY AXIOMS

Set of constructor nestings: The sets which we keep as a base component are regarded "constructively" in two senses: First, they are coextensive with recursively enumerable formal languages, normally even with context-free ones. Second, they contain nestings of data constructors over a subset of atomic data items.

System of equality axioms: The minimum set extension yielding a useful data type is obtained by postulating a system of equality axioms for identifying equivalent elements [mainly constructor nestings] of the base set.

As an introductory semi-numeric data type example, let us consider [non-empty] integer sequences consisting of arbitrarily many arbitrary-digit integers. [Although numbers do not belong to the symbolic collection data types focussed in this paper, their familiarity facilitates the characterization of constructor algebras.] Such integer sequences can be written as formal expressions in which sign applications are used not only for indicating positiveness or negativeness, but also for embracing non-digit integers [with parentheses], so as to mark them off from neighbouring integers. In fact, these integer sequences are "more than" the set [context-free language] generated by BNF grammar rules like the following [terminals are quoted]:

```
intseq  ::= integer | integer intseq
integer ::= digit | posint | negint
posint  ::= "+" digit | "+" "(" digseq ")"
negint  ::= "-" digit | "-" "(" digseq ")"
digseq  ::= digit | digit digseq
digit   ::= "0" | ... | "9"
```

The problem is that certain expressions of this formal language are equivalent as integer sequences, e.g. the integers +0 and -0, 1 and +1, +(24) and +(024), as well as -(007) and -(07), hence also integer sequences like 1+0 and +1-0, +(24)1-(007) and +(024)+1-(07), as well as +01+(24)-(007) and -0+1+(024)-(07). In general, 1. the zeroes -0 and +0, 2. "+"-signed and unsigned digits, and 3. integers only differing in their number of leading zeroes should be identified; furthermore, digits signed with and without the use of parentheses -- like -(4) and -4 -- are regarded as indistinguishable, but this will be simply a parenthesis-omission convention applied implicitly. [If we had permitted multiple signs, even further equivalences of this sort would have arisen.] On the other hand, expressions like +(24) and 24 are not equivalent, the former representing a single two-digit integer and the latter an integer sequence both of whose integers are digits.

A useful integer sequences data type should have a "built-in" mechanism for performing the required identifications over the set generated by the above grammar; the equivalence classes thus created can then be viewed as new individuals ["the integer sequences"] on a higher level of abstraction. Now, algebras provide for exactly this. Only one of their components is a set, called a "carrier", which may be finitely generated from an alphabet-like "generating set", much like a formal language; another component is a family of operators over the carrier, which are used in the generation of an entire carrier from its finite generating subset. The essential additional component, however, is an axiom system [written as a set of equations] for identifying elements or classes of elements of the carrier. In our example, to be treated more technically in section 2, the axioms

```
1.      -0 = +0
2.      +x = x      if x is a digit
3a.     +(0x) = +(x)
3b.     -(0x) = -(x)
```

would perform the above-mentioned identifications. [In this algebra parentheses are merely used for embracing the operands for the "+"/"-"-operators, i.e. they can be omitted from signed digits, so that the right-hand sides of 3a./3b. become +x and -x if x is a digit.] Another axiom would establish the associativity of digit and integer juxtaposition, and further axioms could restrict the signing operations to digit sequences.

Obviously, this is not an ordinary algebra of integer sequences with arithmetic and/or string-processing operators; rather it is a <u>constructor</u> algebra whose operators are [binary] juxtaposition and [unary] "+"/"-"-signing, used for constructing data elements of type integer sequence, not for "calculating" with integer sequences. Therefore the axioms encode equivalences between those constructions rather than arithmetic/string-processing laws. The effect of the axiom system is that carrier elements like +01+(24)-(007) and -0+1+(024)-(07) become indistinguishable, as desired for what has now become a useful [integer sequence] type concept.

If a formal language is viewed as defining the syntax of a data type, a constructor algebra also defines this syntax through its carrier [incidentally, algebraic term generation better reflects the operator/operand syntax of functional languages than does grammatical string generation]; but the algebra augments its carrier syntax with rudimentary semantics through its axioms: They at least specify which syntactic constructions are "really identical" [i.e. semantically equivalent], thus embodying what might be called "synonym semantics". We regard this as the minimum requirement for "getting types from sets", a minimum which, however, is often sufficient. If and when desired, selectors/predicates and their more elaborate "access semantics", as used in abstract data types [ADTs, further discussed below], can be added on top of this, for instance in the form of other axioms [e.g. generalized successor and predecessor operators -- hence further arithmetic operators -- can be defined, selector-like, on top of our above constructor algebra example].

With the help of constructor algebras a uniform axiomatic definition of mathematical entitites like [finite] integer sequences, bags, sets, and graphs as explicitly constructed elements [well-formed "formulas", "expressions", or "terms"] of algebraic structures becomes possible. [While we implicitly use an intuitive concept of recursively enumerable sets for the carriers of our algebraic meta formalism, we also -- among many other collections -- give a precise explicit definition of finite sets as an object formalism.] Furthermore, constructor algebras provide a computationally convenient way of viewing these entities: The equational axioms, defining equivalence classes characteristic for each collection type, can be realized as one-way rewrite rules deriving normal form representatives [the natural left-to-right reading of our axiomatic equations will also be the principal rewrite direction of corresponding rules]. Potentials for parallelism which we will exploit in algebraic proofs thus carry over to rule computations. Moreover, the normalization rule applications may occur implicitly, so that programs can rely on the "self-normalization" of each collection instance constructed [this is related to "built-in" axioms inside the deductive machinery of theorem provers (Raulefs et al. 1979)]. In fact, in the functional programming language FIT -- used among other things for DR<u>L</u>H processing (Boley 1980) -- the normalization rules are built directly into the constructor functions of the collections.

This paper grew out of an attempt to conceive collections [in particular, DR<u>L</u>Hs] as classic finitely generated algebras, thus formalizing the "self-normalization" idea which FIT adopted from QA4 (Rulifson et al. 1972): It supplements the operational semantics of FIT's collections with algebraic/axiomatic collection semantics. However, the present application of algebra to collections may also be regarded as the identification of a subclass of the data types which can be formalized with the "initial" ["free"] algebra approach (Goguen et al. 1978) to abstract data types: Instead of using axioms that specify the equivalence of arbitrary constructor/selector/predicate compositions, we use axioms that only specify equivalent constructor nestings. At the outset, we will also assume a second restriction of the ADT-usual "heterogeneous" algebras to the mathematically more wide-spread "homogeneous" algebras; this, however, will lead to the postulation of axioms which are not "positive conditional", hence do not guarantee initiality [(Thatcher et al. 1979), (Ehrig et al. 1980)]. Hence we will show how to eliminate the [negated] conditions on our axioms -- by making use of heterogeneous algebras.

While constructors are considered here as "data-defining" algebraic operators, establishing a <u>data</u> structure layer, selectors and predicates are considered as "data-utilizing" algebraic operators, introduced only <u>on the basis of</u> this proper data layer. From a perspective of abstract data types, we are exploring the hypothesis that the partitioning of algebras into an underlying constructor algebra [ADT] and a superimposed selector/predicate algebra [ADT] provides a better specification methodology than the usual intermixing of data-defining and data-utilizing algebras [ADTs]. This partitioning is related, for instance, to the use of a basic constructor "term language" and "external functions" over it in LCF-formalized "algorithmic specifications" (Loeckx 1981), to the distinction of "generating" and "defined" operations in SRDL (Klaeren 1982), and to the division of ASPIK operation definitions into constructors and other operations (Beierle & Voss 1983). Our reasons for the partitioning are the following: First, it corresponds to the intuitive data/program

separation, which is a useful structuring principle in most situations [giving back "concrete" data to the programmer]. Second, it permits a natural division of labour, because the data-defining operations can be specified, tested, and proved independently of the data-utilizing operations. Third, the data-definition mechanism need not carry over to the data-utilization method, i.e. -- like (Klaeren 1982) -- we can restrict the use of equations to the specification of constructor equivalences, and employ recursive schemata to specify selectors and predicates [in fact, we normally employ recursive FIT rewrite rules]. Fourth, while the data layer must be defined completely, the superimposed layers can be left open-ended, allowing for repeated new uses of the same data [as already pointed out in (Boley 1979, View 3)]. Furthermore, our approach permits the introduction of a single data-defining ADT as the common basis of different data-utilizing ADTs.

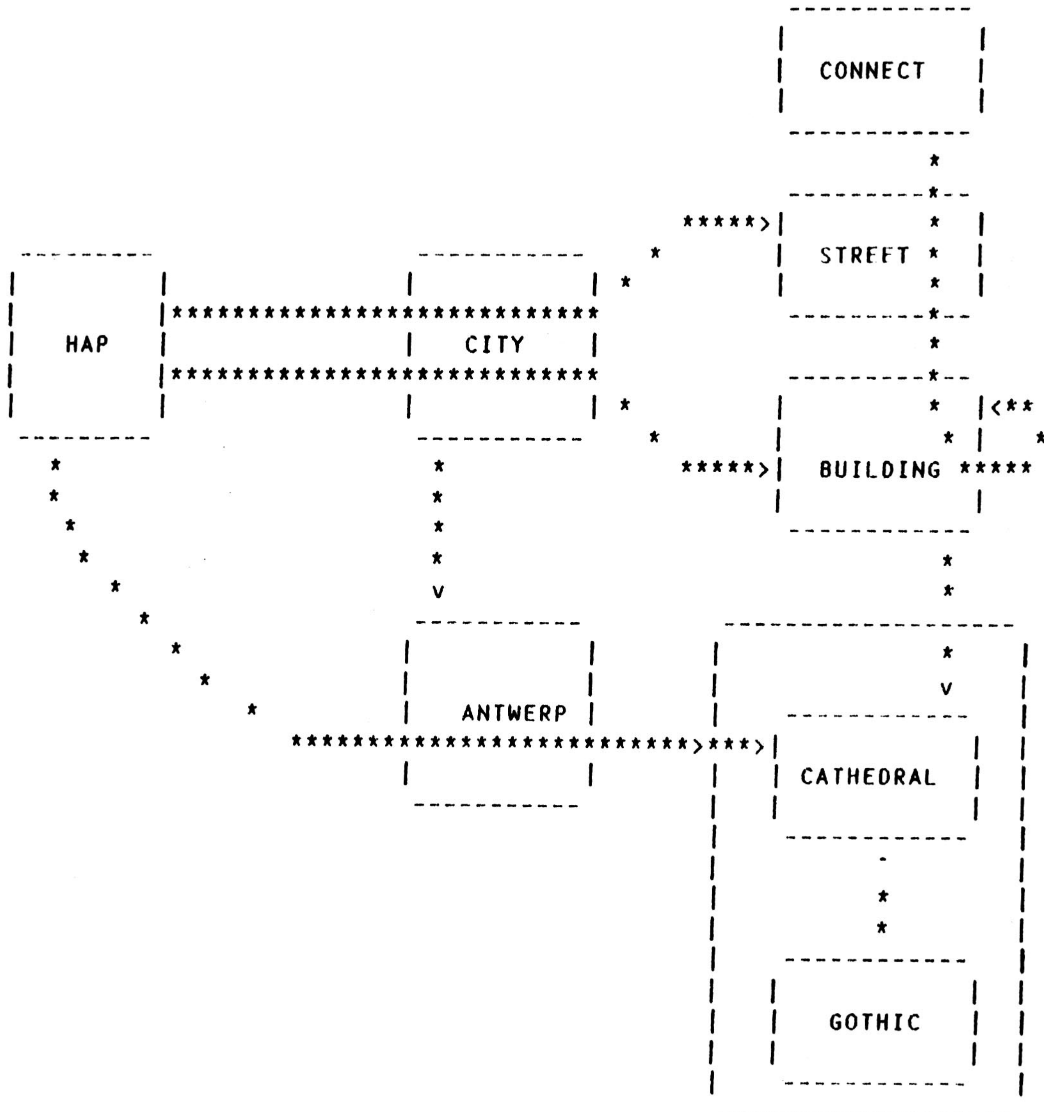
What we will be most concerned with in the following is the definition of constructor algebras for various collection data. These will all build on a fundamental constructor algebra of nested tuples or lists [for some other algebraic characterizations of tuples see (Burstall & Landin 1969), (Oppen 1978), and (McCarthy & Talcott 1980)]. In this way we will arrive at an axiomatic characterization of the differences between -- to take well-known examples -- strings and sets or directed graphs and undirected multigraphs, <u>not</u> those between -- to take the standard ADT examples -- stacks and queues, characterized by their different selection disciplines [in the data layer we simply regard both as tuples]. More importantly, we use constructor algebras to characterize a number of less well-known data structures, in particular DR<u>L</u>Hs. However, the diversity of the examples [ranging from integer sequences to hypergraphs] and the complexity of some of them [e.g. DR<u>L</u>Hs with contact labelnodes] also illustrates the generality of the constructor algebra approach, enabling the formalization of many more data types once they can be viewed as collections [where the collection view of some structures, e.g. of graphs, may not be obvious from the outset].

To summarize a principal goal of this paper, we try to demonstrate that the data-definition component of a considerable subclass of data types -- stacks and queues excluded -- captures a non-trivial part of their specification, indeed often the essential one. For DR<u>L</u>H types, we can even state that their constructor algebra specification [section 5.3] captures <u>the</u> fundamental intuitive concept. Consequently, in this paper we will not attempt to enrich the DR<u>L</u>H algebras with selectors and predicates; a number of these and higher operations were defined as FIT rewrite rules in (Boley 1980). [By this point the reader may have noticed that our approach -- though conceivable as a specialization of algebraic ADTs -- stands in contrast to a central tenet of classical ADT theory, namely that a data type should be defined soley through its "external" access behavior; what we view as being often more characteristic is its "internal" synonym structure. We leave the question open how many data types are better characterized "externally", like -- presumably -- stacks and queues, and how many "internally", like -- presumably -- the collection data in this paper, and which ones will prove to be more useful in practical programming.]

Since collections are typical data structures for artificial intelligence [AI] applications, a more general goal of this work is to contribute to the attempts at cross-fertilization between ADT and AI languages, begun with languages like CLEAR (Burstall & Goguen 1977),

CSSA (Boehm et al. 1977), and TELOS (Travis et al. 1977) around 1977, and then continued, for instance, with HOPE (Burstall et al. 1980), ESP (Chikayama 1983), PROLOG/KR (Nakashima & Suzuki 1983), HIMIKO (Dogen 1983), LCA (Bellia et al. 1984), EQLOG (Goguen & Meseguer 1984), APE (Bartels et al. 1981), ASPIK (Beierle & Voss 1983), and FIT (Boley 1983); the IJCAI-83 proceedings exhibit some further ADT/AI relations [(Guiho 1983), (Dilger & Womann 1983), (Reimer & Hahn 1983)].

To be specific, while around 1976 we based "a theory of [knowledge] representation" -- and the DRLH representation language of 1975-1977 -- on formal languages, so that equivalences between their representation expressions had to be deferred to a less formalized meta level, we can now -- via the FIT programming language of 1978-1983 -- base it on constructor algebras, with equivalences completely formalized on the object level, thus proceeding from "knowledge representation languages" to "knowledge representation algebras". For example, the following DRLH taken from (Boley 1980) is a generalized semantic network that represents facts about the city of Antwerp, using relations like HAP ["has as part"] as starting boxes [labels] of hyperarc arrows visiting other boxes [nodes]:



We can describe this diagram as the following compact expression of a formal language, similar to the DRLH expression in that paper:

```
(DRLH
 (TUPLE BUILDING CATHEDRAL)
 (TUPLE CITY ANTWERP)
 (TUPLE CONNECT STREET BUILDING BUILDING)
 (TUPLE HAP ANTWERP (DRLH @CATHEDRAL (TUPLE GOTHIC CATHEDRAL)))
 (TUPLE HAP CITY BUILDING)
 (TUPLE HAP CITY STREET))
```

But due to the sets-refining nature of DRLHs, we could also have used the redundant expression

```
(DRLH
 (TUPLE CITY ANTWERP)
 (TUPLE HAP CITY STREET)
 BUILDING
 (TUPLE HAP CITY BUILDING)
 (TUPLE CITY ANTWERP)
 (TUPLE HAP ANTWERP (DRLH @CATHEDRAL (TUPLE GOTHIC CATHEDRAL)))
 (DRLH (TUPLE GOTHIC CATHEDRAL))
 (TUPLE BUILDING CATHEDRAL)
 STREET
 BUILDING
 (TUPLE CONNECT STREET BUILDING BUILDING))
```

which is equivalent to the compact expression because its syntactic differences [e.g. the order and duplication of DRLH subexpressions, atomic expressions also occurring in TUPLE expressions of one DRLH] do not affect the intended semantics. Using formal languages, this equality had to be regarded as an equivalence relation extraneous to the knowledge representation language proper. Alternatively, in functional languages like LISP or FIT, we can reinterpret the expressions as collections implemented as calls to a DRLH constructor function with embedded calls to a TUPLE constructor function, etc., such that the redundant function nesting evaluates to the compact one, which, in turn, evaluates to itself because it is in normal form (Boley 1980). What we propose now is to treat such normalizations as a special kind of equivalence proof that is made possible by reinterpreting the expressions as terms of a constructor algebra -- namely of ALCO5[.lr,.fr,hr,*lr,mr,cr,*C,*I,AD,aD,S] in section 5.3, with generating set A = {ANTWERP, BUILDING, ..., STREET}. The equality of the algebraic terms corresponding to the previous expressions,

```
 (BUILDING.CATHEDRAL.`)
*(CITY.ANTWERP.`)
*(CONNECT.STREET.BUILDING.BUILDING.`)
*(HAP.ANTWERP.(@CATHEDRAL*(GOTHIC.CATHEDRAL.`)*\).`)
*(HAP.CITY.BUILDING.`)
*(HAP.CITY.STREET.`)
*\
```

=

```
 (CITY.ANTWERP.`)
*(HAP.CITY.STREET.`)
*BUILDING
*(HAP.CITY.BUILDING.`)
*(CITY.ANTWERP.`)
*(HAP.ANTWERP.(@CATHEDRAL*(GOTHIC.CATHEDRAL.`)*\).`)
*((GOTHIC.CATHEDRAL.`)*\)
*(BUILDING.CATHEDRAL.`)
*STREET
*BUILDING
*(CONNECT.STREET.BUILDING.BUILDING.`)
*\
```

is then part of the knowledge representation algebra itself, which thus provides a higher level of abstraction that frees the "knowledge engineer" for concentrating on the application domain.

AI-adapted ADTs as explored here and object-oriented representations on the one hand, together with functional and relational [logical] programming on the other hand (Boley 1983), should be amenable to an integration in future AI languages that extends the present collections/FIT integration. However, it must be noted that even our restricted ADT/AI combination is still rather tentative and will require further mathematical, computational, and [epistemo]logical studies.

2 A CONSTRUCTOR ALGEBRA OF INTEGER SEQUENCES

Let us exemplify the constructor algebra methodology [not the typical constructor algebra application] by resuming our introductory discussion of integer sequences, now elaborating them formally. Their constructor algebra will use a generating set A, which generates the carrier M with three operations "*" [binary concatenation, like juxtaposition and unlike ordinary multiplication], "+" [unary positive sign], and "-" [unary negative sign]. To provide a value for exceptional situations, A contains the distinguished element "%" [infinity]. The other members of A -- including the distinguished element "0" [zero] and at least one more element -- will be interpreted as digits, i.e. all positional number systems with bases |A|-1 = 2, 3, ... will be permitted for integers. [Our algebra of integer sequences will thus be more complicated than the usual abstract data type sequences-of-integers, which applies a kind of unary number system with the successor operator playing the role of the digit "1"; on the other hand, even a small integer like 10 is represented more concisely as +(1*(0*(1*0))) or +(1*0*1*0) with our binary system -- not to speak of our decimal system -- than as s(s(s(s(s(s(s(s(s(s(0)))))))))) or ssssssssss0 with the ADT-usual unary system.]

The three operations of integer sequences are "algebraic" in the sense that they are defined for all elements of the carrier M and denote elements which are again in M [M is closed under "*", "+", and "-"]. The algebra is a "constructor algebra" because the operations are constructors in the abstract syntax or VDL sense, i.e. they build composite objects from given ones, rather than taking objects apart [selectors] or probing them [predicates]. The "*" operator constructs a digit sequence from two digit sequences [either of which may in

particular be a single digit] or an integer sequence from two integer sequences [either of which may in particular be a single integer or a digit sequence]. While a "*"-application like *(a,b) could thus denote the juxtaposition "ab" of its argument terms [cf. section 1], we let it denote its own infix notation "a*b"; this stresses the fact that, in programming terminology, *(a,b) is "not really evaluated", keeping it open to all possible evaluation results. Similarly, both the "+" and "-" operators construct an integer from a digit sequence or from an integer [thus multiple signs are now allowed] by denoting their own prefix notation, i.e. +(a) denotes "+a" and -(a) denotes "-a" [the "erroneous" application of "+"/"-" to non-unitary integer sequences will be treated later]. Algebraically speaking, we have a term or word algebra construction of a "free" [in the ADT view: "initial"] algebra, since the applications of all operators to all arguments denote the applicative terms themselves. This has the general advantage that -- oversimplifying -- from a free algebra all other algebras with the same "signature" [i.e. the same generating set and corresponding operations of the same arities] are obtainable as special cases. Furthermore, there is a close affinity between the restriction of operators to constructors and the initiality property of algebras: Constructor terms are the ones it is most natural to let denote themselves, and even initial constructor/selector/predicate algebras stick to them for normal forms.

These self-denoting terms are not as "passive" as it might seem, because of the axioms introduced over them, permitting, for example, a term like +(0*b) to "normalize" to the term +b.

As axioms we will not merely allow [unconditional] equations but also "conditional equations" of the form equation if condition [or, as a usual logical implication, condition => equation], which are only applicable if the condition over their variables evaluates to 'true'.

In our case conditions consist of a boolean composition of 1. equalities of the form variable = term, where the term may contain free variables, interpretable as being "innermost" existentially quantified [the quantifiers are as close to the variables as possible], and 2. tests for membership in the generating set [possibly MINUS a distinguished element] of the form variable IN set. The boolean operations may include "=" and IN negation, thus providing for inequalities, "<>", and non-membership tests, NOTIN.

The innermost existential quantifiers pose no specific problems because a. practically, the existentially quantified variables are determinable easily by syntactically matching the right-hand-side term pattern to the term denoted by the left-hand-side variable, and b. theoretically, they are unnecessary: if negated, the innermost existential quantifiers correspond to prenex normal form universal quantifiers [since an equation cannot contain any variables thus quantified in its condition, we will not have to distinguish the "quasi prenex normal form" FORALL(n1,...,nN)(p(n1,...,nN)) => equation from the "proper prenex normal form" FORALL(n1,...,nN)(p(n1,...,nN) => equation)]; if unnegated, they can be eliminated altogether by substituting the quantified term for the variable in the equation. [A general discussion of the use of existential quantifiers in ADTs can be found in (Broy et al. 1979).]

Similarly, the IN predicate can be eliminated by replacing the conditional equation by as many equations as the [finite] cardinality of set indicates, and in these using the set members individually; its negation, NOTIN, can be eliminated by replacing the conditional equation by as many conditional equations as there are operators opj in the algebra, and in these testing variable = opj(n1,...,nNj) [the newly introduced free variables n1, ..., nNj can again be eliminated as discussed in b. above].

While in the present section we will not show the elimination of all kinds of conditional equations [in particular, of negated conditions], such an "unconditioning" will be developed as a means for guaranteeing intitiality, starting in section 3.

After these preliminary remarks we are ready to proceed to our algebraic definition of integer sequences.

Definition1:

A	generating set [a finite set with |A| $\geq$ 3]
% IN A	distinguished element [infinity symbol]
0 IN A	distinguished element [zero symbol]
ALINS = (M,*,+,-)	algebra with
M	carrier generated by A with "*", "+", "-"
* : M x M -> M *(m1,m2) = m1*m2	binary operation [concatenation]
+ : M -> M +(m) = +m	unary operation [positive signing]
- : M -> M -(m) = -m	unary operation [negative signing]

We will now postulate an axiom system for this algebra, starting with axioms for treating exceptional situations [signrestriction, infinitypropagation, integernesting], followed by axioms dealing with the more typical cases [multisign, associativity, signedzero, plusification, leadingzero]. For referring to an axiom, we will often use an acronym [possibly followed by a digit], which is introduced together with its defining equation [by underscoring the corresponding characters]. When we refer to axiom ax [not using a digit suffix] we will mean this single axiom if there are no suffixed axioms for ax, and mean the whole group of axioms ax1, ..., axN otherwise.

The first group of axioms to be introduced in the ALINS algebra "restricts" the signing operations "+" and "-" to arguments which can be used as integers. More precisely, if a "+"-argument is a "*"-concatenation one of whose components has an irreversible "-"-sign in front of an element not equal to "0" [sr3-sr5] or, complementarily, if a "-"-argument contains a "+"-sign in front of an "*"-embedded "*"-concatenation component without any leading "0" [sr8-sr10], then the entire "+"/"-"-signed term is identified with the, respectively, "+"/"-"-signed distinguished element "%". Similarly, non-leading "+"-components in "+"-arguments [sr1,sr2] and non-leading "-"-components in "-"-arguments [sr6,sr7] yield "+%" and "-%",

respectively. Formally, the above phrase "one of whose components" can be notated by using x = -m' or x = +(m'*m'') in axioms with "+"/"-"-arguments of the form m1*x*m2, because the associativity of "*" will permit m1 and m2 to denote arbitrary prefixes and postfixes, respectively, except empty ones [we will not introduce an identity element for "*"]; hence additional axioms with arguments of the forms x*m and m*x are called for. In this notation the above term "non-leading" corresponds to either of the forms m1*x*m2 or m*x. The above term "irreversible" can be expressed as a test for whether the "-"-signed m' is in the set A MINUS {0} [where "MINUS" denotes set difference] or has the form (n1*n2), and hence cannot be negative itself.

```
signrestriction1:  +(m*+(m'*m'')) = +%  if m' <> 0 and m' <> (0*n)
signrestriction2:  +(m1*+(m'*m'')*m2) = +%  if m' <> 0 and m' <> (0*n)

signrestriction3:  +(-m'*m) = +%  if m' IN A MINUS {0} or m' = (n1*n2)
signrestriction4:  +(m*-m') = +%  if m' IN A MINUS {0} or m' = (n1*n2)
signrestriction5:  +(m1*-m'*m2) = +% if m' IN A MINUS {0} or m'=(n1*n2)

signrestriction6:  -(m*-m') = -%  if m' IN A MINUS {0} or m' = (n1*n2)
signrestriction7:  -(m1*-m'*m2) = -% if m' IN A MINUS {0} or m'=(n1*n2)

signrestriction8:  -(+(m'*m'')*m) = -%  if m' <> 0 and m' <> (0*n)
signrestriction9:  -(m*+(m'*m'')) = -%  if m' <> 0 and m' <> (0*n)
signrestriction10: -(m1*+(m'*m'')*m2) = -%  if m' <> 0 and m' <> (0*n)
```

These axioms can be used to explain the working of our conditional equations:

Suppose we are given the term +(7*+((0*0)*4)). To check whether the conditional equation sr1 is applicable to this term, we match the left-hand side of the equation to it, obtaining the bindings m = 7, m' = (0*0), and m'' = 4. Now we can evaluate the conjunctive condition, obtaining 'false' because of the second conjunct: 1. The conjunct m' <> 0, or equivalently, not(m' = 0), is true since the equality (0*0) = 0 does not hold [although +(0*0) = 0 does hold]. 2. The variable n in the other conjunct, m' <> (0*n) [abbreviating not(m' = (0*n))], is understood as being innermost existentially quantified, i.e. this inequation is shorthand for not(EXIST(n)(m'=(0*n))) [in prenex normal form it becomes FORALL(n)(not(m'=(0*n)))] and means that there does not exist an n such that m' [= (0*0)] is equal to (0*n), which is false because we can choose n = 0. [This value of the variable n can be found by matching (0*n) against (0*0), or structurally decomposing (0*0) according to the pattern (0*n), thus reducing "(0*n) = (0*0)" to "0=0 and n=0"; cf. the Equality relation in section 3.] Hence sr1 is not applicable to the given term. The reason for not permitting applications of sr1 to terms like these can be seen by reading the following chain of equations backward [the a, lz1, and p axioms are explained below]:

```
+(7*+((0*0)*4)) =a=
+(7*+(0*(0*4))) =lz1=
+(7*+(0*4)) =lz1=
+(7*+4) =p=
+(7*4)
```

The unconditioned version of sr1 would collapse the positive integers +(0*1), ..., +(7*4), ... to "+%". If, however, we have the term +(7*+((1*0)*4)) the conditional equation sr1 is applicable, correctly yielding "+%", because m' = (1*0) and the conditions (1*0) <> 0 and (1*0) <> (0*n) are true. [Although sr1 would again be inapplicable to +(7*+((0*1)*4)), since (0*1) <> (0*n) is false, this term could be transformed to the term +(7*+(1*4)), to which sr1 is applicable, since 1 <> (0*n) is true.]

Now suppose we are given the term +(--5*8). An attempt to apply the conditional equation sr3 to this term produces the bindings m' = -5 and m = 8, but then fails because both disjuncts of the condition are false: 1. Although -5 is not equal to 0, it is not a member of A. 2. The free variables n1 and n2 in m' = (n1*n2) are understood as being innermost existentially quantified, i.e. this equation is shorthand for EXIST(n1,n2)(m'=(n1*n2)) [already in prenex normal form] and means that m' [= -5] is equal to the "*"-concatenation of some terms n1 and n2, which is false too. [The match (n1*n2) = -5 fails since -5 cannot be structurally decomposed into a "*"-term.] In sr3 the existential quantifier and the IN predicate can be eliminated via 'or-splitting', obtaining two conditional equations, one for each of the disjuncts. The conditional equation +(-m'*m) = +% if EXIST(n1,n2)(m'=(n1*n2)), resulting from the second disjunct, can then be transformed into an equation +(-(n1*n2)*m) = +%, in which m' is replaced by the term (n1*n2), whose variables were existentially quantified. The conditional equation +(-m'*m) = +% if m' IN A MINUS {0}, resulting from the first disjunct, can again be split [for arbitrary A = {%, 0, a1, ..., aN}] into the N+1 equations +(-%*m) = +%, +(-a1*m) = +%, ..., +(-aN*m) = +%. That sr3 should not be applicable to the given term can be seen as above by considering [ms2 is explained below] +(--5*8) =ms2= +(+5*8) =p= +(5*8). If we instead take the term +(-5*8), sr3 is applicable with m' = 5 because 0 <> 5 IN A.

Another group of axioms is responsible for passing "+%"-terms, which are not addressed by the signrestriction axioms, out of integers, thereby inverting the sign of "%" if an integer is negative.

infinitypropagation1: +(+%*m) = +%
infinitypropagation2: +(m*+%) = +%
infinitypropagation3: +(m1*+%*m2) = +%

infinitypropagation4: -(+%*m) = -%
infinitypropagation5: -(m*+%) = -%
infinitypropagation6: -(m1*+%*m2) = -%

Two related axioms handle sign-compatible "left-nestings" of integers in integers; that is, "+"-signs inside leading "+"-arguments and irreversible "-"-signs inside leading "-"-arguments are eliminated.

integernesting1: +(+m'*m) = +(m'*m)
integernesting2: -(-m'*m) = -(m'*m) if m' IN A MINUS {0} or m'=(n1*n2)

A further pair of axioms deals with multiple signs in the obvious arithmetic manner:

multisign1: -+m = -m
multisign2: --m = +m

For the other two sign combinations "++" and "+-" no special axioms are required because of the "plusification" axiom below.

The signrestriction, infinitypropagation, integernesting, and multisign properties could also have been introduced as part of the operator definitions [written here in an axiom-like pattern notation]:

```
+ : M -> M
+(m*+(m'*m'')) = +%  if m' <> 0 and m' <> (0*n)
   . . .
+(m1*-m'*m2) = +%  if m' IN A MINUS {0} or m' = (n1*n2)
+(+%*m) = +%
   . . .
+(m1*+%*m2) = +%
+(+m'*m) = +(m'*m)
+(m) = +m   otherwise

- : M -> M
-(m*-m') = -%  if m' IN A MINUS {0} or m' = (n1*n2)
   . . .
-(m1*+(m'*m'')*m2) = -%  if m' <> 0 and m' <> (0*n)
-(+%*m) = -%
   . . .
-(m1*+%*m2) = -%
-(-m'*m) = -(m'*m)  if m' IN A MINUS {0} or m' = (n1*n2)
-(+m) = -m
-(-m) = +m
-(m) = -m   otherwise
```

However, these operator definitions would obstruct the self-denotation principle -- e.g. +(3*-2) would denote "+%" rather than "+(3*-2)" and -(+8) would denote "-8" rather than "-+8" -- and would thus also prevent the algebra from being initial. [The algebra would be non-initial because it would fail to distinguish terms like +(3*-2) and +%, -+8 and -8, etc., of its signature's word algebra ALINS, although the equality of these pairs of terms would not be derivable <u>from the axioms</u>.] This disadvantage seems not to be offset by the potential advantages that

1. only the actually required left-to-right reading of these equations is specified -- e.g. only +(3*-2) = +% and -(+8) = -8, rather than also +% = +(3*-2) and -8 = -+8 -- and

2. the application of other axioms cannot interfere with them, because they correspond to axiom applications to terms "in statu nascendi".

Therefore we will not use such axioms-incorporating operation definitions further. Regarding 1., it should be noted that in our equational formulation no problems arise through identities like +(6*+(1*1)) =[=sr1= +% =sr4=]= +(3*-2), derivable via "+%", because these only mean that all "+"-terms having an integer-useless argument are "swallowed" by the equivalence class represented by "+%"; similarly, "-"-terms with useless arguments become "-%".

In other words, since "+"/"-"-calls with arguments not usable as integers can be regarded as being "erroneous", their identification with "+%"/"-%" can be regarded as "overloading" these signed infinity symbols with the role of "error elements". In our constructor algebra

no problems arise from such an overloading, so that no additional error element, distinct from "+%"/"-%" and all other elements, is called for here. For our integer sequences, the "+%"/"-%"-overloading only means that there now are additional possibilities for constructing "+%"/"-%" and terms containing it. For instance, 1*(+(3*-2)*6) is merely a more complicated way of constructing 1*(+%*6). Such multiple construction possibilites also exist for "%"-less terms. For example, -++8 is merely a redundant way of constructing -8.

Since non-digit integers are regarded as signed strings of digits and, similarly, non-unitary integer sequences as strings of integers, an important axiom is the associativity of "*"-concatenation.

associativity: (m1*m2)*m3 = m1*(m2*m3)

As customary, we use associativity for omitting parentheses from strings altogether and reinsert them only if and when [and at the string positions where] required for the application of other axioms [however, an entire "*"-concatenation of digits must be put into parentheses before signing it with "+" or "-", because otherwise that sign would apply to the first digit only].

The remaining axioms correspond to those already sketched in the introduction, which we now formulate with the explicit "*"-concatenation operator instead of with simple juxtaposition.

signedzero: -0 = +0

plusification: +m = m if m <> n*n'

leadingzero1: +(0*m) = +m if m <> +n and m <> -n'
leadingzero2: -(0*m) = -m if m <> +n and m <> -n'

Note that the plusification condition, m <> n*n', is not only true if m is a digit [as the condition in the introduction] or "%", but also if m itself has another sign [cf. the multisign axioms]; for a generating set with the non-"0" digits a1, ..., aN, the conditional plusification equation could thus be replaced by the unconditional equations +0 = 0, +a1 = a1, ..., +aN = aN, +% = %, ++m = +m, and +-m = -m. The leadingzero condition, m <> +n and m <> -n', prevents the lz transformation of terms which are "%"-reducible by any of the signrestriction axioms permitting a leading zero [i.e. by all except sr3 and sr8], so that equation chains like

```
+(6*4) =p= ++(6*4) =lz1= +(0*+(6*4)) =sr1= +%
-4 =p= +-4 =lz1= +(0*-4) =sr4= +%
+4 =ms2= --4 =lz2= -(0*-4) =sr6= -%
-(6*4) =ms1= -+(6*4) =lz2= -(0*+(6*4)) =sr9= -%
```

are not allowed.

As a first instantiation of the ALINS algebra let us consider integer sequences in the decimal number system, having the generating set A = {%, 0, 1, ..., 9}. With all axioms available, we can now formally prove the integer identities mentioned in the introduction in the decimal integer sequence algebra generated by A [axioms and previously derived lemmas used in a proof step will be written inside the step's equality sign]:

```
Lem00: +0 =sz= -0
Lem01: 1 =p= +1
Lem02: +(2*4) =lz1= +(0*(2*4)) =a= +(0*2*4)
Lem03: -(0*0*7) =a= -(0*(0*7)) =lz2= -(0*7)
```

Building on these, we can derive the integer sequence identities [the proofs involve parallel transformations of subterms, as indicated by "|" separators between axiom and lemma names]:

```
Lem04: 1*+0 =p|sz= +1*-0
Lem05: +(2*4)*1*-(0*0*7) =Lem02|p|Lem03= +(0*2*4)*+1*-(0*7)
Lem06: +0*1*+(2*4)*-(0*0*7) =sz|p|Lem02|Lem03= -0*+1*+(0*2*4)*-(0*7)
```

The above +24 = +024 and -007 = -07 proofs suggest that the associativity and leadingzero axioms can be used for several leading zeroes in an alternating manner for, respectively, pre/postprocessing the parenthesis structure and eliminating/generating the "0" immediately after the sign "+" or "-". An example involving three such alternations is the proof of +00040200 = +40200, eliminating three leading zeroes; fortunately, with our axioms, intermediate [+00040200 <> +0004200] and trailing [+00040200 <> +0004020] zeroes are never eliminated [all non-final zeroes <u>could</u> be eliminated if lz1 and lz2 were replaced by 0*m = m, i.e. if 0 were a general left identity]:

```
Lem07: +(0*0*0*4*0*2*0*0) = +(4*0*2*0*0)
Proof: +(0*0*0*4*0*2*0*0) =a=
       +(0*(0*0*4*0*2*0*0)) =lz1=
       +(0*0*4*0*2*0*0) =a=
       +(0*(0*4*0*2*0*0)) =lz1=
       +(0*4*0*2*0*0) =a=
       +(0*(4*0*2*0*0)) =lz1=
       +(4*0*2*0*0)
```

A final example for equivalences between decimal integer sequences concerns a sequence of length 6:

```
Lem08: --9*-(0*4)*+(0*2*-3*5)*+-(7*6)*-+8*+(+0*1) = 9*-4*%*-(7*6)*-8*1
Proof: --9*-(0*4)*+(0*2*-3*5)*+-(7*6)*-+8*+(+0*1) =ms2|lz2|a|p|ms1|p=
       +9*-(4)*+(0*(2*-3*5))*-(7*6)*-8*+(0*1) =p|lz1|lz1=
       9*-4*+(2*-3*5)*-(7*6)*-8*+1 =sr5|p=
       9*-4*+%*-(7*6)*-8*1 =p=
       9*-4*%*-(7*6)*-8*1
```

Recall that no axiom is required for parenthesis omissions like -(4) = -4, so that these can be integrated with other axiom applications as in +(0*1) =lz1= +1; also note the harmlessness of first performing spurious transformations on "erroneous" subexpressions such as +(0*2*-3*5) =a= +(0*(2*-3*5)) =lz1= +(2*-3*5) and only then reducing them to "+%" ["confluent" with direct "+%"-reductions like +(0*2*-3*5) =a= +((0*2)*-3*5) =sr5= +%].

As another instantiation of the algebra ALINS let us consider integer sequences in the binary number system, having the minimal generating set A = {%, 0, 1}. Using the axioms we can proof the following identities:

```
Lem09: +(0*1*1) = +(1*1)
Proof: +(0*1*1) =a=
       +(0*(1*1)) =lz1=
       +(1*1)

Lem10: -(0*0*0*1*0*1)*-(0*0*0) = -(1*0*1)*0
Proof: -(0*0*0*1*0*1)*-(0*0*0) =a|a=
       -(0*(0*0*1*0*1))*-(0*(0*0)) =lz2|lz2=
       -(0*0*1*0*1)*-(0*0) =a|lz2=
       -(0*(0*1*0*1))*-0 =lz2|sz=
       -(0*1*0*1)*+0 =a|p=
       -(0*(1*0*1))*0 =lz2=
       -(1*0*1)*0
```

The algebra ALINS can be enriched by a successor operation "s" and a predecessor operation "p" for arbitrary number systems, obtaining the following algebra ALINS-sp. While normally in ADTs the successor operation is used as a constructor, in our formalization of integer sequences it becomes a selector-like operator.

DefinitionII:

```
A = {a, a0, a1, ..., aN}   generating set
a = %                      distinguished element
a0 = 0                     distinguished element

ALINS-sp = (M,*,+,-,s,p)   algebra with
(M,*,+,-)                  as ALINS

s : M -> M                 unary operation [successor]
s(m) = sm

p : M -> M                 unary operation [predecessor]
p(m) = pm
```

In addition to the axioms of ALINS the following successor and predecessor axioms are postulated for ALINS-sp [for "s" and "p" applications readability will be enhanced by retaining the application parentheses].

```
successorof+%:      s(+%)       = +%
successorof-%:      s(-%)       = -%

successorofa0:      s(a0)       = a1
      . . .
successorofaN-1:    s(aN-1)     = aN
successorofaN:      s(aN)       = +(a1*a0)

successorofma0:     s+(m*a0)    = +(m*a1)
      . . .
successorofmaN-1:   s+(m*aN-1)  = +(m*aN)
successorofmaN:     s+(m*aN)    = +(s+(m)*a0) if m <> -a1 and m <> (a0*n)

successorof-m:      s-(m)       = -p+(m)

successorofm1m2:    s(m1*m2)    = s(m1)*s(m2)
```

```
predecessorof+%:    p(+%)        = +%
predecessorof-%:    p(-%)        = -%

predecessorofa0:    p(a0)        = -a1
predecessorofa1:    p(a1)        = a0
      . . .
predecessorofaN:    p(aN)        = aN-1

predecessorofma0:   p+(m*a0)     = +(p+(m)*aN)   if m <> a0 and m <> (a0*n)
predecessorofma1:   p+(m*a1)     = +(m*a0)
      . . .
predecessorofmaN:   p+(m*aN)     = +(m*aN-1)

predecessorof-m:    p-(m)        = -s+(m)

predecessorofm1m2:  p(m1*m2)     = p(m1)*p(m2)
```

In the following discussion examples will be given for a decimal ALINS-sp instantiation. The above equations define "s" and "p" for all carrier elements, even though there are no explicit equations for, say, "s" and "p" applied to "+"-signed digits: Since the plusification axiom identifies +aI with aI for 0 ≤ I ≤ N, the corresponding equations s(+aI) = s(aI) and p(+aI) = p(aI) hold implicitly also. In general, once equivalence classes have been established through a constructor algebra, we are free to choose any of their representatives for defining the higher operators of its enrichments. While for the |A|-1 equivalence classes {+aI, aI}, containing two elements, little is gained by picking representatives, for the 2 equivalence classes {+%, %, ..., +(6*+(1*1)), ..., +(3*-2), ...} and {-%, ..., -(6*-(1*1)), ..., -(3*+2), ...}, containing infinitely many elements, their use is more interesting: To take one example, the equation s+%, defining s(+%) as +%, represents an infinity of further equations [derived with our constructor algebraic equalities]

```
s% =[=p= s(+%) =s+%=]= +%
   . . .
s+(6*+(1*1)) =[=sr1= s(+%) =s+%=]= +%
s+(3*-2) =[=sr4= s(+%) =s+%=]= +%
```

Incidentally, in the "error" view of terms like +(3*-2) and +(6*+(1*1)), the equation s+% can be regarded as handling "erroneous" arguments to "s". Like the application of spurious leadingzero axioms, the application of spurious "s" or "p" transformations to such terms is harmless. For example, the derivation s+(-1*0) =sm0= +(-1*1) =sr3= +% is just a "confluent variation" of s+(-1*0) =sr3= s(+%) =s+%= +%. However, for one situation the "s" and "p" applicability must be prohibited explicitly. A "s"/"p" transition between the "erroneous" term +(-1*9) and the "non-erroneous" term +(0*0) is only prevented by the first conjunct of the conditions on the carry-generation-simulating equations smN and pm0. In one direction, the conjunct m <> -a1 makes smN inapplicable to the term s+(-1*9), thus prohibiting its transformation to +(0*0). In the other direction, the conjunct m <> a0 makes pm0 inapplicable to the term +(0*0), prohibiting its transformation to +(-1*9). The second conjunct m <> (a0*n) of both conditions prevents the application of smN and pm0 to integers with [redundant] leading zeroes, since otherwise equalities like s+((0*-1)*9) = +(0*0) and p+((0*0)*0) = +(-1*9) would be still derivable.

For the binary number system, with aN = a1 = 1, the successor axioms s0, ..., sN, sm0, ..., smN and the corresponding predecessor axioms specialize as follows.

```
successorofa0:    s(0)     = 1
successorofa1:    s(1)     = +(1*0)
successorofma0:   s+(m*0)  = +(m*1)
successorofma1:   s+(m*1)  = +(s+(m)*0)  if m <> -1 and m <> (0*n)

predecessorofa0:  p(0)     = -1
predecessorofa1:  p(1)     = 0
predecessorofma0: p+(m*0)  = +(p+(m)*1)  if m <> 0 and m <> (0*n)
predecessorofma1: p+(m*1)  = +(m*0)
```

Some "computations" in this algebra are [notice the integernesting normalization after "carry propagation"]:

```
Lem11: s+(1*0) =sm0= +(1*1)
Lem12: s+(1*1) =sm1= +(s+(1)*0) =p= +(s1*0) =s1= +(+(1*0)*0)
                                       =in1= +((1*0)*0) =a= +(1*0*0)
Lem13: s+(1*0*1) =sm1= +(s+(1*0)*0) =sm0= +(+(1*1)*0) =in1= +((1*1)*0)
                                                            =a= +(1*1*0)
Lem14: s+(1*1*1) =sm1= +(s+(1*1)*0) =Lem12= +(+(1*0*0)*0)
                                   =in1= +((1*0*0)*0) =a= +(1*0*0*0)
Lem15: p+(1*0) =pm0= +(p+(1)*1) =p= +(p1*1) =p1= +(0*1) =lz1= +1
Lem16: p+(1*1) =pm1= +(1*0)
Lem17: s-(1*0) =s-m= -p+(1*0) =Lem15= -+1 =ms1= -1
Lem18: s(+(1*0)*-(1*0)) =smm= s+(1*0)*s-(1*0) =Lem11|Lem17= +(1*1)*-1
```

The algebra ALINS-sp could be further enriched to an algebra with the usual arithmetic operators generalized to sequences. Alternatively, the following equations for a length operation "l" on integer sequences could be used as the key component of another ALINS-sp enrichment:

```
l(m)      = s(a0)  if m <> n*n'
l(m1*m2) = s(l(m2))  if m1 <> n*n'
```

Besides these ALINS-sp enrichments, it is also possible to enrich the original algebra ALINS directly in an alternative manner; as a groundwork for string processing, the sequence constructor operation "*" could be complemented by sequence selector operations such as head and tail.

3 FUNDAMENTAL CONSTRUCTOR ALGEBRAS FOR LIST PROCESSING

As a basis for all subsequent development, this section first introduces an algebraic formalization of lists and then augments it by allowing negated list elements [used for erasing unnegated ones].

3.1 The CONS Algebra of N-tuples

It is well-known that the data structure of strings -- whose expansion to integer sequences was discussed in the previous section -- is definable algebraically as a semigroup [in particular, a monoid], where the binary operation is interpreted simply as the juxtaposition of two semigroup elements, forming another semigroup element. Since only the property of associativity is postulated as an axiom for the semigroup operation, semigroups are often regarded as the most poorly structured algebras conceivable. Paradoxically however, the data structure of tuples [lists], richer in structure than strings because of the ability to have tuples nested in tuples, is definable by an even more poorly structured [in fact, unstructured] algebra, by dropping the associativity axiom as well. Clearly, the "paradox" can be resolved by more careful terminological distinctions: The property of associativity, increasing the <u>axiomatic structure of an algebra</u> [like any other postulate], decreases the <u>syntactic structure of an algebra's elements</u> [by removing parentheses].

Like the string operation, the binary operation on tuples could be interpreted as juxtaposition, though with interspersed parentheses. For example, a*b = ab and (a*b)*(c*(d*e)) = (ab)(c(de)) would then hold for both strings and tuples; (ab)(c(de)) = abcde, permitted by associativity, would however only hold for strings. [As in (Goguen et al. 1978) the parentheses in expressions like <u>(</u>ab<u>)(</u>c<u>(</u>de<u>))</u> could be underlined to emphasize that they are part of the alphabet from which the "stringified" tuples are constructed.] Later, following the self-denotation principle of initial algebras, the complete infix function call notation -- even trivializing juxtaposition -- will be used as the operation result instead: Thus a*b [here preferably written as *(a,b), to stress that "*" is being called with arguments a and b] will just denote the term "a*b".

An obvious generality advantage originating from the associativity of strings is their "variable length", i.e. the fact that the originally binary operation can now be interpreted as an N-ary operation, where N may vary from call to call [the capitalization of "N" symbolizes this variable arity]. For example, to obtain abcde we can -- instead of (a*b)*(c*(d*e)), a*(b*(c*(d*e))), or any other equivalent nesting of four binary "*"-calls -- write in short a*b*c*d*e, and regard this as a single N=5-ary "*"-call in mixfix notation [corresponding to *(a,b,c,d,e) in prefix notation]. For strings, in general, the parenthesisless form m1*m2*...*mN-1*mN represents the equivalence class of all binary bracketings of m1m2...mN-1mN.

Now the question arises whether tuples, instead of being restricted to pairs [or to fixed-length triples, quadruples, etc.], can also be interpreted as "variable length" operator applications in spite of their lack of associativity. This can in fact be done, although in a

manner differing considerably from the usual one for strings [an unusual one will be implicit in section 4.4]. For such tuples, in general, the parentheses-saving form m1*m2*...*mN*mN+1 is only allowed if mN+1 is a distinguished element, call it "`", and in this case m1*m2*...*mN*` represents the unique right-associating nesting m1*(m2*...*(mN*`)...), and can be interpreted as the N-tuple (m1, m2, ..., mN). For example, neither (a*b)*(c*(d*e)) nor a*(b*(c*(d*e))) could be rewritten into the N-ary parentheses-saving form, but their "`"-terminated expansions (a*(b*`))*(c*(d*(e*`))) and a*(b*(c*(d*(e*`)))) can be abbreviated as (a*b*`)*c*d*e*` and a*b*c*d*e*`, and can be interpreted as the 4-tuple ((a, b), c, d, e) and the 5-tuple (a, b, c, d, e), respectively. A single "listrestriction" axiom will suffice for realizing this variable-length technique with the usual "one-sorted" algebras [giving variable-length N-tuples an axiomatic structure comparable to that of strings], and even this can be dropped with the help of "generator-separated" algebras [leaving the tuples no axiomatic structure at all].

In the programming language LISP such an interpretation of nested pairs as N-tuples was introduced to _implement_ _variable-length_ _lists_ _using_ _fixed-length_ _memory_ _cells_. Curiously, we can not only use it for this very concrete purpose but also for a very abstract one, namely to _formalize_ _variable-arity_ _constructors_ _using_ _fixed-arity_ _algebraic_ _operators_. In fact, the dot of "dotted pairs" in LISP can be regarded as the algebraic multiplication operator "*", used as an infix version of the binary CONS prefix function [thus for LISP lists we have "*" = "." and, of course, "`" = "NIL"]. However, there is a subtle difference between dotted pairs [data structures] and the algebraic infix notation [operator applications]: While in the dotted pair (x.y) the parentheses are a proper syntactic constituent [hence can never be omitted], in the algebraic operation (x*y) or x*y they are only necessary for grouping in sublevels [hence can be omitted from the top-level]. For example, while (a.NIL).(b.NIL) is not a legitimate dotted pair nesting, (a.`).(b.`) is a legitimate nesting of applications of a "."-operator. For mixed collection nestings [such as sets of tuples] we will also deviate from LISP's "list monoculture" by allowing several kinds of CONS-like multiplication operators, together with several associated kinds of NIL; although this representation of multiple collections is clearer algebraically, the LISP/QA4/FIT convention of using the first list element as a tag naming the collection seems at least as apt in practical programming.

In early versions of the PROLOG language, lists were not only represented by dotted pairs, as in LISP, but were actually replaced by dotted-pair-like nestings of "."-terms, which could be abbreviated in the right-associating manner. For example, the LISP list (a b c d e) was replaced by the nesting a.(b.(c.(d.(e.nil)))), which could be abbreviated as a.b.c.d.e.nil. However, while the nested pair view of N-tuples is advantageous for concrete implementation and abstract mathematical/logical formalization, we do not regard it as a very high-level _programming_ _language_ feature (Boley 1983), e.g. because of its asymmetry and its use of a "nil" auxiliary [even in our somewhat further shortened algebraic form a.b.c.d.e.` the end marker "`" -- although single-character -- would be bothersome in day-to-day programming]. Thus, our algebraic formalization of collections should be taken to _justify_ variable-length list notation, not to _replace_ it by [disguised] pair notation.

We now give the formal definition of the constructor algebra ALC01 of N-tuples.

<u>Definition1</u>:

A   generating set [a finite set]
` IN A   distinguished empty element

ALC01 = (M,.)   algebra with

M   carrier generated by A with "."

. : M x M -> M   binary operation
.(m1,m2) = m1.m2

<u>Axiomatic operator restriction</u>:

listrestriction: m1.m2 = `  if m2 <> n.n' and m2 <> `

According to the above definition the generating set A must at least contain the special element "`", interpretable as the empty tuple [0-tuple]. The listrestriction axiom reduces a result of the binary operation "." to "`" if its second argument was neither equal to another product n.n' [a non-empty tuple] nor to "`" [the empty tuple]. Otherwise the "."-result remains a binary term constructed from the two "."-arguments. Assuming the generating set A contains besides "`" the elements a1, ..., aN, the conditional equation for ALC01's listrestriction axiom could be replaced equivalently by the N unconditional equations m.a1 = `, ..., m.aN = `.

The special element "`" -- the only one which must be present in every A-generated algebra -- is thus chosen as the value lr-equal to all "erroneous" "."-terms. Alternatively, this value could be returned directly by "." for an "erroneous" second argument, thus also retaining the total definition of "." on M x M and avoiding the introduction of "partial algebras" [a generally applicable method for this would be introducing another distinguished element (Graetzer 1968), but for our purpose "`" can play the role of both this "error" element and the empty tuple]. However, as discussed in section 2, we prefer the "self-denotation for all arguments" technique of initial algebras, axiomatically equating "erroneous" terms with the distinguished element only afterwards.

If the listrestriction axiom were omitted, the resulting -- even simpler -- algebra would define binary nestings that could be interpreted as general LISP s-expressions. This simpler "anarchic algebra" [obeying no laws] would generate the entire Herbrand universe of all possible terms constructed as "."-nestings over A = {`, a1, ..., aN}, coextensive with the context-free language generated by the BNF rule

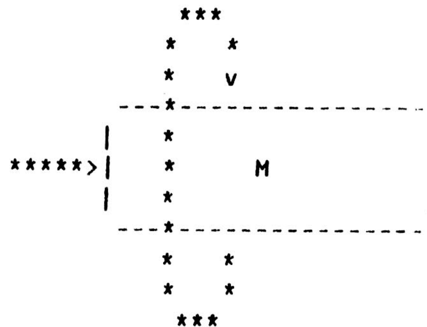
sym-expr ::= "a1" | ... | "aN" | "`" | "(" sym-expr "." sym-expr ")"

Because of the axiomatic restriction of the second "."-argument to what in LISP terminology may be called "NIL and list-representing dotted pairs" [while the first argument may also contain non-NIL atoms], the carrier produced is not the entire Herbrand universe which the generating set originally generates with the binary operator; instead,

the "`"-identification of "erroneous" terms reduces it to the desired subset, which in LISP terminology could be described as "atoms and list-representing dotted pairs", coextensive with the context-free language generated by the BNF rules [the first rule defines our usual version without top-level parentheses]

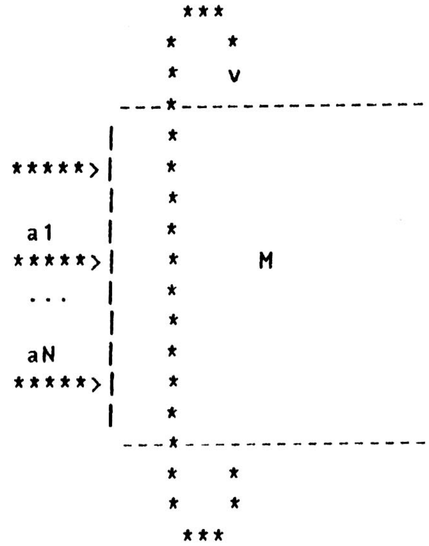
```
top-expr ::= "a1" | ... | "aN" | "`" | syl-expr "." lis-expr
syl-expr ::= "a1" | ... | "aN" | lis-expr
lis-expr ::= "`"  | "(" syl-expr "." lis-expr ")"
```

The carriers and operations of algebras can be depicted in a diagram notation as follows [this is inspired by the use of polyadic graphs for signatures in (Goguen et al. 1978), but uses directed labeled hypergraphs as put into context in section 5.2]: A carrier becomes a node [drawn as a box] bearing the carrier name, and an operation becomes a directed labeled hyperarc [drawn as an arrow] labeled with the operation name and starting at its first argument, cutting its second argument, ..., cutting its last argument, and ending at its value [result]; a distinguished element of a generating set is regarded as a nullary operator, thus becoming a hyperarc labeled with the element name and ending at [pointing to] the corresponding carrier. Since the "homogeneous algebras" considered until now have only one carrier [hence the synonym "one-sorted algebras"], the corresponding directed labeled hypergraphs have only one node also. For example, ALC01 is depicted thus:



The distinguished element "`" has become a directed hyperarc of length 1 [leading from nowhere to the result] and the binary operation "." has become a directed hyperarc of length 3 [leading from the first argument via the second argument to the result].

Like distinguished elements, the remaining elements of a given generating set can be represented as length-1 hyperarcs, but the representation of the indefiniteness of our arbitrary generating set remainders {a1, ..., aN}, as in the second ALC01 depiction,



may impair the readability of diagrams, hence will usually be omitted.

Example:

A = {`, a, b}

The carrier M generated by this set A with the restricted operation "." can be seen as a Herbrand subuniverse, i.e. the limit of the sequence of sets M0 = A, M1 = M0 U {`.`, a.`, b.`}, ..., where for general n>0 Mn = Mn-1 U { x.y | x IN Mn-1 and y IN Mn-1 and (y = n.n' or y = `)}, with the set membership condition (y = n.n' or y = `) embodying the negated listrestriction axiom.

The table below depicts three notations for M3: The "basic binary form" column gives an unabbreviated binary term notation; the "N-ary/paren-sparing" column gives the abbreviated notation corresponding to the N-ary interpretation; the "LISP list" column gives the list notation as used in LISP. A tilde ["~"] in the second and/or third of these columns means that an entry does not differ from the one in the first column of the same row.

The "LISP list" column down to Mn can be obtained as a set-representing list by a call (ENUMLST A n NIL) of the following LISP function ENUMLST [A is a list representing the set A MINUS {`}]:

; (DEN (f a1 ... aN) b) denotes a function f with formals aI and body b

```
(DEN                                            ; enumerate lists
 (ENUMLST ALPH NUM PREV)
 (COND
  ((ZEROP NUM) (CONS NIL (APPEND ALPH PREV)))
  (T (ENUMLST
      ALPH
      (SUB1 NUM)
      (UNIORD
       PREV
       ((LAMBDA (AA)
         (MAPCAN '(LAMBDA (D) (MAPCAR '(LAMBDA (A) (CONS A D)) AA))
                 (CONS NIL PREV)))
        (CONS NIL (APPEND ALPH PREV))))))))

(DEN                                            ; order-preserving UNION
 (UNIORD L1 L2)
 (COND ((NULL L1) L2)
       (T (CONS (CAR L1) (REMOVE (CAR L1) (UNIORD (CDR L1) L2))))))

(DEN                                            ; non-destructive MAPCAN
 (MAPCAN FN L)
 (AND L (APPEND (APPLY FN (LIST (CAR L))) (MAPCAN FN (CDR L)))))
```

no.		basic binary form	N-ary/paren-sparing	LISP list
1	|	`	~	NIL or ()
2	|	a	~	~
3	V	b	~	~
-- M0				
4	|	`.`	~	(NIL)
5	|	a.`	~	(a)
6	V	b.`	~	(b)
-- M1				
7	|	(`.`).`	~	((NIL))
8	|	(a.`).`	~	((a))
9	|	(b.`).`	~	((b))
10	|	`.(`.`)	`.`.`	(NIL NIL)
11	|	a.(`.`)	a.`.`	(a NIL)
12	|	b.(`.`)	b.`.`	(b NIL)
13	|	(`.`).(`.`)	(`.`).`.`	((NIL) NIL)
14	|	(a.`).(`.`)	(a.`).`.`	((a) NIL)
15	|	(b.`).(`.`)	(b.`).`.`	((b) NIL)
16	|	`.(a.`)	`.a.`	(NIL a)
17	|	a.(a.`)	a.a.`	(a a)
18	|	b.(a.`)	b.a.`	(b a)
19	|	(`.`).(a.`)	(`.`).a.`	((NIL) a)
20	|	(a.`).(a.`)	(a.`).a.`	((a) a)
21	|	(b.`).(a.`)	(b.`).a.`	((b) a)
22	|	`.(b.`)	`.b.`	(NIL b)
23	|	a.(b.`)	a.b.`	(a b)
24	|	b.(b.`)	b.b.`	(b b)
25	|	(`.`).(b.`)	(`.`).b.`	((NIL) b)
26	|	(a.`).(b.`)	(a.`).b.`	((a) b)
27	V	(b.`).(b.`)	(b.`).b.`	((b) b)
-- M2				

28	((`.`).`).`	~	(((NIL)))
29	((a.`).`).`	~	(((a)))
30	((b.`).`).`	~	(((b)))
31	(`.(`.`)).`	(`.`.`).`	((NIL NIL))
32	(a.(`.`)).`	(a.`.`).`	((a NIL))
33	(b.(`.`)).`	(b.`.`).`	((b NIL))
34	((`.`).(`.`)).`	((`.`).`.`).`	(((NIL)NIL))
35	((a.`).(`.`)).`	((a.`).`.`).`	(((a) NIL))
36	((b.`).(`.`)).`	((b.`).`.`).`	(((b) NIL))
37	(`.(a.`)).`	(`.a.`).`	((NIL a))
38	(a.(a.`)).`	(a.a.`).`	((a a))
39	(b.(a.`)).`	(b.a.`).`	((b a))
40	((`.`).(a.`)).`	((`.`).a.`).`	(((NIL) a))
41	((a.`).(a.`)).`	((a.`).a.`).`	(((a) a))
42	((b.`).(a.`)).`	((b.`).a.`).`	(((b) a))
43	(`.(b.`)).`	(`.b.`).`	((NIL b))
44	(a.(b.`)).`	(a.b.`).`	((a b))
45	(b.(b.`)).`	(b.b.`).`	((b b))
46	((`.`).(b.`)).`	((`.`).b.`).`	(((NIL) b))
47	((a.`).(b.`)).`	((a.`).b.`).`	(((a) b))
48	((b.`).(b.`)).`	((b.`).b.`).`	(((b) b))
...	. . .	. . .	. . .
111	((b.`).(b.`)).(b.`)	((b.`).b.`).b.`	(((b) b) b)
112	`.((`.`).`)	`.(`.`).`	(NIL (NIL))
113	a.((`.`).`)	a.(`.`).`	(a (NIL))
114	b.((`.`).`)	b.(`.`).`	(b (NIL))
...	. . .	. . .	. . .
139	`.((a.`).`)	`.(a.`).`	(NIL (a))
140	a.((a.`).`)	a.(a.`).`	(a (a))
141	b.((a.`).`)	b.(a.`).`	(b (a))
...	. . .	. . .	. . .
166	`.((b.`).`)	`.(b.`).`	(NIL (b))
167	a.((b.`).`)	a.(b.`).`	(a (b))
168	b.((b.`).`)	b.(b.`).`	(b (b))
...	. . .	. . .	. . .
193	`.(`.(`.`))	`.`.`.`	(NIL () NIL)
194	a.(`.(`.`))	a.`.`.`	(a NIL NIL)
195	b.(`.(`.`))	b.`.`.`	(b NIL NIL)
...	. . .	. . .	. . .
220	`.(a.(`.`))	`.a.`.`	(NIL a NIL)
221	a.(a.(`.`))	a.a.`.`	(a a NIL)
222	b.(a.(`.`))	b.a.`.`	(b a NIL)
...	. . .	. . .	. . .
247	`.(b.(`.`))	`.b.`.`	(NIL b NIL)
248	a.(b.(`.`))	a.b.`.`	(a b NIL)
249	b.(b.(`.`))	b.b.`.`	(b b NIL)
...	. . .	. . .	. . .
355	`.(`.(a.`))	`.`.a.`	(NIL NIL a)
356	a.(`.(a.`))	a.`.a.`	(a NIL a)
357	b.(`.(a.`))	b.`.a.`	(b NIL a)
...	. . .	. . .	. . .
382	`.(a.(a.`))	`.a.a.`	(NIL a a)
383	a.(a.(a.`))	a.a.a.`	(a a a)
384	b.(a.(a.`))	b.a.a.`	(b a a)
...	. . .	. . .	. . .
409	`.(b.(a.`))	`.b.a.`	(NIL b a)
410	a.(b.(a.`))	a.b.a.`	(a b a)
411	b.(b.(a.`))	b.b.a.`	(b b a)

```
... | . . .                           | . . .                | . . .
516 | ((b.`).(b.`)).((b.`).(a.`))     | ((b.`).b.`).(b.`).a.`| (((b)b)(b)a)
517 | `.(`.(b.`))                     | `.`.b.`              | (NIL NIL b)
518 | a.(`.(b.`))                     | a.`.b.`              | (a NIL b)
519 | b.(`.(b.`))                     | b.`.b.`              | (b NIL b)
... | . . .                           | . . .                | . . .
544 | `.(a.(b.`))                     | `.a.b.`              | (NIL a b)
545 | a.(a.(b.`))                     | a.a.b.`              | (a a b)
546 | b.(a.(b.`))                     | b.a.b.`              | (b a b)
... | . . .                           | . . .                | . . .
571 | `.(b.(b.`))                     | `.b.b.`              | (NIL b b)
572 | a.(b.(b.`))                     | a.b.b.`              | (a b b)
573 | b.(b.(b.`))                     | b.b.b.`              | (b b b)
... | . . .                           | . . .                | . . .
678 V ((b.`).(b.`)).((b.`).(b.`))     | ((b.`).b.`).(b.`).b.`| (((b)b)(b)b)
-- M3 --------------------------------+----------------------+-------------
... | . . .                           | . . .                | . . .
```

The following Equality relation [like EQUAL in LISP] recursively reduces the equality of entire applications of the "."-operator [in general, of binary multiplication operators] to the equality of its arguments, until it reaches elements of A, for which it can decide equality in a trivial manner.

```
Equality:  m1 = m2   obvious if m1 IN A or m2 IN A
           m1 = m.m' = m''.m''' = m2   <=>   m = m'' and m' = m'''
```

Example:

```
(a.`).(b.`) = (a.`).(b.`)
(a.`) = (a.`)  and  (b.`) = (b.`)
a = a and ` = `  and  b = b and ` = `
```

As a special kind of the "many-sorted algebras" which -- starting in section 3.2 -- will complement our usual "one-sorted algebras", here we introduce "generator-separated algebras". In these the [always finite] "generator" -- as we shall call the generating set minus possible distinguished elements -- becomes a carrier of its own, separated from the [usually infinite] "generated" carrier. This will permit the operations to map between the two carriers in such a way that certain operator restriction axioms become unnecessary.

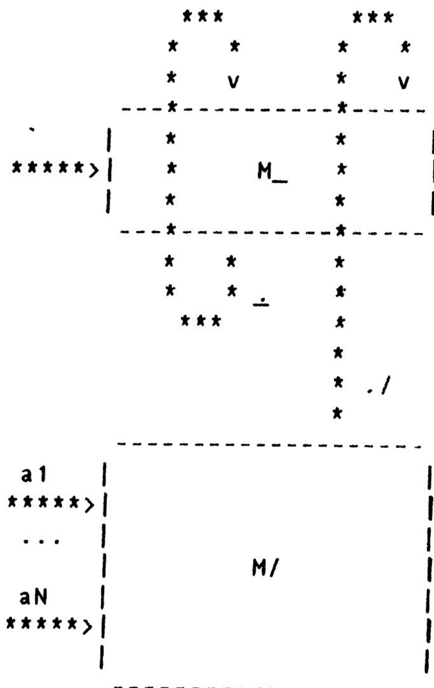
The one-sorted algebra ALC01 thereby becomes a generator-separated algebra ALC01$ [generator-separated versions of one-sorted algebras and definitions will be suffixed with a "$"-mark], where the ALC01 carrier M of atoms and lists is divided into a carrier M/ of non-"`" atoms and a carrier M_ of lists [incl. "`"]. At the same time the CONS operation "." is divided into an operation "./" for constructing atoms onto lists and an operation "._" for constructing lists onto lists; since no terms are constructed onto non-"`" atoms, the listrestriction axiom thus becomes unnecessary, leaving ALC01$ as an "anarchic" algebra [in which no axioms are postulated]. Besides this axiom elimination, the separation of the generator has the further advantage that the set of lists can now be introduced without carrying along as a subset the set of their constituent non-"`" atoms.

<u>Definition1$</u>:

A_	generating set [a singleton set]
` IN A_	distinguished empty element
A/	generating set [a generator]
ALCO1$ = (M_,M/;._,./)	algebra with
M_, M/	carriers generated by A_, A/ with "._", "./" [M/ = A/]
._ : M_ x M_ -> M_	binary operation [consing lists to lists]
._(m1,m2) = m1._m2	[m1 IN M_ and m2 IN M_]
./ : M/ x M_ -> M_	binary operation [consing non-"`" atoms to lists]
./(m1,m2) = m1./m2	[m1 IN M/ and m2 IN M_]

The listrestriction axiom has become superfluous, because the second arguments of both "."-derivates, "._" and "./", must be elements of M_, hence either (n.n')-terms or "`".

The directed labeled hypergraphs corresponding to generator-separated algebras have a separate node for the generator carrier and the usual node for the generated carrier. For example, ALCO1$ can be depicted thus [the elements of the generator A/ -- call them a1, ..., aN -- are included as hyperarcs, because their role is essential here]:



Comparing this diagram with the second diagram for ALCO1, it can be seen that the original carrier M has become separated into a "given" carrier M/ and a "derived" carrier M_; in the latter only lists of "`"-elements could be built, were it not initialized with the non-"`" atoms from M/ [notice that, despite the geometric proportions, M/ is

finite and M_ infinite]. It is likewise possible to see how the original (MxM)-operation "." has become separated into an initializing (M/xM_)-operation "./" and a "."-like (M_xM_)-operation "⊥".

Example:

A_ = {`}
A/ = {a, b}

The two carriers M_ and M/ generated by these sets A_ and A/ with the unrestricted operations "⊥" and "./" can be seen as heterogeneous Herbrand subuniverses, i.e. the limits of the sequences of sets M_0 = A_, M_1 = M_0 U {`⊥`, a./`, b./`}, ... and M/0 = A/, M/1 = M/0, ..., where for general n>0 M_n = M_(n-1) U { x⊥y | x IN M_(n-1) and y IN M_(n-1)} U { x./y | x IN M/(n-1) and y IN M_(n-1)} and M/n = M/(n-1) [=A/].

The following tables depict three notations for M_2 and M/2. It is worthwhile to compare the operators of "`"-list terms like no. 11 with those of flat-list terms like no. 16 in M_2: Lists of "`"-elements are the ones that can be built using only the "⊥"-part of ".", while lists without sublists result from using only the "./"-part of ".".

no.	basic binary form	N-ary/paren-sparing	LISP list
1	`	~	NIL or ()
-- M_0			
2	`⊥`	~	(NIL)
3	a./`	~	(a)
4	V b./`	~	(b)
-- M_1			
5	(`⊥`)⊥`	~	((NIL))
6	(a./`)⊥`	~	((a))
7	(b./`)⊥`	~	((b))
8	`⊥(`⊥`)	`⊥`⊥`	(NIL NIL)
9	a./(`⊥`)	a./`⊥`	(a NIL)
10	b./(`⊥`)	b./`⊥`	(b NIL)
11	(`⊥`)⊥(`⊥`)	(`⊥`)⊥`⊥`	((NIL) NIL)
12	(a./`)⊥(`⊥`)	(a./`)⊥`⊥`	((a) NIL)
13	(b./`)⊥(`⊥`)	(b./`)⊥`⊥`	((b) NIL)
14	`⊥(a./`)	`⊥a./`	(NIL a)
15	a./(a./`)	a./a./`	(a a)
16	b./(a./`)	b./a./`	(b a)
17	(`⊥`)⊥(a./`)	(`⊥`)⊥a./`	((NIL) a)
18	(a./`)⊥(a./`)	(a./`)⊥a./`	((a) a)
19	(b./`)⊥(a./`)	(b./`)⊥a./`	((b) a)
20	`⊥(b./`)	`⊥b./`	(NIL b)
21	a./(b./`)	a./b./`	(a b)
22	b./(b./`)	b./b./`	(b b)
23	(`⊥`)⊥(b./`)	(`⊥`)⊥b./`	((NIL) b)
24	(a./`)⊥(b./`)	(a./`)⊥b./`	((a) b)
25	V (b./`)⊥(b./`)	(b./`)⊥b./`	((b) b)
-- M_2			
...	. . .	. . .	. . .

```
no. | basic binary form              | N-ary/paren-sparing  | LISP list
----+--------------------------------+----------------------+-------------
1   | a                              | ~                    | ~
2   V b                              | ~                    | ~
-- M/0 --------------------------------+----------------------+-------------
-- M/1 --------------------------------+----------------------+-------------
-- M/2 --------------------------------+----------------------+-------------
-- ... --------------------------------+----------------------+-------------
```

In definition1 the finite cardinality of the generating set A also restricts the number of atoms over which N-tuples are built to the same finite number. Although this is not a limitation in practice [let |A| = 64^1 + 64^2 + ... + 64^132 equal the number of different atoms that can be composed from a character set like {"A", ..., "Z", "a", ..., "z", "0", ..., "9", "-", ":"} in, say, a lineprinter line], nor in theory [every |A| can be augmented to |A| + 1], the use of large numbers of individual "symbols" instead of symbol-composed "words" may appear unsatisfactory. The following two-level construction, as implicit in LISP, may be preferred: First, a [countably] infinite number of atoms is composed from a [small] finite set of symbols in the form of symbol strings. Second, the N-tuples are constructed from these string atoms. This combination of strings and tuples can be formalized by the following definition of the constructor algebra ALC01'.

Definition1':

```
A                    generating set [a finite set]
` IN A               distinguished empty element

ALC01' = (M,|,.)     algebra with

M                    carrier generated by A with "|", "."

| : M x M -> M       binary operation [constructing strings]
|(m1,m2) = m1|m2

. : M x M -> M       binary operation [constructing tuples]
.(m1,m2) = m1.m2
```

Axiomatic operator restriction:

```
|wordrestriction: m1|m2 = `  if  (m1 <> n|n' and m1 NOTIN A MINUS {`})
                              or (m2 <> n|n' and m2 NOTIN A MINUS {`})

.listrestriction: m1.m2 = `  if m2 <> n.n' and m2 <> `
```

Of course, associativity is also postulated for the "|"-operator.

```
|associativity:  (m1|m2)|m3 = m1|(m2|m3)
```

[If an axiom could be postulated for various operators, its postulation for a given operator may be disambiguated by prefixing the axiom name with that operator.]

The wordrestriction axiom prevents the illegal construction of "atoms" with list components [inverse to the legal construction of lists from

atoms] by identifying "erroneous" terms like a|` and (a.b.c.`)|b|d|a|(e.d.`)|b [in LISP-like juxtaposition syntax a` and (a.b.c.`)bda(e.d.`)b] with "`". The w axiom can be transformed to four unconditional equations, thus illustrating the elimination of the NOTIN predicate from conditional equations [cf. section 2]:

```
Or-splitting:
m1|m2 = `  if  m1 <> n|n' and m1 NOTIN A MINUS {`}
m1|m2 = `  if  m2 <> n|n' and m2 NOTIN A MINUS {`}

NOTIN removal:
m1|m2 = `  if  m1 <> n|n' and (m1 = n.n' or m1 = n|n' or m1 = `)
m1|m2 = `  if  m2 <> n|n' and (m2 = n.n' or m2 = n|n' or m2 = `)

And/or distribution:
m1|m2 = `  if  (m1 <> n|n' and m1 = n.n')
            or (m1 <> n|n' and m1 = n|n')
            or (m1 <> n|n' and m1 = `)
m1|m2 = `  if  (m2 <> n|n' and m2 = n.n')
            or (m2 <> n|n' and m2 = n|n')
            or (m2 <> n|n' and m2 = `)

Contradiction law:
m1|m2 = `  if  (m1 <> n|n' and m1 = n.n')
            or FALSE
            or (m1 <> n|n' and m1 = `)
m1|m2 = `  if  (m2 <> n|n' and m2 = n.n')
            or FALSE
            or (m2 <> n|n' and m2 = `)

FALSE neutral or element:
m1|m2 = `  if  (m1 <> n|n' and m1 = n.n') or (m1 <> n|n' and m1 = `)
m1|m2 = `  if  (m2 <> n|n' and m2 = n.n') or (m2 <> n|n' and m2 = `)

And commutativity:
m1|m2 = `  if  (m1 = n.n' and m1 <> n|n') or (m1 = ` and m1 <> n|n')
m1|m2 = `  if  (m2 = n.n' and m2 <> n|n') or (m2 = ` and m2 <> n|n')

<> removal [incl. or associativity]:
m1|m2 = `  if  (m1=n.n' and (m1=n.n' or (m1=` or m1 IN A MINUS {`})))
            or (m1=` and (m1=` or (m1=n.n' or m1 IN A MINUS {`})))
m1|m2 = `  if  (m2=n.n' and (m2=n.n' or (m2=` or m2 IN A MINUS {`})))
            or (m2=` and (m2=` or (m2=n.n' or m2 IN A MINUS {`})))

And/or absorption:
m1|m2 = `  if  m1 = n.n' or m1 = `
m1|m2 = `  if  m2 = n.n' or m2 = `

Or-splitting:
m1|m2 = `  if  m1 = n.n'
m1|m2 = `  if  m1 = `
m1|m2 = `  if  m2 = n.n'
m1|m2 = `  if  m2 = `
```

```
Substitution:
(n.n')|m2 = `
     `|m2 = `
m1|(n.n') = `
m1|`      = `
```

In the following we will not use the primed version of definition1, so that we can avoid another level of complexity that is not essential in our subsequent development. Should the unprimed versions of later definitions have to be replaced by primed versions, conditional equations referring to the set A would have to be modified so as to refer to both A and "|"-terms. As an alternative to the direct introduction of string atoms in definition1', it would also be possible to use the strings defined as one of the basic collections in section 4.4, preventing the nesting of tuples into strings by a variant of the above wordrestriction axiom.

3.2 A List Algebra with Negated Elements

In order to accomodate the algebra ALCO1 of N-tuples to the generalized Idempotence to be introduced in section 4.2, we augment it here into an algebra ALCO2 with an additional [auxiliary] unary operation, "-", leaving the three Remove axioms governing its behavior for that section. This new list algebra will permit "negative" elements [which, however, should not be confused with ordinary negation, as used in section 2], produced by non-negative elements to remove other occurrences of themselves. Since terms with a top-level "-"-sign [with "-" as their "principal" operator] will not be reducible to "-"-less terms [see below table], in the ADT view the "-"-auxiliary can be regarded as a "hidden" or "private" operation.

Definition2:

```
A                   generating set [a finite set]
` IN A              distinguished empty element

ALCO2 = (M,.,-)     algebra with

M                   carrier generated by A with ".", "-"

. : M x M -> M      binary operation
.(m1,m2) = m1.m2

- : M -> M          unary operation
-(m) = -m
```

Axiomatic operator restriction:

listrestriction: m1.m2 = ` if m2 <> n.n' and m2 <> `

Although this listrestriction axiom consists of the same conditional equation as the one postulated for ALCO1, in ALCO2 it is applicable additionally to terms with "-"-signed second arguments, so that besides the equations m.aI = ` [1 ≤ I ≤ N] one further unconditional equation, m1.-m2 = `, would be required for making ALCO2's listrestriction unconditional.

In our diagram notation ALCO2 can be depicted thus:

```
                 ***
                *   *
                *   v
             ---*-----------------
            |   *                 |**********
      *****>|   *       M         |          *
            |   *                 |<*********
             ---*-----------------
                *   *
                *   *
                 ***
```

The new unary operation "-" has become a directed hyperarc of length 2 [leading from the single argument to the result].

Example:

A = {`, a, b}

As in section 3, the following table again depicts three notations for M3. The additional column gives Remove reductions [explained in section 4.2] of terms which are "."-products with "-"-signed left factors to terms without these negative signs.

no.	basic binary form	N-ary/paren-sp.	LISP list	reduction
1	`	~	NIL or ()	~
2	a	~	~	~
3 V	b	~	~	~
--- M0				
4	`.`	~	(NIL)	~
5	a.`	~	(a)	~
6	b.`	~	(b)	~
7	-`	~	-NIL or -()	~
8	-a	~	~	~
9 V	-b	~	~	~
--- M1				
10	(`.`).`	~	((NIL))	~
11	(a.`).`	~	((a))	~
12	(b.`).`	~	((b))	~
13	-`.`	~	(-NIL)	`
14	-a.`	~	(-a)	`
15	-b.`	~	(-b)	`
16	`.(`.`)	`.`.`	(NIL NIL)	~
17	a.(`.`)	a.`.`	(a NIL)	~
18	b.(`.`)	b.`.`	(b NIL)	~
...	. . .	. . . .	. . .	. . .
22	-`.(`.`)	-`.`.`	(-NIL NIL)	`
23	-a.(`.`)	-a.`.`	(-a NIL)	`.`
24	-b.(`.`)	-b.`.`	(-b NIL)	`.`
25	`.(a.`)	`.a.`	(NIL a)	~
26	a.(a.`)	a.a.`	(a a)	~
27	b.(a.`)	b.a.`	(b a)	~
...	. . .	. . . .	. . .	. . .
31	-`.(a.`)	-`.a.`	(-NIL a)	a.`
32	-a.(a.`)	-a.a.`	(-a a)	`

```
33    | -b.(a.`)            | -b.a.`         | (-b a)        | a.`
34    | `.(b.`)             | `.b.`          | (NIL b)       | ~
35    | a.(b.`)             | a.b.`          | (a b)         | ~
36    | b.(b.`)             | b.b.`          | (b b)         | ~
...   | . . .               | . . .          | . . .         | . . .
40    | -`.(b.`)            | -`.b.`         | (-NIL b)      | b.`
41    | -a.(b.`)            | -a.b.`         | (-a b)        | b.`
42    | -b.(b.`)            | -b.b.`         | (-b b)        | `
43    | -(`.`)              | ~              | -(NIL)        | ~
44    | -(a.`)              | ~              | -(a)          | ~
45    | -(b.`)              | ~              | -(b)          | ~
46    | --`                 | ~              | --NIL         | ~
47    | --a                 | ~              | ~             | ~
48    V --b                 | ~              | ~             | ~
--- M2 ---------------------+----------------+---------------+-----------
...   | . . .               | . . .          | . . .         | . . .
52    | (-`.`).`            | ~              | ((-NIL))      | `.`
53    | (-a.`).`            | ~              | ((-a))        | `.`
54    | (-b.`).`            | ~              | ((-b))        | `.`
...   | . . .               | . . .          | . . .         | . . .
82    | -(`.`).`            | ~              | (-(NIL))      | ~
83    | -(a.`).`            | ~              | (-(a))        | ~
84    | -(b.`).`            | ~              | (-(b))        | ~
85    | --`.`               | ~              | (--NIL)       | ~
86    | --a.`               | ~              | (--a)         | ~
87    | --b.`               | ~              | (--b)         | ~
...   | . . .               | . . .          | . . .         | . . .
349   | `.(-`.`)            | `.-`.`         | (NIL -NIL)    | `.`
350   | a.(-`.`)            | a.-`.`         | (a -NIL)      | a.`
351   | b.(-`.`)            | b.-`.`         | (b -NIL)      | b.`
...   | . . .               | . . .          | . . .         | . . .
355   | -`.(-`.`)           | -`.-`.`        | (-NIL -NIL)   | `
356   | -a.(-`.`)           | -a.-`.`        | (-a -NIL)     | `
357   | -b.(-`.`)           | -b.-`.`        | (-b -NIL)     | `
...   | . . .               | . . .          | . . .         | . . .
397   | `.(-a.`)            | `.-a.`         | (NIL -a)      | `.`
398   | a.(-a.`)            | a.-a.`         | (a -a)        | a.`
399   | b.(-a.`)            | b.-a.`         | (b -a)        | b.`
...   | . . .               | . . .          | . . .         | . . .
403   | -`.(-a.`)           | -`.-a.`        | (-NIL -a)     | `
404   | -a.(-a.`)           | -a.-a.`        | (-a -a)       | `
405   | -b.(-a.`)           | -b.-a.`        | (-b -a)       | `
...   | . . .               | . . .          | . . .         | . . .
445   | `.(-b.`)            | `.-b.`         | (NIL -b)      | `.`
446   | a.(-b.`)            | a.-b.`         | (a -b)        | a.`
447   | b.(-b.`)            | b.-b.`         | (b -b)        | b.`
...   | . . .               | . . .          | . . .         | . . .
451   | -`.(-b.`)           | -`.-b.`        | (-NIL -b)     | `
452   | -a.(-b.`)           | -a.-b.`        | (-a -b)       | `
453   | -b.(-b.`)           | -b.-b.`        | (-b -b)       | `
...   | . . .               | . . .          | . . .         | . . .
1792  | -(-`.`)             | ~              | -(-NIL)       | -`
1793  | -(-a.`)             | ~              | -(-a)         | -`
1794  | -(-b.`)             | ~              | -(-b)         | -`
...   | . . .               | . . .          | . . .         | . . .
1827  V ---b                | ~              | ~             | ~
--- M3 ---------------------+----------------+---------------+-----------
...   | . . .               | . . .          | . . .         | . . .
```

Instead of generating terms with a principal "."-operator and with a principal "-"-operator as elements of the same set [carrier], it is also possible to divide these "sorts" of terms into two sets [carriers], thus generalizing the generator-separated algebras of section 3.1 and definitively proceeding from homogeneous or one-sorted algebras to heterogeneous or many-sorted algebras as developed in (Higgins 1963), (Birkhoff & Lipson 1970), and (Goguen et al. 1978). For our constructor algebras the general advantage of heterogeneity is the possibility of putting terms with auxiliary top-level constructors [e.g. the negated terms above] and terms which are only relevant as subterms [e.g. the non-"`" atoms in section 3.1 and the arcs and hyperarcs in section 5] into carriers distinct from the carrier containing the terms with the "interesting" top-level constructors; also some operator restriction axioms [e.g. listrestriction in ALCO1$ of section 3.1] and conditions on axioms become superfluous through the heterogeneous "sort structure" [the elimination of negated conditions is important for guaranteeing initiality]. On the other hand, the differentiation of several carriers calls for a differentiation of the operations [e.g. "⊥" and "./" in ALCO1$], which sometimes may appear artificial and furthermore necessitates a corresponding increase of axioms to be postulated for obtaining quotient algebras.

Our heterogeneous version ALCO2~ of ALCO2 will be two-sorted, differentiating the original carrier M into a carrier M. for products and a carrier M- for negated terms. [Heterogeneous versions of homogeneous algebras, definitions, axioms, and lemmas will be suffixed with a "~"-mark; names of carriers -- and their generating sets -- will be suffixed with the name of the principal operation for constructing their element terms.] This gives rise to a differentiation of the original multiplication operation "." into an operation ".'" for multiplying unnegated with unnegated terms and an operation ".''" for multiplying negated with unnegated terms. [Operation names necessitated by carrier partitionings will be primed versions of the original operation names.]

Definition2~:

A.	generating set [a finite set]
` IN A.	distinguished empty element
A- = {}	generating set [the empty set]
ALCO2~ = (M.,M-;.',.'',-)	algebra with
M., M-	carriers generated by A., A- with ".'", ".''", "-"
.' : M. x M. -> M. .'(m1,m2) = m1.'m2	binary operation [unnegated multiplication] [m1 IN M. and m2 IN M.]
.'' : M- x M. -> M. .''(m1,m2) = m1.''m2	binary operation [negated multiplication] [m1 IN M- and m2 IN M.]
- : M. -> M- -(m) = -m	unary operation [m IN M.]

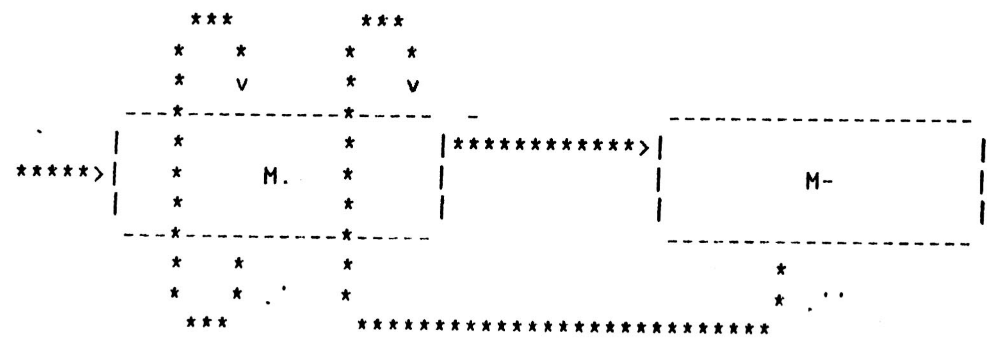
Axiomatic operator restriction:

.'listrestriction~: m1.'m2 = ` if m2 IN A. MINUS {`}

.''listrestriction~: m1.''m2 = ` if m2 IN A. MINUS {`}

The conditions of the listrestricition~ axioms exploit the fact that m2 must denote a member of M. [rather than of M-], due to the "sorted" definitions of ".'" and ".''", so that the only remaining possibility of "erroneous" products is m2 being a member of the generating subset A. minus the distinguished element "`"; without using this space-saving fact, the axioms would become the rather lengthy conditional equations
.'lr~: m1.'m2 = ` if m2 <> n.'n' and m2 <> n.''n' and m2 <> ` and
.''lr~: m1.''m2 = ` if m2 <> n.'n' and m2 <> n.''n' and m2 <> `.

The directed labeled hypergraphs corresponding to heterogeneous algebras have as many nodes as there are carriers. For example, ALCO2~ is depicted thus:



This diagram can be regarded as an expansion of the ALCO2 diagram with the node M divided into the nodes M. and M- and the "."-arrow divided correspondingly into the ".'"-arrow and the ".''"-arrow. Notice that the hyperarc notation clearly indicates the ordering of the arguments, independently from geometric layout [the ".''"-operation has M- as its first argument and M. as its second], thus avoiding a problem with

polyadic edges in (Goguen et al. 1978). [A disadvantage of hyperarcs is the smaller degree to which their last element [value] is set off from the other ones [arguments] pictorially; however, this will turn into an advantage in general many-sorted relational structures, which do not have an argument/value distinction.]

__Example__:

```
A. = {`, a, b}
A- = {}
```

The two carriers M. and M- generated by these sets A. and A- with the restricted operations ".'" and ".''" can be seen as heterogeneous Herbrand subuniverses, i.e. the limits of the sequences of sets M.0 = A., M.1 = M.0 U {`.'`, a.'`, b.'`}, ... and M-0 = A-, M-1 = M-0 U {-`, -a, -b}, ..., where for general n>0

```
M.n = M.(n-1) U { x.'y | x IN M.(n-1) and y IN M.(n-1)
                                  and (y = n.'n' or y = n.''n' or y = `)}
                U { x.''y | x IN M-(n-1) and y IN M.(n-1)
                                  and (y = n.'n' or y = n.''n' or y = `)}
```

and

```
M-n = M-(n-1) U {-x | x IN M.(n-1)}
```

with the set membership conditions (y = n.'n' or y = n.''n' or y = `) embodying the negated listrestriction axiom.

The following tables depict three notations and a Remove reduction for M.2 and M-2. Note that --`, --a, and --b do not appear in M-2 because multiple negations are excluded in ALCO2~ by the fact that "-" maps from M. to M- only, not from M- to M-.

```
no.   | basic binary form | N-ary/paren-sp.  | LISP list        | reduction
------+-------------------+------------------+------------------+----------
1     | `                 | ~                | NIL or ()        | ~
2     | a                 | ~                | ~                | ~
3     V b                 | ~                | ~                | ~
--- M.0 ------------------+------------------+------------------+----------
4     | `.'`              | ~                | (NIL)            | ~
5     | a.'`              | ~                | (a)              | ~
6     V b.'`              | ~                | (b)              | ~
--- M.1 ------------------+------------------+------------------+----------
7     | (`.'`).'`         | ~                | ((NIL))          | ~
8     | (a.'`).'`         | ~                | ((a))            | ~
9     | (b.'`).'`         | ~                | ((b))            | ~
10    | -`.''`            | ~                | (-NIL)           | `
11    | -a.''`            | ~                | (-a)             | `
12    | -b.''`            | ~                | (-b)             | `
13    | `.'(`.'`)         | `.'`.'`          | (NIL NIL)        | ~
14    | a.'(`.'`)         | a.'`.'`          | (a NIL)          | ~
15    | b.'(`.'`)         | b.'`.'`          | (b NIL)          | ~
...   | . . .             | . . .            | . . .            | . . .
19    | -`.''(`.'`)       | -`.''`.'`        | (-NIL NIL)       | `
20    | -a.''(`.'`)       | -a.''`.'`        | (-a NIL)         | `.'`
21    | -b.''(`.'`)       | -b.''`.'`        | (-b NIL)         | `.'`
22    | `.'(a.'`)         | `.'a.'`          | (NIL a)          | ~
23    | a.'(a.'`)         | a.'a.'`          | (a a)            | ~
24    | b.'(a.'`)         | b.'a.'`          | (b a)            | ~
...   | . . .             | . . .            | . . .            | . . .
28    | -`.''(a.'`)       | -`.''a.'`        | (-NIL a)         | a.'`
29    | -a.''(a.'`)       | -a.''a.'`        | (-a a)           | `
30    | -b.''(a.'`)       | -b.''a.'`        | (-b a)           | a.'`
31    | `.'(b.'`)         | `.'b.'`          | (NIL b)          | ~
32    | a.'(b.'`)         | a.'b.'`          | (a b)            | ~
33    | b.'(b.'`)         | b.'b.'`          | (b b)            | ~
...   | . . .             | . . .            | . . .            | . . .
37    | -`.''(b.'`)       | -`.''b.'`        | (-NIL b)         | b.'`
38    | -a.''(b.'`)       | -a.''b.'`        | (-a b)           | b.'`
39    V -b.''(b.'`)       | -b.''b.'`        | (-b b)           | `
--- M.2 ------------------+------------------+------------------+----------
...   | . . .             | . . .            | . . .            | . . .
```

```
no.   | basic binary form | N-ary/paren-sp.  | LISP list        | reduction
------+-------------------+------------------+------------------+----------
0     V                   |                  |                  |
--- M-0 ------------------+------------------+------------------+----------
1     | -`                | ~                | -NIL or -()      | ~
2     | -a                | ~                | ~                | ~
3     V -b                | ~                | ~                | ~
--- M-1 ------------------+------------------+------------------+----------
4     | -(`.'`)           | ~                | -(NIL)           | ~
5     | -(a.'`)           | ~                | -(a)             | ~
6     V -(b.'`)           | ~                | -(b)             | ~
--- M-2 ------------------+------------------+------------------+----------
...   | . . .             | . . .            | . . .            | . . .
```

Many-sorted algebras as considered here can be combined with the generator-separated algebras as introduced in section 3.1, by separating the generators of certain sorts from their generated carriers. For example, the carrier M. of the many-sorted algebra ALCO2˜ can be separated into the generator carrier M/ and the generated carrier M_, leading to a many-sorted generator-separated algebra ALCO2˜$, much like the division of the carrier M of the one-sorted algebra ALCO1 led to the generator-separated algebra ALCO1$. This, in turn, enforces not only a division of the operator ".'" into ".̲" and "./", corresponding to the division of ".", but also a division of "-" into a negation operator "-/" for elements in M/ and a negation operator "-̲" for elements in M_.

<u>Definition2˜$</u>:

A_ — generating set [a singleton set]
` IN A_ — distinguished empty element

A/ — generating set [a generator]

A- = {} — generating set [the empty set]

ALCO2˜$ = (M_,M/,M-;.̲,./,.'',-̲,-/) algebra with

M_, M/, M- — carriers generated by A_, A/, A- with ".̲", "./", ".''", "-̲", "-/" [M/ = A/]

.̲ : M_ x M_ -> M_ — binary operation [consing lists to lists]
.̲(m1,m2) = m1.̲m2 — [m1 IN M_ and m2 IN M_]

./ : M/ x M_ -> M_ — binary operation [consing non-"`" atoms to lists]
./(m1,m2) = m1./m2 — [m1 IN M/ and m2 IN M_]

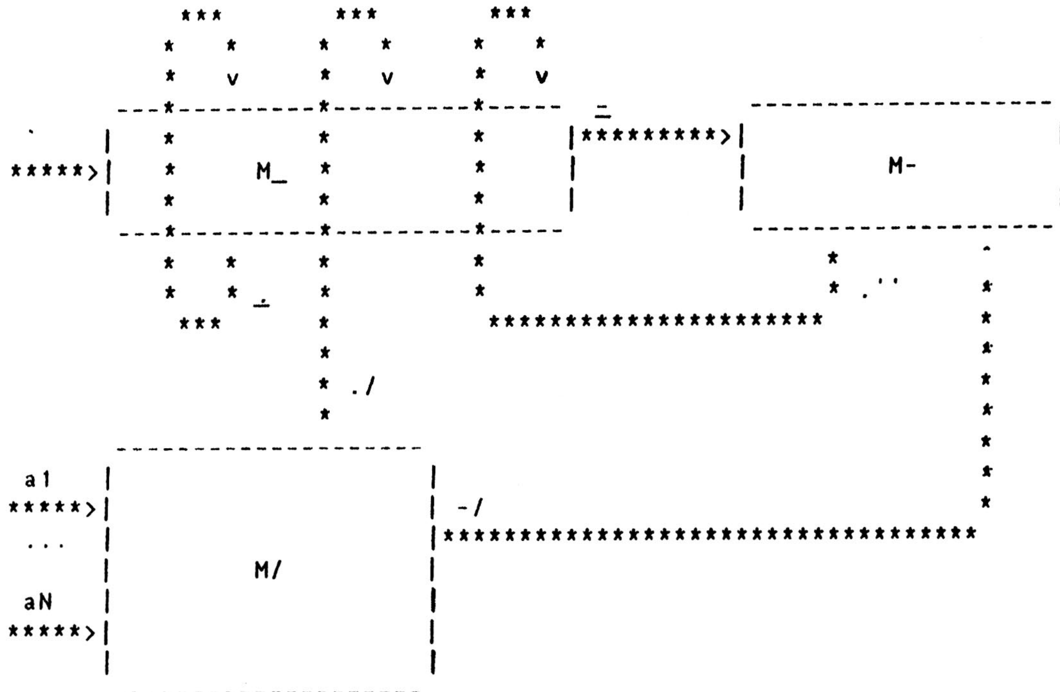
.'' : M- x M_ -> M_ — binary operation [consing negatives to lists]
.''(m1,m2) = m1.''m2 — [m1 IN M- and m2 IN M_]

-̲ : M_ -> M- — unary operation [for negative lists]
-̲(m) = -̲m — [m IN M_]

-/ : M/ -> M- — unary operation [for negative non-"`" atoms]
-/(m) = -/m — [m IN M/]

The listrestriction˜ axioms have become superfluous, because the second arguments of both ".'"-derivates, ".̲" and "./", as well as the second argument of ".''" cannot lie in M/ = A/ but must be elements of M_ [as illustrated by the three congruent "inverse-j"-like arrow parts below].

As a diagram ALCO2˜$ can be depicted thus [assuming the generator is A/ = {a1, ..., aN}]:



Sample terms from M_2 corresponding to nos. 20 and 38 in M.2 of the ALCO2~ example are, respectively, the basic binary forms -/a.''(`⊥`) and -/a.''(b./`) and their N-ary/paren-sparing short forms, -/a.''`⊥` and -/a.''b./`.

A version of the meta-level Equality relation introduced in section 3.1 can also be formalized as an object-level algebraic operator, "=". This could have already be done for the "-"-less list algebras in that section but is introduced here for ALCO2~$, because the application of "=" in section 4.2 will additionally call for negative elements. Instead of the operator "=" for the generator-separated algebra ALCO2~$, the usual "generic" equality operator could be defined, even more easily, for the non-generator-separated algebra ALCO2~, permitting the comparison of both atoms and lists; again, however, the intended application of "=" is in ALCO2~$-based algebras, because this operator will conclude a development largely completed with generator separation already -- the elimination of conditional axioms.

To reinterpret the equality predicate [relation] as an operator, "=", the heterogeneous algebra ALCO2~$ is first extended by the boolean carrier Mb = {T, F} to the algebra ALCO2~$b. This Mb is then used as the codomain of the "="-operation, whose domain will be M_ x M_ [therefore "=" can only compare lists, not atoms]. For the "="-axioms we will also need an "if then else"-operation over Mb, which we call "[]" and use in the mixfix notation p[t]e, meaning "if p then t else e" [thus the brackets are abused as two infixes separating the three arguments of the single operator they represent]. The application of "=" will also necessitate a version "[]" of "[]", which uses elements in M_, not in Mb, as "then" and "else" parts.

Clearly the "unconditioned" algebra ALCO2~$b -- with the Mb-valued operator "=" being a predicate rather than a constructor -- is no longer a proper constructor algebra. However, in the ADT view the entire carrier Mb = {T, F} can be regarded as an auxiliary or "helper" type (Goguen et al. 1978), with the operations "=", "[]", and "[]", involving Mb, being "hidden", i.e. invisible from the outside. For boolean-extended algebras which could be obtained similarly from ALCO3~-ALCO5~ in section 5 the same remarks would apply.

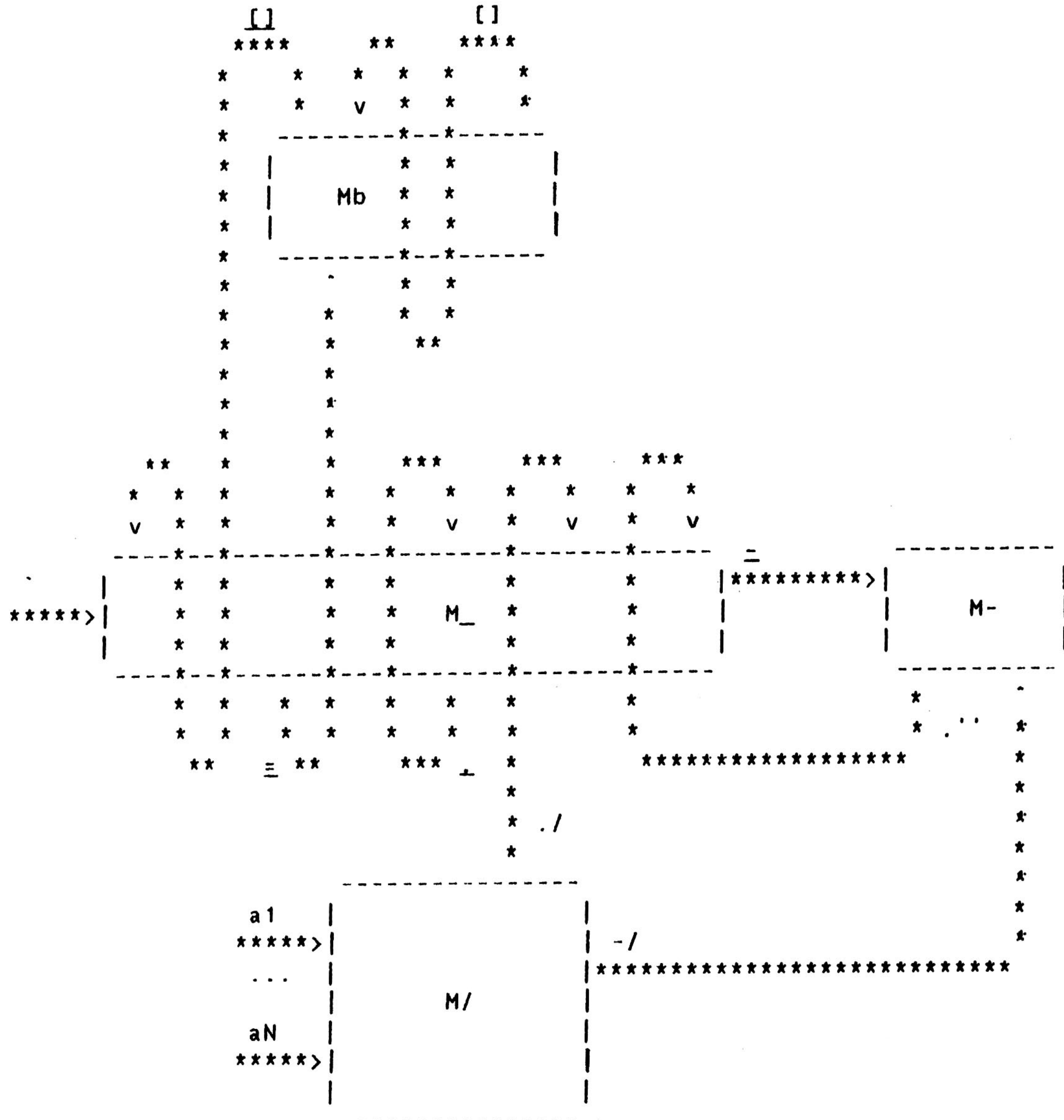
Definition2~$b:

A_ = {`}	generating set [a singleton set]
A/ = {a1, ..., aN}	generating set [a generator]
A- = {}	generating set [the empty set]

ALCO2~$b = (Mb,M_,M/,M-;[],[],=,⊥,./,.'',-,-/) algebra with
(M_,M/,M-;⊥,./,.'',-,-/) as ALCO2~$

Mb	finite carrier [a two-element set]
T, F IN Mb	boolean elements
[] : Mb x Mb x Mb -> Mb	ternary operation ["if then else" on booleans]
[](m1,m2,m3) = m1[m2]m3	[m1, m2, m3 IN Mb]
[] : Mb x M_ x M_ -> M_	ternary operation ["if then else" on lists]
[](m1,m2,m3) = m1[m2]m3	[m1 IN Mb and m2, m3 IN M_]
= : M_ x M_ -> Mb	binary operation [list equality]
=(m1,m2) = m1=m2	[m1 IN M_ and m2 IN M_]

As a diagram ALCO2~$b can be depicted thus [assuming the generator is A/ = {a1, ..., aN}]:



The axioms below, defining "=", "[]", and "[]", are postulated for ALCO2~$b. Let us start with the easy ones, the branch axioms for "[]" and "[]":

```
[]branch1: T[m]m' = m
[]branch2: F[m]m' = m'

[]branch1: T[m]m' = m
[]branch2: F[m]m' = m'
```

To express the equality axioms for "=", we will use the following conventions: First, indexes I and J are written in parentheses after certain axiom names to indicate how these are parameterized as axiom schemata. Second, it must be emphasized that the "for" clauses on eq1-eq4 -- unlike the usual "if" clauses -- are meant as meta-level statements: Since it will be used as part of our "unconditioning" method, we also have to define "=" itself without resorting to conditional equations. Third, in order to avoid extensive case analysis, we let ".'" generically denote "./" if its left argument is a

non-"`" atom and "." otherwise; similarly, in the "for" clauses we refer to a reunited generating set A = {a0, a1, ..., aN}, where a0=`. To give two examples, eq1(I) will abbreviate the N+1 unconditional equations

```
`.m=`.m'       = m=m'
a1./m=a1./m'   = m=m'
               ...
aN./m=aN./m'   = m=m'
```

and eq8 will stand for the N+1 equations

```
(m1.m2)=`    = F
(a1./m2)=`   = F
             ...
(aN./m2)=`   = F
```

With these conventions, we can now specify the eight eq axioms:

```
equality1(I):    (aI.'m)=(aI.'m') = m=m'  for aI IN A
equality2(I,J):  (aI.'m)=(aJ.'m') = F  for aI, aJ IN A with aI <> aJ
equality3(I):    (aI.'m)=((m1.'m2).m') = F  for aI IN A
equality4(I):    ((m1.'m2).m)=(aI.'m') = F  for aI IN A
equality5:       ((m1.'m2).m)=((m3.'m4).m') = ((m1.'m2)=(m3.'m4))[m=m']F
equality6:       `=` = T
equality7:       `=(m1.'m2) = F
equality8:       (m1.'m2)=` = F
```

Presupposing the generator A/ = {a1, a2, a3}, where a1=a, a2=b, a3=c, some terms of the carriers Mb and M_ of ALCO2~$b and their []b/[]b/eq-reductions are the following.

```
Mb: T[F]T =[]b1= F
M_: (T[F]T)[`](a1./`) =[]b1= F[`](a1./`) =[]b2= (a1./`)
Mb: (a1./`)=(a2./`) =eq2(1,2)= F
Mb: (a1./`)=(a1./`) =eq1(1)= `=` =eq6= T
Mb: (a2./((a3./`).`))=(a2./((a3./`).(a1./`))) =eq1(2)=
    ((a3./`).`)=((a3./`).(a1./`)) =eq5=
    ((a3./`)=(a3./`))[(`=(a1./`))]F =eq1(3)=
    (`=`)[(`=(a1./`))]F =eq6=
    T[(`=(a1./`))]F =[]b1=
    `=(a1./`) =eq7=
    F
M_:  (((a3./(a1./`)).(a2./(a1./(a3./`))))=(a1./(a2./(a1./(a3./`)))))
    [
     (a1./(a2./(a1./(a3./`))))
    ]
     `

    =eq4(1)=
    F[(a1./(a2./(a1./(a3./`))))]`
    =[]b2=
    `
```

Notice that the last but first reduction could be shortened by introducing the "short-cut" axiom []b3: m[m']m' = m', because, after the third reduction state, we could then continue without ever reducing the "if" part, ... =eq7= ((a3./`)=(a3./`))[F]F =[]b3= F. An even more drastic shortening effect for all T-valued "="-terms could be achieved

by generalizing eq6 to eq6': m≡m = T, thus permitting one-step proofs for arbitrarily large equal lists m, e.g. (a1./(a2./`))≡(a1./(a2./`)) =eq6'= T, as usable to shorten the proof of Lem22~$b in section 4.2 [the other eq axioms do not thereby become unnecessary, of course, because F-valued "≡"-terms must also be reduced -- our algebras "don't know" something like "negation by failure"].

4 THE BASIC PROPERTIES AND THE BASIC COLLECTIONS

In this section, we first introduce the basic properties of Commutativity, Idempotence, and Associativity as axioms into our list algebra. The initial capital letters are used to indicate that these notions get their meaning from the N-ary interpretation of the algebra. Not taking into account this N-ary interpretation, the "capital initial" notions differ considerably from the corresponding classic binary "small initial" notions.

Using the basic properties, we then go on to define the basic collections as different "quotients" [algebras derived by adding axioms to given algebras] of the algebra ALCO2 introduced in section 3.2. The discussion of basic collections will include another possible representation of [variable-length] strings as nested [fixed-length] terms, complementing the well-known representation of nested terms as strings with parenthesis characters. [Since both can represent each other, neither strings nor terms can claim to be "more basic", and the traditional "string orientation" in theoretical computer science may be worth reconsidering in the light of the "term orientation" in term-rewriting systems, computer algebra, and algebraic ADTs, as well as in automatic theorem proving, functional/relational list processing, and other AI techniques.]

4.1 Commutativity

What we interpret as Commutativity on N-tuples, is really a new property on pairs, quite different from classic binary commutativity, i.e. from

commutativity: m1.m2 = m2.m1

While commutativity is applicable to two subterms connected by the binary operator, Commutativity is applicable only to three subterms connected by two occurrences of the operator in a right-associating manner. Whereas commutativity exchanges the two operands of the single operator, Commutativity exchanges the first operand of the outer operator with the first operand of the inner operator, the second inner operand remaining unchanged.

```
Commutativity: m1.(m2.m3) = m2.(m1.m3)  if m1 <> -n and m2 <> -n'
shorter:       m1.m2.m3   = m2.m1.m3    if m1 <> -n and m2 <> -n'
```

The condition on the Commutativity equation is only needed for its use in algebras like ALCO2, having negated elements: Neither of the subterms m1 and m2 may have a negative sign, as introduced by Idempotence and specially treated by the Remove axioms [section 4.2].

For the many-sorted algebra ALCO2~ the following variant of Commutativity can be used [for the additional generator separation in ALCO2~$ we get C~$ by just adopting the generic view of ".'" as "./" or "⊥", as exemplified with the eq axioms in section 3.2]:

```
Commutativity~:  m1.'(m2.'m3) = m2.'(m1.'m3)
shorter:         m1.'m2.'m3   = m2.'m1.'m3
```

In this heterogeneous formulation the condition m1 <> -n and m2 <> -n' of the homogeneous version has become implicit, because the very use of the single-primed operator ".'" [rather than the double-primed ".''"] to the right of m1 and m2 forces the values of these variables to be unnegated.

For the many-sorted algebras ALCO3~-ALCO5~ in section 5 we will need a more general kind of Commutativity, involving two possibly different binary operators "*'" and "*''" [extending our previous naming convention to axioms with two equal-right operators, we prefix the axiom name with both of these operators]:

```
*'*''Commutativity~:  m1*'(m2*''m3) = m2*''(m1*'m3)
shorter:              m1*'m2*''m3   = m2*''m1*'m3
```

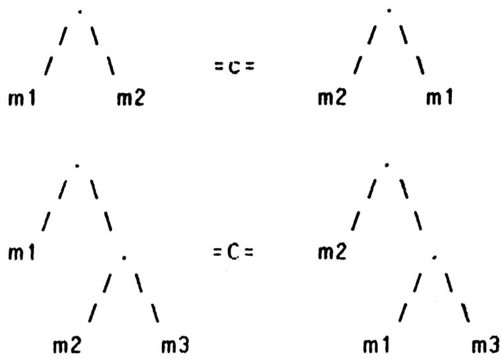
This kind of Commutativity, *'*''C~, generalizes the previous one, .'C~, because by setting *' = .' and *'' = .' in *'*''C~, we obtain .'.'C~ [shortenable to .'C~ by joining the now identical operators].

A single C application can exchange adjacent elements only [see Lem19 below for an example]. Repeated C applications, however, can exchange elements between which there is an arbitrary number of intervening elements [see Lem20 below for one intervening element]; more generally, they can produce arbitrary permutations of the elements.

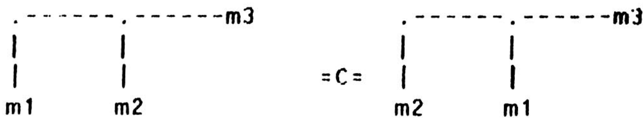
Examples:

```
Lem19: a.(b.(c.`)) = a.(c.(b.`))
Proof: b.(c.`) =C= c.(b.`)
       a = a and b.(c.`) = c.(b.`)
       a.(b.(c.`)) =E= a.(c.(b.`))

Lem20: b.(a.(c.`)) = c.(a.(b.`))
Proof: b.(a.(c.`)) =C= a.(b.(c.`))
       a.(b.(c.`)) =Lem19= a.(c.(b.`))
       a.(c.(b.`)) =C= c.(a.(b.`))
```

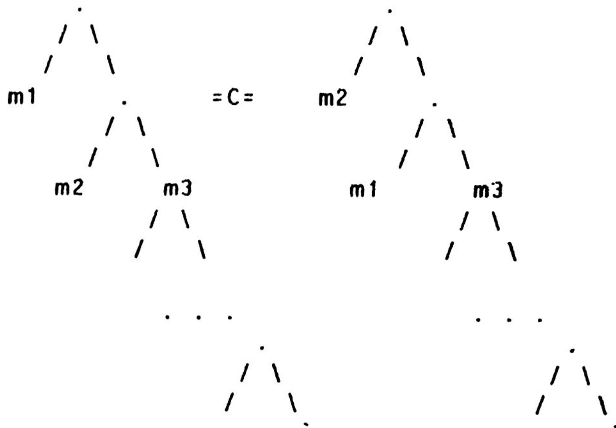
If viewed as transformations on binary trees, commutativity can be seen to exchange subtrees on the same level, while Commutativity exchanges subtrees on different levels.



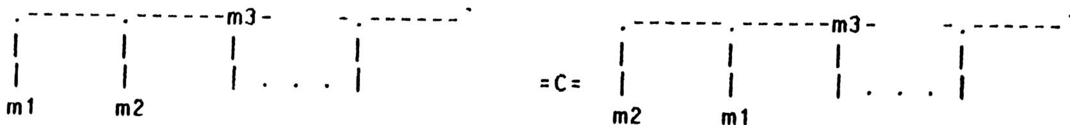
When visualizing our N-ary interpretation through a "stretched node" representation of N-ary trees, however, Commutativity can also be seen to involve only subtrees on one level [as indicated by the short form m1.m2.m3 = m2.m1.m3].



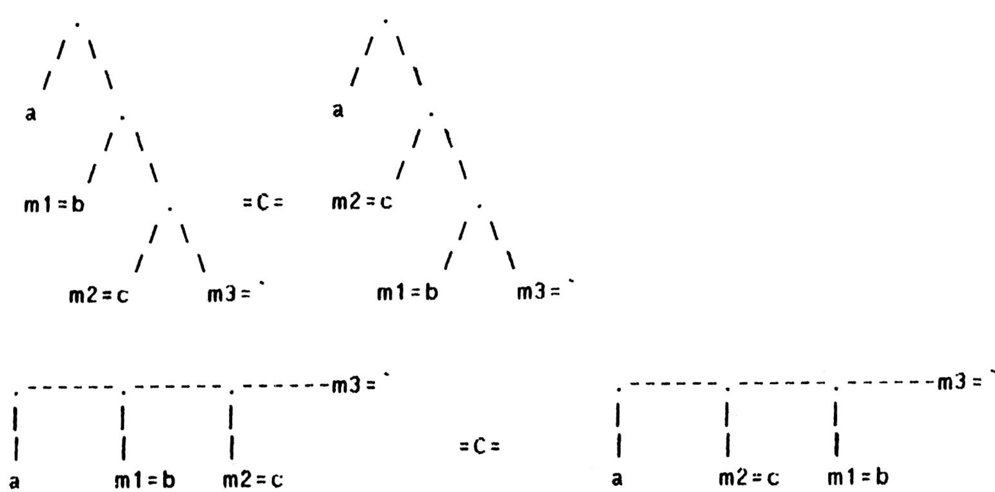
Therefore, while in the binary interpretation of Commutativity m3 stands for the root of an arbitrarily deep binary subtree remainder,



in the N-ary interpretation m3 abbreviates the remainder of a stretched node from which any number of further branches may fan out:



In most later proofs, instead of first showing the equality of parts of expressions and then using E explicitly to show the Equality of the entire expressions, we will -- as is usual -- directly apply properties like Commutativity to well-formed subexpressions <u>inside</u> expressions, thus e.g. making the proof of Lem19 immediate, a.(b.(c.`)) =C= a.(c.(b.`)). Here, C is applied to the well-formed subexpression b.(c.`) with bindings m1=b, m2=c, and m3=`. In the tree view of expressions, this application can be illustrated thus:



4.2 Idempotence

Normally, idempotence is defined as replacing an operator with two identical operands by any one of these operands.

idempotence: m.m = m

Suppose, then, we define N-ary Idempotence as "Adjacentpotence" as follows:

Adjacentpotence: m1.(m1.m2) = m1.m2
shorter: m1.m1.m2 = m1.m2

This generalizes binary idempotence in a manner analogous to the way Commutativity generalizes commutativity, but observe that one "."-operator always remains under Adjacentpotence, whereas the "."-operator is removed under idempotence. The difference is clearly seen if there are only two non-"`" elements, as in a.a =i= a versus a.a.` =Aj= a.`. A definition like Aj would not, however, permit the idempotent removal of one out of two non-adjacent identical elements [hence the name "Adjacentpotence"]. Of course, this would not be necessary for sets, since their additional property of Commutativity would allow the identical elements to "commute" together before Adjacentpotence would have to be applied, as e.g. in [we use informal set notation] {a, b, a} =C= {a, a, b} =Aj= {a, b}. If, instead, we would like to have more general idempotent data structures, such as "non-commutative sets" [i.e. ordered sets or communes], idempotence would have to be capable of removing non-adjacent elements without relying on their commuting together beforehand. Since we do in fact want to formalize such data structures, we will introduce a generalized concept of Idempotence, in this way also ensuring maximum independence among the basic properties.

For this we will make use of the "-"-operator of ALCO2. The semantics of the negated elements in ALCO2 combines aspects of left inverse elements in groups and left zero elements in semigroups. Like an inverse, a negated element can only remove unnegated occurrences of itself; like a zero it itself remains intact after removal for further action [R1]. Unlike ordinary inverses and zeroes it can commute with

those elements to its right which are not its unnegated versions [R2] until it disappears at the "end marker" element "`" [R3]. Altogether, negated elements may sweep "horizontally through" [N-ary interpretation] or "diagonally down" [binary interpretation] a "."-product, cleaning it from all their unnegated versions.

```
Remove1:   -m1.(m1.m2) = -m1.m2
shorter:   -m1.m1.m2   = -m1.m2
Remove2:   -m1.(m2.m3) = m2.(-m1.m3)    if m2 <> m1
shorter:   -m1.m2.m3   = m2.-m1.m3      if m2 <> m1
Remove3:   -m.`        = `
```

The condition on the R2 equation prevents a negated element to just skip an unnegated [in general, once less negated] occurrence of itself. A negated element may also move from right to left by reading R2 in this direction; however, the R2 condition must be fulfilled for that reading too, i.e. the negated element cannot commute over its unnegated version in the right-to-left direction either. If it could, the unnegated element would enter into the removal scope of R1, so that R1 could wrongly remove elements in both directions; even worse, in combination with a right-to-left reading of R3 [or with Idempotence] for generating negative elements, <u>every</u> element could be removed, like the element a below [a "near miss" axiom which <u>cannot</u> be used for an equation chain step is written inside an inequality sign]:

```
a.` =R3= a.(-a.`)
    <R2> -a.(a.`)
    =R1= -a.`
    =R3= `
```

For the many-sorted algebra ALCO2~ the Remove axioms look like this [for their generic interpretation as R~$ see the I~$ remarks below]:

```
Remove1~:  -m1.''(m1.'m2) = -m1.''m2
shorter:   -m1.''m1.'m2   = -m1.''m2
Remove2~:  -m1.''(m2.'m3) = m2.'(-m1.''m3)   if m2 <> m1
shorter:   -m1.''m2.'m3   = m2.'-m1.''m3     if m2 <> m1
Remove3~:  -m.''`         = `
```

The heterogeneous axioms R1~ and R2~ do not reflect the complete capabilities of the homogeneous axioms R1 and R2, respectively: 1. While in R1 the variable m1 may denote some term which is negated itself, in R1~ the m1 denotation must be unnegated [m1 occurs as a left ".'"-argument]; however, the doubly negated elements with which R1 may thus redundantly remove singly negated ones cannot occur in ALCO2~ in any case. 2. Only with the additional heterogeneous R2~ variant -m1.''-m2.''m3 = -m2.''-m1.''m3 could negated elements commute with other negated elements; however, although this additional capability is implicit in the homogeneous axiom R2, it is never required, hence is omitted in the heterogeneous version.

Apparently, the condition m2 <> m1 of R2 proved immune against the homogeneous/heterogeneous transition. The heterogeneous axiom R2~ is actually the only remaining conditional axiom in the many-sorted generator-separated versions of our algebras of collection data. To guarantee initiality it is therefore worthwhile to eliminate the R2~ condition in generator-separated algebras. This is possible by using

the operators "=" and "[]" introduced in section 3.2. The use of "[]" is related to the use of explicit "if then else" operators as a means for avoiding conditional equations in (Goguen et al. 1978). The use of "=" will become clear in the following description of our elimination method.

Instead of ensuring equality implicitly with two occurrences of the variable m1 on the left-hand side of R1~ and ensuring inequality with the condition m2 <> m1 on R2~, these axioms are joined to an axiom R1+2~$b, which uses "[]" as its principal right-hand-side operator for branching on the outcome of an explicit "="-test into an R1~-like right-hand side, if the test yields T, and into an R2~-like right-hand side, if the test yields F. Since "=" is only applicable to lists, some supplementary axioms deal with the cases where one [R2~$b2, R2~$b3] or both [R1~$b, R2~$b1] of m1 and m2 are atoms; the last axiom [R3~$b] is simply a copy of R3~.

The Remove axioms, then, become [we use the conventions stated for the eq axioms in section 3.2, and furthermore let "-" generically denote "-/" if its argument is a non-"`" atom and "-" otherwise]:

```
Remove1~$b(I):      -aI.''(aI.'m) = -aI.''m  for aI IN A
shorter:            -aI.''aI.'m   = -aI.''m  for aI IN A

Remove2~$b1(I,J):  -aI.''(aJ.'m) = aJ.'(-aI.''m)  for aI <> aJ both IN A
shorter:            -aI.''aJ.'m   = aJ.'(-aI.''m)  for aI <> aJ both IN A
Remove2~$b2(I):     -aI.''((m1.'m2).m)  = (m1.'m2).(-aI.''m)  for aI IN A
shorter:            -aI.''(m1.'m2).m    = (m1.'m2).-aI.''m    for aI IN A
Remove2~$b3(I):     -(m1.'m2).''(aI.'m) = aI.'(-(m1.'m2).''m) for aI IN A
shorter:            -(m1.'m2).''aI.'m   = aI.'-(m1.'m2).''m   for aI IN A

Remove1+2~$b:       -(m1.'m2).''((m3.'m4).m) =
        ((m1.'m2)=(m3.'m4))[(-(m1.'m2).''m)]((m3.'m4).(-(m1.'m2).''m))
shorter:            -(m1.'m2).''(m3.'m4).m   =
        ((m1.'m2)=(m3.'m4))[(-(m1.'m2).''m)]((m3.'m4).-(m1.'m2).''m)

Remove3~$b:         -m.''`             = `
```

For the many-sorted algebras ALCO3~-ALCO5~ we will need more general Remove axioms, involving several binary operators "*'", "*''", "*'3'", ..., "*'M', along with several negation operators "-'", "-''", "-'3'", ..., "-'N', where "*'3'" or "*'''" is used to construct arbitrarily negated terms onto other terms. We have adopted the convention that op'X' stands for an op occurrence with X primes, so that, for instance, op'3' is equivalent to op'''. As a further convention, an operator op will denote itself if it has no primed versions and will generically denote any of its primed versions op'X' otherwise. With this convention the axioms can be expressed thus:

```
*Remove1~:  -m1*'''(m1*m2) = -m1*'''m2
shorter:    -m1*'''m1*m2   = -m1*'''m2
*Remove2~:  -m1*'''(m2*m3) = m2*(-m1*'''m3)   if m2 <> m1
shorter:    -m1*'''m2*m3   = m2*-m1*'''m3     if m2 <> m1
*Remove3~:  -m*'''`        = `
```

Each of the three axioms is really a scheme using the operators "*" and "-" as place-holders for multiplication and negation operators with any

required numbers of primes. Of course, on instantiation of an axiom scheme all of its "*"-occurrences ["-"-occurrences] must be given the same number of primes, whereas across the operators "*" and "-" the numbers of primes may be different. For R2~ this permits instantiations like -'m1*'''m2*''m3 = m2*''-'m1*'''m3 if m2 <> m1. Although the R1~ scheme permits similar instantiations, only instantiations of the form [j denotes the same number of primes for "*" and for "-"] -'j'm1*'''m1*'j'm2 = -'j'm1*'''m2 will have meaningful interpretations, because m1 will only be able to denote the same value both as the argument of "-" and as the first argument of "*" if both operators expect the same "sort" in these arguments, as ensured by their identical number of primes.

The condition m2 <> m1 could be eliminated here as shown for the ALCO2~ versions of the Remove axioms.

On the basis of the Remove axioms we can now define Idempotence as simply introducing a negated element to the right of an unnegated element.

```
Idempotence:   m1.m2 = m1.(-m1.m2)     if m1 <> -n
shorter:       m1.m2 = m1.-m1.m2       if m1 <> -n
```

The condition m1 <> -n would not be strictly necessary, but prevents a redundant generation of double negations.

For the many-sorted algebra ALCO2~ Idempotence becomes [if "-" and ".'" are viewed generically as in the R~$b and eq axioms, this I~ definition can also be used as I~$ and I~$b]:

```
Idempotence~:  m1.'m2 = m1.'(-m1.''m2)
shorter:       m1.'m2 = m1.'-m1.''m2
```

The condition m1 <> -n of the homogeneous algebra has become implicit here, because the use of m1 as a first ".'"-argument again "types" it to be unnegated.

For the many-sorted algebras ALCO3~-ALCO5~ we will need the following more general kind of Idempotence, expressed as an axiom scheme ["*" and "-" carry the same number, j, of primes]:

```
*'j'Idempotence~: m1*'j'm2 = m1*'j'(-'j'm1*'''m2)
shorter:          m1*'j'm2 = m1*'j'-'j'm1*'''m2
```

Since a generated negated element, -'j'm1, gets the same number of primes on its "-"-prefix as the unnegated original, m1*'j'..., has on the "*"-infix", the sort restriction of the chosen negation operator will be automatically fulfilled.

The first example below shows the elimination of a duplicated element a which is atomic, i.e. is taken from the generating set A [= {a0, a1, a2, a3}, where a0=`, a1=a, a2=b, a3=c]; the second demonstrates the removal of a duplicated complex element a.b.`, taken from the carrier M minus the generating set A.

<u>Examples</u>:

```
Lem21: c.(a.(b.(a.`))) = c.(a.(b.`))
Proof: a.(b.(a.`)) =I=
       a.(-a.(b.(a.`))) =R2=
       a.(b.(-a.(a.`))) =R1=
       a.(b.(-a.`)) =R3=
       a.(b.`)
       c = c and a.(b.(a.`)) = a.(b.`)
       c.(a.(b.(a.`))) =E= c.(a.(b.`))

Lem22: (a.(b.`)).((a.(b.`)).`) = (a.(b.`)).`
shorter: (a.b.`).(a.b.`).` = (a.b.`).`
Proof: (a.(b.`)).((a.(b.`)).`) =I=
       (a.(b.`)).(-(a.(b.`)).((a.(b.`)).`)) =R1=
       (a.(b.`)).(-(a.(b.`)).`) =R3=
       (a.(b.`)).`
```

The following two examples parallel the previous ones, reformulated for the many-sorted generator-separated boolean-extended algebra ALCO2~$b [see end of section 3.2].

<u>Examples</u>:

```
Lem21~$b: c./(a./(b./(a./`))) = c./(a./(b./`))
Proof:    a./(b./(a./`)) =I~$b=
          a./(-/a.''(b./(a./`))) =R2~$b1(1,2)=
          a./(b./(-/a.''(a./`))) =R1~$b(1)=
          a./(b./(-/a.''`)) =R3~$b=
          a./(b./`)
          c = c and a./(b./(a./`)) = a./(b./`)
          c./(a./(b./(a./`))) =E= c./(a./(b./`))

Lem22~$b: (a./(b./`))⊥((a./(b./`))⊥`) = (a./(b./`))⊥`
shorter:  (a./b./`)⊥(a./b./`)⊥` = (a./b./`)⊥`
Proof:    (a./(b./`))⊥((a./(b./`))⊥`) =I~$b=
          (a./(b./`))⊥(-̲(a./(b./`)).''((a./(b./`))⊥`))
          =R1+2~$b=
           (a./(b./`))
          ⊥(((a./(b./`))≡(a./(b./`)))
            [
            (-̲(a./(b./`)).''`)
            ]
            ((a./(b./`))⊥(-̲(a./(b./`)).''`)))
          =eq1(1)=
           (a./(b./`))
          ⊥(((b./`)≡(b./`))
            [
            (-̲(a./(b./`)).''`)
            ]
            ((a./(b./`))⊥(-̲(a./(b./`)).''`)))
```

```
=eq1(2)=
 (a./(b./`))
⊥((`=`)
  [
  (-(a./(b./`)).'``)
  ]
  ((a./(b./`))⊥(-(a./(b./`)).'``)))
=eq6=
 (a./(b./`))
⊥(T
  [
  (-(a./(b./`)).'``)
  ]
  ((a./(b./`))⊥(-(a./(b./`)).'``)))
=[]b1=
(a./(b./`))⊥(-(a./(b./`)).'``) =R3~$b=
(a./(b./`))⊥`
```

Since Idempotence can generate a negated element to the right of every unnegated element, it may happen that an element first idempotently removes some elements to its right but is then itself likewise removed by an element to its left [Proof1 below]; this is however just an inefficient deviation from the principle of applying Idempotence always to the left-most occurrence of repeated elements [Proof2 below].

```
Lem23:  b.(a.(b.(c.(b.`)))) = b.(a.(c.`))

Proof1: b.(a.(b.(c.(b.`)))) =I=
        b.(a.(b.(-b.(c.(b.`))))) =R2=
        b.(a.(b.(c.(-b.(b.`))))) =R1=
        b.(a.(b.(c.(-b.`)))) =R3=
        b.(a.(b.(c.`))) =I=
        b.(-b.(a.(b.(c.`)))) =R2=
        b.(a.(-b.(b.(c.`)))) =R1=
        b.(a.(-b.(c.`))) =R2=
        b.(a.(c.(-b.`))) =R3=
        b.(a.(c.`))

Proof2: b.(a.(b.(c.(b.`)))) =I=
        b.(-b.(a.(b.(c.(b.`))))) =R2=
        b.(a.(-b.(b.(c.(b.`))))) =R1=
        b.(a.(-b.(c.(b.`)))) =R2=
        b.(a.(c.(-b.(b.`)))) =R1=
        b.(a.(c.(-b.`))) =R3=
        b.(a.(c.`))
```

In some equality proofs and similar applications [e.g. the matching of idempotent collections] it may be useful to <u>generate</u> duplicates besides removing them. This could be accomplished by postulating not only Idempotence but also Adjacentpotence [see above], because through a right-to-left reading of the latter every element can be duplicated in one step. However, we will not do this because two complementary Idempotence readings [with a crucial backward reading of R1] can also duplicate arbitrary [unnegated] elements, though requiring three steps. This is shown in the following lemma, involving an arbitrary element x not equal to -n in an arbitrary right-context y [where the entire term

x.y may be arbitrarily embedded into another term]; the lemma and its proof make use of the N-ary parentheses-sparing short notation.

```
Lem24: x.y = x.x.y  if x <> -n
Proof: x.y =I=
       x.-x.y =R1=
       x.-x.x.y =I=
       x.x.y
```

[The condition x <> -n accounts for the way we defined I.]

As an example of the many-sorted reformulation of a lemma, we prove the Lem24 variant Lem24~:

```
Lem24~: x.'y = x.'x.'y
Proof:  x.'y =I~=
        x.'-x.''y =R1~=
        x.'-x.''x.'y =I~=
        x.'x.'y
```

[The condition x <> -n has become implicit as in I~.]

With the help of that lemma, we can now solve the equation [the matching problem] involving the variables u and v,

```
a.u.v.c.` = a.b.c.`
```

or in FIT's collection notation [the idempotent "."-operator is represented by the COMMUNE tag], (COMMUNE a u v c) = (COMMUNE a b c):

First, the right-hand side is "semantically rewritten" [using Idempotence] as

```
a.b.c.` =Lem24= a.b.b.c.`
```

where Lem24 is applied at the underlined subterm with the bindings x = b and y = c.`.

Second, the new problem

```
a.u.v.c.` = a.b.b.c.`
```

is "syntactically reduced" [using Equality] to

```
a = a  and  u = b  and  v = b  and  c = c  and  ` = `.
```

Thus the equation is valid [the match succeeds], producing the solutions [bindings] u = b and v = b [i.e. b is "shared idempotently" by the variables u and v].

The possibility of reading Idempotence backward -- exemplified in the last step of the above Lem24 proof -- permits negated elements to disappear at the source where they were generated, rather than at the end marker. Since a negated element immediately before "`" -- the only position where R3 is applicable -- can move back to the place of its generation, this means that R3 is really redundant for

Idempotence-generated negated elements. However, the R3 axiom is required for "cleaning" terms containing non-Idempotence-generated negative signs, like no. 399 in the first table in section 3.2, i.e. reducing them to "-"-less representatives by equalities like b.-a.` =R3= b.`. There are also computational reasons for keeping R3, namely 1. from the place where a negated element removes its last element it is often nearer to the end marker than to the unnegated source and 2. a systematic check for duplicates should walk down an entire expression until reaching the end marker, so that it is easiest to remove it there in one step.

4.3 Associativity

The central equation A1 for N-ary Associativity only differs from the usual binary notion of associativity in an additional condition preventing transitions between "erroneous" and "non-erroneous" terms, e.g. (a.b).`, producible by `.`, is not A1-equal to a.(b.`):

`.` =lr= (a.b).` <A1> a.(b.`)

However, a supplementary equation A2 is necessary in the N-ary case because N-ary Associativity should also be able to remove embedded "`"-elements [as introduced by applications of A1 or in a direct manner]; since in the binary representation this means that the empty element "`" is used as a left factor, we define it as a correspondingly conditioned left identity. This is a typical example of an intensional difference, illustrated by two quite dissimilar informal tree equations [conditions are immaterial for this comparison]

```
.-    -.-----.-----.-    -.-----`              .-    -.-----.-    -.-----`
| ... |      |     | ... |                =    | ... |      | ... |
x1    xI-1   `     xI+1  xN                    x1    xI-1   xI+1  xN


               .
              / \                         =              y
             `   y
```

between the extensionally indistinguishable N-ary and binary interpretations: Because it can be applied to subexpressions, the formal equation defining the embedding of "`" inside N-ary expressions is the same as the one defining "`" as a binary left identity.

Associativity1:	(m1.m2).m3 = m1.(m2.m3)	if m2 = n.n' or m2 = `
shorter:	(m1.m2).m3 = m1.m2.m3	if m2 = n.n' or m2 = `
Associativity2:	`.m = m	if m = n.n' or m = `

The conditions m2 = n.n' or m2 = ` as well as m = n.n' or m = ` could, of course, be eliminated easily, by or-splitting followed by substitution.

For the many-sorted algebra ALCO2~ the one-sorted axioms A1 and A2 can be taken over by setting "." = ".'" and adding a disjunct to the A2 condition:

```
Associativity1~:  (m1.'m2).'m3 = m1.'(m2.'m3)  if m2 = n.'n' or m2 = `
shorter:          (m1.'m2).'m3 = m1.'m2.'m3    if m2 = n.'n' or m2 = `
Associativity2~:  `.'m = m        if m  = n.'n' or m = n.''n' or m  = ``
```

It is true that only with the additional heterogeneous A1~ variant (-m1.''m2).'m3 = -m1.''m2.'m3 if m2 = n.'n' or m2 = n.''n' or m2 = `` and with an additional disjunct, m2 = n.''n', also in the A1~ condition could negated elements be embedded associatively, but this capability, implicit in the homogeneous axiom A1, is never required [e.g. normalizations can and should apply A before I], hence it is omitted in the heterogeneous case.

Finally, for the many-sorted generator-separated algebra ALCO2~$ the ".'"-occurrences with a non-generator left argument are replaced by "._" and the remaining ones are reinterpreted generically as "./" or "._", as usual:

```
Associativity1~$:  (m1.'m2)._m3 = m1.'(m2._m3)
shorter:           (m1.'m2)._m3 = m1.'m2._m3
Associativity2~$:  `._m = m
```

Recall that for a generator A/ = {a1, ..., aN} the generic abbreviation convention lets the axiom A1~$ stand for the N+1 ".'"-less equations

```
(m1._m2)._m3  = m1._m2._m3
(a1./m2)._m3 = a1./m2._m3
            ...
(aN./m2)._m3 = aN./m2._m3
```

Also note that the conditions on both the A1~ and A2~ equations have become implicit in A1~$ and A2~$, since the second arguments of "._" and "./" cannot lie in M/ = A/.

<u>Example</u>:

```
Lem25: (a.(b.`)).(c.`) = a.(b.(c.`)) = a.((b.(c.`)).`)
shorter: (a.b.`).c.` = a.b.c.` = a.(b.c.`).`
Proof: (a.(b.`)).(c.`) =A1=
       a.((b.`).(c.`)) =A1=
       a.(b.(`.(c.`))) =A2=
       a.(b.(c.`)) =A2=
       a.(b.(c.(`.`))) =A1=
       a.(b.((c.`).`)) =A1=
       a.((b.(c.`)).`)
```

4.4 The Eight Basic Collections

The eight basic collections can now be defined by adding a combination of zero or more of the three basic axioms A, I, C to the algebra ALCO2. [The algebra ALCO1 would be sufficient for non-idempotent collections, but we prefer ALCO2 for reasons of consistency.] We will denote an algebra ALG for which the axioms ax1, ..., axN are postulated as ALG[ax1,...,axN]; using technical terms explained in (Goguen et al. 1978), ALG[ax1,...,axN] is the quotient of ALG by the congruence relation obtained from the set of equations

{ax1, ..., axN}. For making this notation more concise, subordinate properties required for the axioms, such as Remove required for Idempotence, will be assumed implicitly. To illustrate our notation with the largest example, associative-idempotent-commutative collections [i.e. heaps] will be written as ALCO2[1r,A,I,C], which by our previous suffixing convention is a short form of ALCO2[1r,A1,A2,I,C] and with the additional subordinates convention stands for the quotient algebra ALCO2[1r,A1,A2,I,R1,R2,R3,C].

In general, a collection term will be said to be in <u>normal form</u> if no possible application of its axiomatic operator restriction or other size-changing axioms can [further] decrease its size ["term size" is measured as the number of its operators plus its elements from A], and no possible application of its size-preserving axioms can add [any more] to its lexicographic order [we use an extended concept of "lexicographic order", where, e.g., atoms like "c" precede "."-terms like "(a.b.`)"]. In particular, a basic collection term is in normal form if no possible application of listrestriction, Associativity, or Idempotence [including the Remove subordinates] can decrease its size, and no possible application of Commutativity [which never changes sizes] can add to its lexicographic order. [Tuples, only repeated here for completeness, are always in normal form, given that listrestriction is inapplicable.]

Below, each of the basic collections is exemplified through a "normalization chain", i.e. a sequence of equations leading to an expression in normal form. The eight normalizations will all start from the same expression to clearly exhibit the differences between the collections. In these examples we will use the N-ary parentheses-saving notation.

<u>1. Tuples</u>

ALCO2[1r]

```
c.(a.b.`).c.a.` = c.(a.b.`).c.a.`
```

<u>2. Strings</u>

ALCO2[1r,A]

```
c.(a.b.`).c.a.` =A1= c.a.(b.`).c.a.`
                =A1= c.a.b.`.c.a.`
                =A2= c.a.b.c.a.`
```

<u>3. Communes</u>

ALCO2[1r,I]

```
c.(a.b.`).c.a.` =I=  c.-c.(a.b.`).c.a.`
                =R2= c.(a.b.`).-c.c.a.`
                =R1= c.(a.b.`).-c.a.`
                =R2= c.(a.b.`).a.-c.`
                =R3= c.(a.b.`).a.`
```

4. Acommunes

ALCO2[1r,A,I]

```
c.(a.b.`).c.a.` =A1= c.a.(b.`).c.a.`
                =A1= c.a.b.`.c.a.`
                =A2= c.a.b.c.a.`
                =I= c.-c.a.b.c.a.`
                =R2= c.a.-c.b.c.a.`
                =R2= c.a.b.-c.c.a.`
                =R1= c.a.b.-c.a.`
                =R2= c.a.b.a.-c.`
                =R3= c.a.b.a.`
                =I= c.a.-a.b.a.`
                =R2= c.a.b.-a.a.`
                =R1= c.a.b.-a.`
                =R3= c.a.b.`
```

5. Bags

ALCO2[1r,C]

```
c.(a.b.`).c.a.` =C= c.c.(a.b.`).a.`
                =C= c.c.a.(a.b.`).`
                =C= c.a.c.(a.b.`).`
                =C= a.c.c.(a.b.`).`
```

6. Abags

ALCO2[1r,A,C]

```
c.(a.b.`).c.a.` =A1= c.a.(b.`).c.a.`
                =A1= c.a.b.`.c.a.`
                =A2= c.a.b.c.a.`
                =C= a.c.b.c.a.`
                =C= a.b.c.c.a.`
                =C= a.b.c.a.c.`
                =C= a.b.a.c.c.`
                =C= a.a.b.c.c.`
```

7. Sets

ALCO2[1r,I,C]

```
c.(a.b.`).c.a.` =I= c.-c.(a.b.`).c.a.`
                =R2= c.(a.b.`).-c.c.a.`
                =R1= c.(a.b.`).-c.a.`
                =R2= c.(a.b.`).a.-c.`
                =R3= c.(a.b.`).a.`
                =C= c.a.(a.b.`).`
                =C= a.c.(a.b.`).`
```

8. Heaps

ALCO2[1r,A,I,C]

```
c.(a.b.`).c.a.` =A1= c.a.(b.`).c.a.`
                =A1= c.a.b.`.c.a.`
                =A2= c.a.b.c.a.`
                =I= c.-c.a.b.c.a.`
                =R2= c.a.-c.b.c.a.`
                =R2= c.a.b.-c.c.a.`
                =R1= c.a.b.-c.a.`
                =R2= c.a.b.a.-c.`
                =R3= c.a.b.a.`
                =I= c.a.-a.b.a.`
                =R2= c.a.b.-a.a.`
                =R1= c.a.b.-a.`
                =R3= c.a.b.`
                =C= a.c.b.`
                =C= a.b.c.`
```

A comparison of these basic collections may reveal interesting relationships: For one example, strings can be viewed as "associative tuples", ALCO2[1r,A] = ALCO2[1r][A], just as heaps can be viewed as "associative sets", ALCO2[1r,A,I,C] = ALCO2[1r,I,C][A]. Thus in 7., where "." is the set constructor, Associativity is not postulated [while in 8., with "." being the heap constructor, it is]; in this sense, in our constructor algebraic formalization sets <u>are</u> <u>not</u> associative, e.g. in this algebra (a.b.`).c.` <> a.(b.c.`).`, or in the usual notation, {{a,b},c} <> {a,{b,c}}. This statement is of course unrelated to the fact that in the usual "algebra of sets" the UNION and INTERSECTION operations <u>are</u> associative. In fact, we could enrich ALCO2[1r,I,C] by UNION and INTERSECTION [and COMPLEMENTATION] operations, so as to obtain a model of the algebra of finite sets, in which "." is not associative but UNION and INTERSECTION are.

Instead of taking quotients of the homogeneous algebra ALCO2, the basic collections can also be defined as quotients of the many-sorted algebra ALCO2~ [left column] and of the many-sorted generator-separated algebra ALCO2~$ [right column]:

1.	ALCO2~[.'1r~,.''1r~]	ALCO2~$[]
2.	ALCO2~[.'1r~,.''1r~,A~]	ALCO2~$[A~$]
3.	ALCO2~[.'1r~,.''1r~,I~]	ALCO2~$[I~$]
4.	ALCO2~[.'1r~,.''1r~,A~,I~]	ALCO2~$[A~$,I~$]
5.	ALCO2~[.'1r~,.''1r~,C~]	ALCO2~$[C~$]
6.	ALCO2~[.'1r~,.''1r~,A~,C~]	ALCO2~$[A~$,C~$]
7.	ALCO2~[.'1r~,.''1r~,I~,C~]	ALCO2~$[I~$,C~$]
8.	ALCO2~[.'1r~,.''1r~,A~,I~,C~]	ALCO2~$[A~$,I~$,C~$]

Let us exemplify normalization chains in the many-sorted case with the last basic collection, heaps, since with these all three basic properties can be demonstrated:

```
c.'(a.'b.'`).'c.'a.'`  =A1~= c.'a.'(b.'`).'c.'a.'`
                       =A1~= c.'a.'b.'`.'c.'a.'`
                       =A2~= c.'a.'b.'c.'a.'`
                       =I~= c.'-c.''a.'b.'c.'a.'`
                       =R2~= c.'a.'-c.''b.'c.'a.'`
                       =R2~= c.'a.'b.'-c.''c.'a.'`
                       =R1~= c.'a.'b.'-c.''a.'`
                       =R2~= c.'a.'b.'a.'-c.''`
                       =R3~= c.'a.'b.'a.'`
                       =I~= c.'a.'-a.''b.'a.'`
                       =R2~= c.'a.'b.'-a.''a.'`
                       =R1~= c.'a.'b.'-a.''`
                       =R3~= c.'a.'b.'`
                       =C~= a.'c.'b.'`
                       =C~= a.'b.'c.'`
```

In the many-sorted generator-separated case the chain would become:

```
c./(a./b./`)_.c./a./`  =A1~$= c./a./(b./`)_.c./a./`
                               . . .
                       =C~$= a./b./c./`
```

To formalize mixed nestings of the eight collections, one algebra ALCOL = (M,*1,*2,...,*8,-) can be introduced in place of the eight separate algebras ALCO2[lr], ALCO2[lr,A], ..., ALCO2[lr,A,I,C], with the operator *i [and an associated distinguished empty element \i] corresponding to the ith basic collection [1 ≤ i ≤ 8], and with the axioms being postulated for each operator individually [an axiom with respect to an operation "O" is called an Oaxiom]:

ALCOL[*1lr,...,*8lr,*2A,*3I,*4A,*4I,*5C,*6A,*6C,*7I,*7C,*8A,*8I,*8C]

Each of the eight ALCO2 constructor algebras as well as their ALCOL combination can be enriched by generic selectors "h" [head] and "t" [tail], obtaining algebras ALCO2-ht = (M,.,-,h,t) and ALCOL-ht = (M,*1,*2,...,*8,-,h,t) with the essential additional axioms

```
h(m1.m2) = m1
t(m1.m2) = m2
```

and for 1 ≤ i ≤ 8   [the right-hand side illustrates tuples and sets]

```
h(m1 *i m2) = m1     h(a *1 b *1 \1) = a
t(m1 *i m2) = m2     t(a *7 b *7 \7) =*7C= t(b *7 a *7 \7) = (a *7 \7)
```

Similarly, a generic append operator "&" -- for sets corresponding to UNION -- can be introduced in all basic collections:

```
Append1: (m1.m2)&m3 = m1.(m2&m3)  if m2 = n.n' or m2 = `
Append2:         `&m = m          if m  = n.n' or m  = `
```

If Associativity holds for the operator "." then x&y = x.y holds for all x = x1.(...(xN.`)..) [i.e. x must end in "`"] or x = ` and for all y = y1.y2 or y = `, because by induction over the length l of x we have [the forms of x and y cause the A and Ap conditions to be always satisfied]:

```
x = `;       l(x) = 0 :                      `&y =Ap2= y =A2 = `.y
x = x1.x2; l(x2) = n-1 -> l(x1.x2) = n : (x1.x2)&y =Ap1=
                                           x1.(x2&y) =induct.hypothesis=
                                           x1.(x2.y) =A1=
                                           (x1.x2).y
```

In particular, for the four associative basic collections [strings, acommunes, abags, and heaps] the concatenation operator "." <u>is</u> the append operator "&". For the other four basic collections the operators "." and "&" differ precisely in the dimension of associativity, as exemplified for sets above [where "&" = "UNION"].

The "h", "t", and "&" operators could also be introduced over both the arcs [hyperarcs] and the graphs [complex labelnodes] of the non-basic collections studied in the following section.

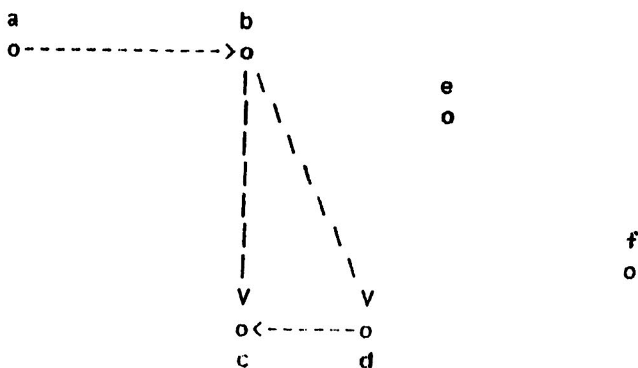
5 THE ADSORPTION PROPERTIES AND GRAPH COLLECTIONS

Usually, directed graphs are defined as pairs consisting of a node set and an arc relation. For reasons to be explained in subsection 5.1 we prefer to define directed graphs as sets consisting of isolated node individuals and arc tuples. These node and arc sets should however have one more property than ordinary sets: A node also occurring in an arc should be removable from the set because, through its additional arc occurrence, it ceases to be isolated. We call this property "adsorption", since it is similar to the absorption property of lattices. Node and arc sets will be formalized by algebras ALCO3-ALCO5, characterized by adsorption axioms.

In subsection 5.1. essentially binary adsorption axioms will be used to define ordinary directed graphs. Then, in subsection 5.2 directed recursive labelnode hypergraphs without contact labelnodes will be axiomatized by truly N-ary versions of the adsorption property. Finally, in subsection 5.3 DR<u>L</u>Hs with contact labelnodes will be formalized using the new "similpotence" property, capable of merging certain complex labelnodes that only differ in their contact labelnodes.

5.1 Binary/N-ary Adsorption and the Ordinary Graph Collections

To formalize a diagrammatically given directed graph like



normally a kind of relational structure G1 = (N,A) is used, where N = {a,b,c,d,e,f} is the set of nodes and A = {(a,b),(b,c),(b,d),(d,c)} is the extensionally given binary relation of arcs [a subset of NxN].

This formalization, however, has two practical [computational] disadvantages:

1. The nodes also occurring in an arc are unnecessarily symbolized twice, first in N and then again in A.

2. Graphs do not gracefully degenerate into sets, in particular a "graph" without any arcs is still a pair, (N,{}), not a set.

Perhaps these disadvantages, together with further inconveniences like the indirectness of first specifying a pair and then its component sets, contributed to the limited computational attractiveness of this formalization, so that other, non-set-oriented formalizations [and programming languages] like adjacency matrices [in ALGOL-like languages] and property lists [in LISP-like languages] were mostly used instead. Yet, these disadvantages can also be overcome in a set-oriented manner, by formalizing a graph as a single set containing both the nodes and the arcs, for the example leading to

G2 = {e,f,(a,b),(b,c),(b,d),(d,c)}

[By definition, such a "graph set" is non-homogeneous, consisting of individuals and pairs, but so is the original "graph pair", consisting of an ordinary set [set of individuals] and a relation [set of pairs].] Thus we have the following two advantages:

1'. Only the *isolated* nodes remain *as elements* in the node and arc set.

2'. If we remove arc by arc from G2, in the order they are written there, the graph becomes more and more of an ordinary set,

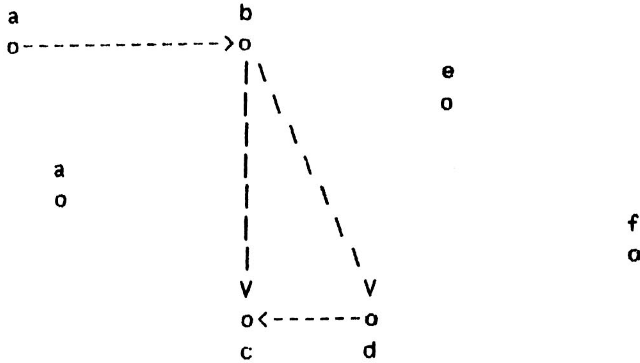
{e,f,a,b,(b,c),(b,d),(d,c)} = {a,b,e,f,(b,c),(b,d),(d,c)}
{e,f,a,b,b,c,(b,d),(d,c)} = {a,b,c,e,f,(b,d),(d,c)}
{e,f,a,b,b,c,b,d,(d,c)} = {a,b,c,d,e,f,(d,c)}
{e,f,a,b,b,c,b,d,d,c} = {a,b,c,d,e,f}

until it is just a set of individual nodes.

Note, however, that a redundancy like 1. appears in the non-final degeneration states of 2'., e.g. both the node b and the arc (b,c) occur in the first state. This phenomenon becomes even more noticeable if we try to produce G2 by just uniting the nodes and arcs of G1:

G2' = N U A = {a,b,c,d,e,f,(a,b),(b,c),(b,d),(d,c)}

Obviously, if G2 and G2' are regarded as sets only, then G2 <> G2', because of the redundant elements a-d occurring in G2' but not in G2. Yet, if G2 and G2' are to be regarded as graphs, the equality G2' = G2 should hold, because, for instance, a node like a is no longer isolated if it occurs in an arc like (a,b). Thus the diagram



should be regarded as nothing but a distorted version of the previous diagram, where both a occurrences were merged [the letters a, b, ... are names of the nodes themselves, not marks on nodes, hence they must be unique].

To obtain the desired equality formally, we regard G2 and G2' as elements of an algebra in which versions [that are binary with respect to "." and N-ary with respect to "*"] of the following two "adsorption" properties [that are binary with respect to both "." and "*"] are postulated as axioms:

adsorption1: m1*(m1.m2) = m1.m2
adsorption2: m2*(m1.m2) = m1.m2

These properties are "complementary" to the following absorption laws for lattices [since in lattices not only "*" = "UNION" but also "." = "INTERSECTION" is commutative, only one of them were needed]:

absorption1: m1*(m1.m2) = m1
absorption2: m2*(m1.m2) = m2

For example, in propositional calculus with "*" = "OR" and "." = "AND" the absorption laws specialize to

m1 OR (m1 AND m2) = m1
m2 OR (m1 AND m2) = m2

Of course, adsorption laws do not hold in propositional calculus, but -- to digress somewhat from formal algebra -- in the "belief system of common-sense value preferences" adsorption laws for AND/OR-related operators AND'/OR' could indeed be given a meaningful interpretation, as exemplified by the self-explaining "value equations"

rich OR' (rich AND' healthy) = rich AND' healthy
healthy OR' (rich AND' healthy) = rich AND' healthy

Unlike in lattices, the two operations "." and "*" are not "dual" in graphs. That is, we do not postulate adsorption axioms like m1.(m1*m2) = m1*m2 and m2.(m1*m2) = m1*m2, dual to ad1 and ad2, and complementary to the absorption laws m1.(m1*m2) = m1 and m2.(m1*m2) = m2, dual to ab1 and ab2 [hence holding in lattices]. In fact, m1.(m1*m2) and m2.(m1*m2) would yield the "error" element "\" in the formalization of ordinary graphs below, because they do not represent well-formed arcs.

Let us mention another difference between the usual graph formalization and our algebraic treatment: Normally *a* graph is defined as *a* relational structure. Although, of course, "a" stands for "any", this means that another relational structure has to be defined for each new graph instance; on the other hand, the algebra below will directly define the set of *all* graphs over a given node set, so that a graph instance is a term *in* that algebra.

We now proceed to the formal definition of the algebra ALCO3 used for directed graphs; it differs from ALCO2 in having two binary operations, "." and "*" [x.y is interpreted as the "arc pair" (x,y) and r*s as the "graph set" {r} U s].

Definition3:

```
A                     generating set [a finite set]
\ IN A                distinguished empty element [with respect to "*"]

ALCO3 = (M,.,*,-)     algebra with

M                     carrier generated by A with ".", "*", "-"

. : M x M -> M        binary operation [constructing arcs]
.(m1,m2) = m1.m2

* : M x M -> M        binary operation [constructing graphs]
*(m1,m2) = m1*m2

- : M -> M            unary operation [for negative elements]
-(m) = -m
```

Axiomatic operator restriction:

```
arcrestriction:  m1.m2 = \  if  m1 NOTIN A MINUS {\}
                             or m2 NOTIN A MINUS {\}

listrestriction: m1*m2 = \  if m2 <> n*n' and m2 <> \

flatrestriction: m1*m2 = \  if m1 = n*n' or m1 = \
```

Note the strong arcrestriction axiom over the arguments of "."-terms, which must be unnested here: If either of the arguments was not taken directly from the generating set A minus the distinguished element "\" then no legitimate arc was constructed, so that the term is identified with the empty element "\". The listrestriction axiom is the same as in the earlier collection algebras, but is now postulated for the "*"-operator and is applicable to terms whose second arguments are non-"\" atoms, negated, or arcs. Finally, the flatrestriction axiom reduces graphs with graphs [incl. "\"] as elements to "\".

Example:

A = {\, a, b}

We derive one directed graph of M:

basic binary form	N-ary/paren-sparing
\	~
a	~
b	~
(a.a)	~
(b.a)	~
(a.b)	~
(b.a)*\	~
(a.b)*((b.a)*\)	(a.b)*(b.a)*\
(a.a)*((a.b)*((b.a)*\))	(a.a)*(a.b)*(b.a)*\
b*((a.a)*((a.b)*((b.a)*\)))	b*(a.a)*(a.b)*(b.a)*\
(a.b)*(b*((a.a)*((a.b)*((b.a)*\))))	(a.b)*b*(a.a)*(a.b)*(b.a)*\

The heterogeneous counterpart ALCO3~ to the homogeneous graph algebra ALCO3 is three-sorted, partitioning the homogeneous carrier M into a carrier M. for arcs, a carrier M* for graphs [incl. atomic nodes], and a carrier M- for negated arcs and graphs.

Definition3~:

```
A.                    generating set [a singleton set]
` IN A                distinguished empty element [with respect to "."]

A*                    generating set [a finite set]
\ IN A                distinguished empty element [with respect to "*"]

A- = {}               generating set [the empty set]

ALCO3~ = (M.,M*,M-;.,*',*'',*''',-',-'')  algebra with

M., M*, M-                carriers generated by A., A*, A-
                          with ".", "*", "-"

.    : M* x M* -> M.      binary operation [constructing arcs]

*'   : M. x M* -> M*      binary operation [consing arcs to graphs]

*''  : M* x M* -> M*      binary operation [consing graphs to graphs]

*''' : M- x M* -> M*      binary operation [consing negatives to graphs]

-'   : M. -> M-           unary operation [for negative arcs]

-''  : M* -> M-           unary operation [for negative graphs]
```

Axiomatic operator restriction:

arcrestriction~:   m1.m2   = ` if m1 NOTIN A* MINUS {\}
                                  or m2 NOTIN A* MINUS {\}

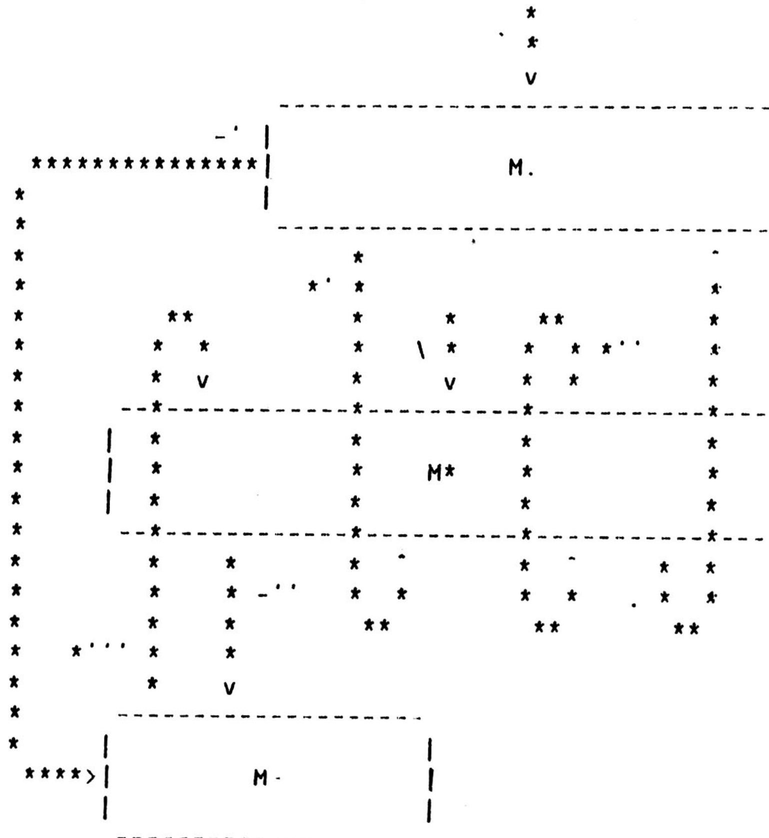
*listrestriction~:   m1*m2   = \ if m2 IN A* MINUS {\}

'emptyrestriction~: `'m   = \

''flatrestriction~: m1''m2 = \ if m1 NOTIN A* MINUS {\}

The arcrestriction~ axiom cannot yield "\" IN A* as an "error element" but must yield the newly introduced empty arc "`" IN A., because both sides of the equation m1.m2 = ` must denote elements in the same carrier. The *listrestriction~ axiom corresponds to those in ALCO2~, but is now postulated for arbitrarily primed "*"-operators, because through our priming convention it abbreviates the three axioms m1*'m2 = \ if m2 IN A* MINUS {\}, m1*''m2 = \ if m2 IN A* MINUS {\}, and m1*'''m2 = \ if m2 IN A* MINUS {\}; without utilizing the IN predicate for the condition, the axiom would have to be written m1*m2 = \ if m2 <> n*'n' and m2 <> n*''n' and m2 <> n*'''n' and m2 <> \ -- itself being still an abbreviation of three differently primed axioms. The new emptyrestriction~ axiom, *'er~, propagates empty arcs, "`", as empty nodes, "\"; the flatrestriction~ axiom, *''fr~, shortens m1*''m2 = \ if m1 = n*'n' or m1 = n*''n' or m1 = n*'''n' or m1 = \, an expanded version of the homogeneous *fr axiom.

In our diagram notation ALCO3~ can be depicted thus:



The example for the homogeneous algebra ALCO3 can be transferred to the heterogeneous algebra ALCO3~ by using the generating sets A. = {`}, A* = {\, a, b}, and A- = {}, so that the basic binary form (a.b)*'(b*''((a.a)*'((a.b)*'((b.a)*'\)))) and its N-ary/paren-sparing short form (a.b)*'b*''(a.a)*'(a.b)*'(b.a)*'\ can be derived in M*.

A considerable simplification of the many-sorted algebra ALCO3~ can be achieved by developing it into a many-sorted generator-separated algebra ALCO3~$ through a division of M* into M/ and M_: Not only do all four operator restriction axioms become unnecessary, but the ALCO3~-introduced empty arc "`" can also be removed again.

<u>Definition3~$</u>:

A. = {} generating set [the empty set of arcs]

A_ generating set [a singleton set]
\ IN A distinguished empty element [with respect to "*"]

A/ generating set [a generator]

A- = {} generating set [the empty set of negatives]

ALCO3~$ = (M.,M_,M/,M-;.,*',*/,*''',-',-/) algebra with

M., M_, M/, M- carriers generated by A., A_, A/, A-
with ".", "*'", "*/", "*'''", "-'", "-/"
[M/ = A/]

. : M/ x M/ -> M. binary operation [constructing arcs]

*' : M. x M_ -> M_ binary operation [consing arcs to graphs]

*/ : M/ x M_ -> M_ binary operation [consing isolates to graphs]

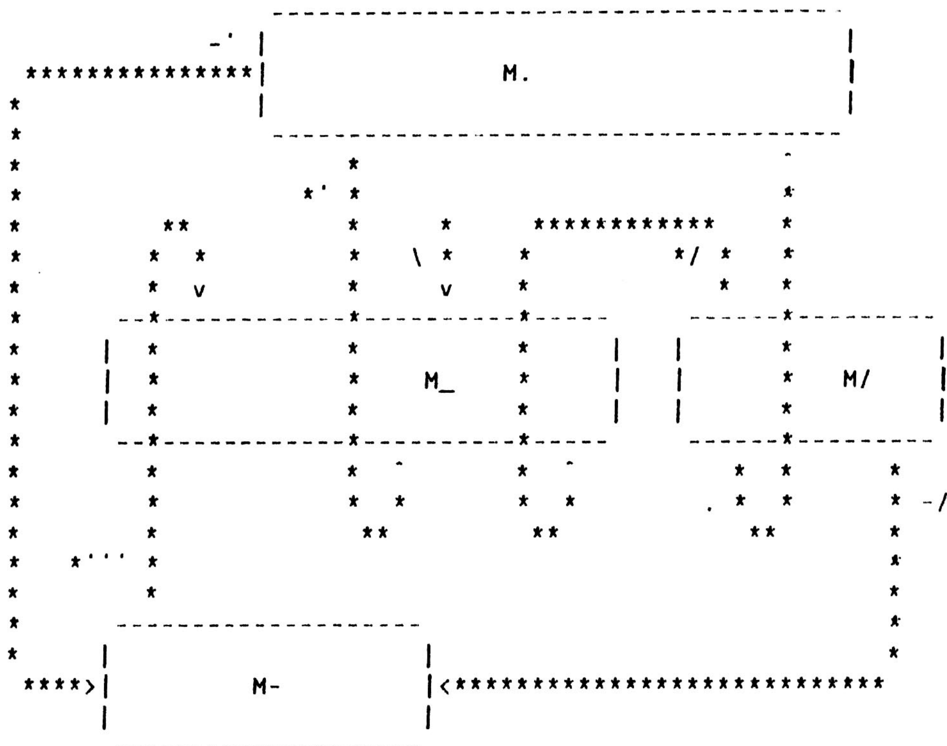
*''' : M- x M_ -> M_ binary operation [consing negatives to graphs]

-' : M. -> M- unary operation [for negative arcs]

-/ : M/ -> M- unary operation [for negative isolates]

Of course, 1. the M/-restriction of the "."-arguments makes arcrestriction, 2. the M_-restriction of the second arguments of "*'", "*/", and "*'''" makes listrestriction, and 3. the omission of "`" makes emptyrestriction unnecessary as operator restriction axioms. Observe that, for the generator separation, it was not necessary to actually divide the operator *'' into the two operators */: M/ x M_ -> M_ and *: M_ x M_ -> M_, because the latter can be omitted since its effect would be immediately undone by the flatrestriction axiom, which thus can also be omitted. Similarly, the graph negation operator "-''" of ALCO3~ was overly general, because it is only required for non-"\" atoms [though we did not bother to undo this redundant generality by an operator restriction axiom]; therefore in ALCO3~$ there is only an operator "-/", applicable to elements in M/, and no operator "-", applicable to elements in M_.

In our diagram notation ALCO3~$ can be depicted thus [hyperarcs for the generator elements of M/ are omitted here]:



There is only a trivial change in the term used as an ALCO3~ example to obtain the ALCO3~$ term (a.b)*'(b*/((a.a)*'((a.b)*'((b.a)*'\)))) in basic binary form and (a.b)*'b*/(a.a)*'(a.b)*'(b.a)*'\ in N-ary/paren-sparing short form.

The subsequent many-sorted DRLH algebras ALCO4~ and ALCO5~ could be analogously augmented to many-sorted generator-separated algebras, thus sparing the remaining operator restriction axioms. However, we will not carry this out, in order to avoid a corresponding increase in the numbers of operators.

For defining directed graphs we postulate the N-ary set axioms for "*" [i.e. *Commutativity and *Idempotence] and the following N-ary/binary Adsorption axioms for "*"/".":

```
Adsorption1: m1*((m1.m2)*m3) = (m1.m2)*m3  if m1 IN A MINUS {\}
shorter:     m1*(m1.m2)*m3   = (m1.m2)*m3  if m1 IN A MINUS {\}
Adsorption2: m2*((m1.m2)*m3) = (m1.m2)*m3  if m2 IN A MINUS {\}
shorter:     m2*(m1.m2)*m3   = (m1.m2)*m3  if m2 IN A MINUS {\}
```

The conditions on the Ad equations prevent arcs from being adsorbed by "erroneous" arc-containing arcs, which may originate as ar-derivates from "\", and afterwards could be ar-reduced to "\" again, e.g.:

```
(a.b)*\ =fr=
(a.b)*\*\ =ar=
(a.b)*((a.b).c)*\ <Ad1>
((a.b).c)*\ =ar=
\*\ =fr=
\
```

The heterogeneous Adsorption~ axioms are:

```
Adsorption1~: m1*''((m1.m2)*'m3) = (m1.m2)*'m3
shorter:      m1*''(m1.m2)*'m3   = (m1.m2)*'m3
Adsorption2~: m2*''((m1.m2)*'m3) = (m1.m2)*'m3
shorter:      m2*''(m1.m2)*'m3   = (m1.m2)*'m3
```

The conditions on the Ad equations have become implicit in the Ad~ equations, since 1. the graph-to-graph-consing operator "*''" does not permit an arc as a left argument, and 2. the remaining possibility of adsorbing complex or empty graphs from the complement of A* MINUS {\} can be permitted in Ad~ -- although the analogue is forbidden by the Ad conditions -- because terms for which this can happen can be reduced to "\" both before the adsorption [using *''fr~] and after it [using ar~ followed by *'er~], e.g.:

```
\ =*''fr~=
(a*''\)*''((a*''\).b)*'\ =Ad1~=
((a*''\).b)*'\ =ar~=
~*'\ =*'er~=
\
```

Directed graphs

ALCO3[.ar,*lr,*fr,*C,*I,Ad1,Ad2]

Directed graphs~

ALCO3~[.ar~,*lr~,*'er~,*''fr~,*'C~,*''C~,*'*''C~,*'I~,*''I~,Ad1~,Ad2~]

Note that by our abbreviation convention *lr~ stands for *'lr~, *''lr~, *'''lr~. Furthermore, *'I~ and *''I~ are the axioms obtained for, respectively, j = 1 and j = 2 from the *'j'I~ scheme in section 4.2.

The following example is a normalization proof starting with the graph derived in the previous example. In general, a [multi]graph collection is in normal form [cf. section 4.4] if no possible .ar, *lr, *fr, *I [later also *Iv, .i], or Ad application can decrease its size and no possible *C [later also .c] application can add to its lexicographic order.

```
Lem26: (a.b)*(b*((a.a)*((a.b)*((b.a)*\)))) = (a.a)*((a.b)*((b.a)*\))
Proof: (a.b)*(b*((a.a)*((a.b)*((b.a)*\)))) =*C=
       b*((a.b)*((a.a)*((a.b)*((b.a)*\)))) =Ad2=
       (a.b)*((a.a)*((a.b)*((b.a)*\))) =*C=
       (a.a)*((a.b)*((a.b)*((b.a)*\))) =*I=
       (a.a)*((a.b)*(-(a.b)*((a.b)*((b.a)*\)))) =*R1=
       (a.a)*((a.b)*(-(a.b)*((b.a)*\))) =*R2=
       (a.a)*((a.b)*((b.a)*(-(a.b)*\))) =*R3=
       (a.a)*((a.b)*((b.a)*\))
```

We can now also show the equality G2' = G2 of the introductory example [here we will use parallel transformations of non-interacting subexpressions in the parentheses-saving notation, as indicated by "|" separators between axiom names]:

```
Lem27: G2' = G2
Proof: a*b*c*d*e*f*(a.b)*(b.c)*(b.d)*(d.c)*\ =C=
       a*b*c*e*d*f*(a.b)*(b.c)*(b.d)*(d.c)*\ =C|C=
       a*b*e*c*f*d*(a.b)*(b.c)*(b.d)*(d.c)*\ =C|C|C=
       a*e*b*f*c*(a.b)*d*(b.c)*(b.d)*(d.c)*\ =C|C|C|C=
       e*a*f*b*(a.b)*c*(b.c)*d*(b.d)*(d.c)*\ =C|Ad2|Ad2|Ad2=
       e*f*a*(a.b)*(b.c)*(b.d)*(d.c)*\ =Ad1=
       e*f*(a.b)*(b.c)*(b.d)*(d.c)*\
```

The directed graph axiomatization can form a starting point for the definition of further ordinary graph types, much like the way tuples were used as the starting point for the definition of further basic collections. While the arc pairs of directed graphs are neither commutative nor idempotent, three other well-known graph types can be obtained by adding combinations of .commutativity and .idempotence to ALCO3 for dissolving arc direction and length-1 cycles, respectively [pairs of the form x.x represent [arc] cycles of length 1]. The heterogeneous formulations of these graph types will be omitted because they are analogous to that of the heterogeneous directed graphs~. Here and below we will use sample normalizations starting with the same initial expression as in the example that was used above for directed graphs, now given in parentheses-saving notation [in the normal forms derived, it is now also required that no possible .i application decrease size and no possible .c application add to lexicographic order].

Undirected graphs

ALCO3[.ar,*lr,*fr,*C,*I,Ad1,.c]

Note that the addition of .c makes one Adsorption axiom, say Ad2, superfluous [the additional .commutativity applications called for by the lack of Ad2 may sometimes be replaced by *Commutativity applications, as illustrated by the second step below].

```
(a.b)*b*(a.a)*(a.b)*(b.a)*\ =*C=
(a.b)*(a.a)*b*(a.b)*(b.a)*\ =*C=
(a.b)*(a.a)*(a.b)*b*(b.a)*\ =Ad1=
(a.b)*(a.a)*(a.b)*(b.a)*\ =.c=
(a.b)*(a.a)*(a.b)*(a.b)*\ =*C=
(a.a)*(a.b)*(a.b)*(a.b)*\ =*I=
(a.a)*(a.b)*-(a.b)*(a.b)*(a.b)*\ =*R1=
(a.a)*(a.b)*-(a.b)*(a.b)*\ =*R1=
(a.a)*(a.b)*-(a.b)*\ =*R3=
(a.a)*(a.b)*\
```

Directed graphs without length-1 cycles

```
ALCO3[.ar,*lr,*fr,*C,*I,Ad1,Ad2,.i]

(a.b)*b*(a.a)*(a.b)*(b.a)*\ =.i=
(a.b)*b*a*(a.b)*(b.a)*\ =Ad1=
(a.b)*b*(a.b)*(b.a)*\ =Ad2=
(a.b)*(a.b)*(b.a)*\ =*I=
(a.b)*-(a.b)*(a.b)*(b.a)*\ =*R1=
(a.b)*-(a.b)*(b.a)*\ =*R2=
(a.b)*(b.a)*-(a.b)*\ =*R3=
(a.b)*(b.a)*\
```

Undirected graphs without length-1 cycles

```
ALCO3[.ar,*lr,*fr,*C,*I,Ad1,.c,.i]

(a.b)*b*(a.a)*(a.b)*(b.a)*\ =.i=
(a.b)*b*a*(a.b)*(b.a)*\ =Ad1=
(a.b)*b*(a.b)*(b.a)*\ =*C=
(a.b)*(a.b)*b*(b.a)*\ =Ad1=
(a.b)*(a.b)*(b.a)*\ =.c=
(a.b)*(a.b)*(a.b)*\ =*I=
(a.b)*-(a.b)*(a.b)*(a.b)*\ =*R1=
(a.b)*-(a.b)*(a.b)*\ =*R1=
(a.b)*-(a.b)*\ =*R3=
(a.b)*\
```

To define multigraphs, which permit multiple arcs between two nodes [undirected cycles of length 2], the algebra can first be simplified: Instead of both set axioms only the bag axiom is postulated for "*" [i.e. *Commutativity], thus preventing identical arcs from merging idempotently. However, since the now omitted *Idempotence applied likewise to isolated nodes, the multigraph concept thus defined would permit not only multiple arcs but also multiple isolated nodes; this can be prevented by retaining an idempotence property for node individuals, call it "Indivpotence" [the MINUS clauses are really redundant for our *Iv uses because flatrestriction gives us *m2 = \]:

```
Indivpotence:  m1*m2 = m1*(-m1*m2)    if m1 IN A MINUS {\}
shorter:       m1*m2 = m1*-m1*m2      if m1 IN A MINUS {\}
```

*Indivpotence uses the same subordinate *Remove properties [and plays the same role for normal forms] as did *Idempotence. The following example shows how *Iv deletes a node duplicate, a, and leaves an arc duplicate, (b.c), untouched.

```
a*a*(b.c)*(b.c)*\ =*Iv=
a*-a*a*(b.c)*(b.c)*\ =*R1=
a*-a*(b.c)*(b.c)*\ =*R2=
a*(b.c)*-a*(b.c)*\ =*R2=
a*(b.c)*(b.c)*-a*\ =*R3=
a*(b.c)*(b.c)*\
```

In the heterogeneous formulation Indivpotence postulation becomes unnecessary, because in that case it is equivalent to simply omitting *'I~ and only postulating *''I~. Only the heterogeneous directed multigraphs~ will be formulated below, the remaining heterogeneous multigraphs being analogous.

Directed multigraphs

ALCO3[.ar,*lr,*fr,*C,*Iv,Ad1,Ad2]

Directed multigraphs~

ALCO3~[.ar~,*lr~,*'er~,*''fr~,*'C~,*''C~,*'*''C~,*''I~,Ad1~,Ad2~]

(a.b)*b*(a.a)*(a.b)*(b.a)*\ =*C=
b*(a.b)*(a.a)*(a.b)*(b.a)*\ =Ad2=
(a.b)*(a.a)*(a.b)*(b.a)*\ =*C=
(a.a)*(a.b)*(a.b)*(b.a)*\

Undirected multigraphs

ALCO3[.ar,*lr,*fr,*C,*Iv,Ad1,.c]

Again, the addition of .c makes one Adsorption axiom, say Ad2, superfluous.

(a.b)*b*(a.a)*(a.b)*(b.a)*\ =*C=
(a.b)*(a.a)*b*(a.b)*(b.a)*\ =*C=
(a.b)*(a.a)*(a.b)*b*(b.a)*\ =Ad1=
(a.b)*(a.a)*(a.b)*(b.a)*\ =.c=
(a.b)*(a.a)*(a.b)*(a.b)*\ =*C=
(a.a)*(a.b)*(a.b)*(a.b)*\

Directed multigraphs without length-1 cycles

ALCO3[.ar,*lr,*fr,*C,*Iv,Ad1,Ad2,.i]

(a.b)*b*(a.a)*(a.b)*(b.a)*\ =.i=
(a.b)*b*a*(a.b)*(b.a)*\ =Ad1=
(a.b)*b*(a.b)*(b.a)*\ =Ad2=
(a.b)*(a.b)*(b.a)*\

Undirected multigraphs without length-1 cycles

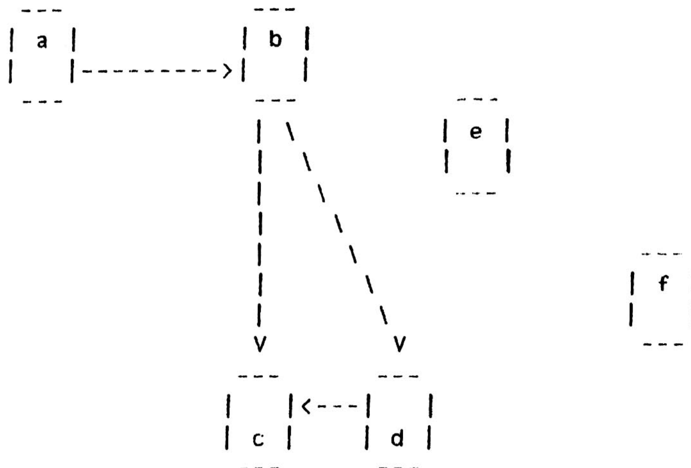
ALCO3[.ar,*lr,*fr,*C,*Iv,Ad1,.c,.i]

(a.b)*b*(a.a)*(a.b)*(b.a)*\ =.i=
(a.b)*b*a*(a.b)*(b.a)*\ =Ad1=
(a.b)*b*(a.b)*(b.a)*\ =*C=
(a.b)*(a.b)*b*(b.a)*\ =Ad1=
(a.b)*(a.b)*(b.a)*\ =.c=
(a.b)*(a.b)*(a.b)*\

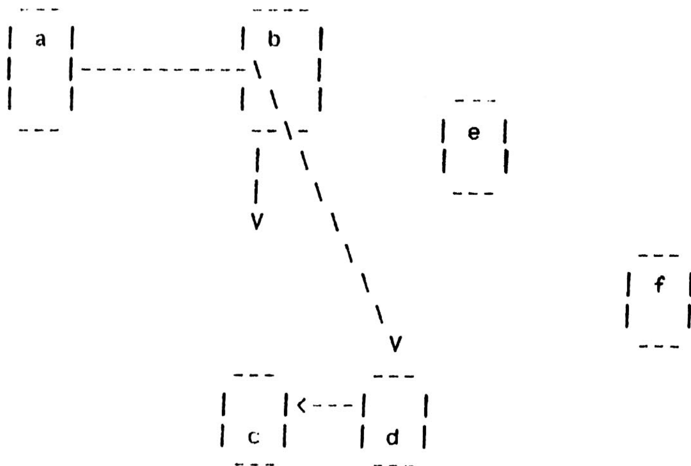
5.2 N-ary/N-ary ADsorption and DRLHs without Contact Labelnodes

Starting with the introductory directed graph example of section 5.1, a directed recursive labelnode hypergraph [DRLH] example can be developed as follows:

In a preparatory step, simplifying hyperarc cuts and assimilating atomic with complex nodes, the nodes are redrawn from small circles with attached names to boxes containing the names [this diagrammatic modification does not change anything graph-theoretically]:



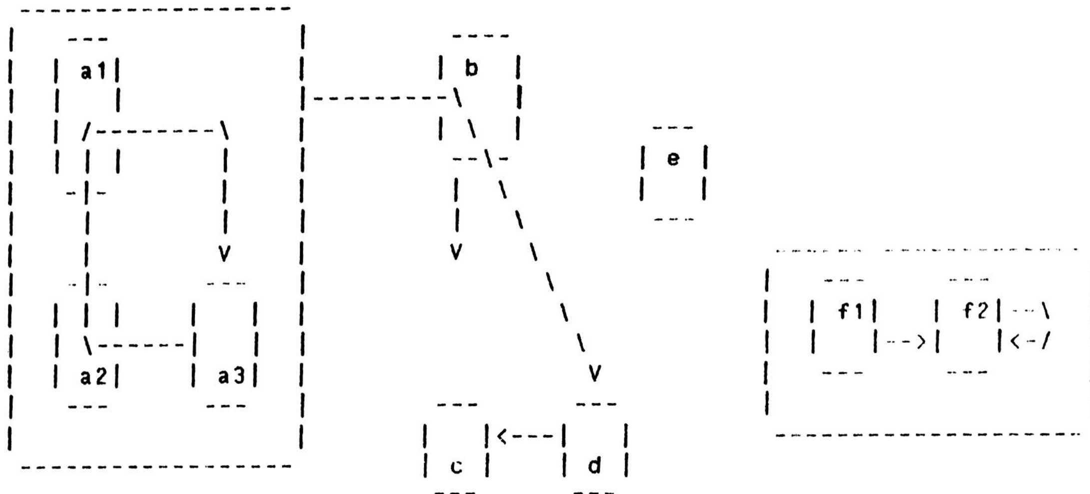
Proceeding to directed hypergraphs, some of the previous arcs, which always connected exactly two nodes, are changed to hyperarcs, which may connect an arbitrary finite number of nodes [the directed path consisting of the arcs (a,b) and (b,d) is "contracted" to a directed hyperarc (a,b,d), whose arrow cuts node b; the arc (b,c) is "shortened" to a hyperarc (b), whose arrow does not lead to another node]:



[These directed hypergraphs are unlabeled versions of the directed labeled hypergraphs used throughout this paper to depict heterogeneous algebras.]

Coming to directed recursive hypergraphs [without contact labelnodes], some of the previous nodes, which always were atomic [unstructured],

are expanded to complex nodes, which have an internal [recursive] graph structure [the atomic node a is expanded to a complex node {(a3,a2,a1,a3)}, containing the single hyperarc (a3,a2,a1,a3), and the isolated atomic node f is expanded to an isolated complex node {(f1,f2),(f2,f2)}, containing the arcs (f1,f2) and (f2,f2)]:



The final change to directed recursive labelnode hypergraphs [without contact labelnodes] is just a change in interpretation: While previously all types of graphs were unlabeled, i.e. had no labels on arcs, we now interpret the first element of a hyperarc as functioning as its label, only the remaining ones functioning as its nodes. Since a single element, such as d above, can function as a label in one hyperarc, here in (d,c), and as a node in another hyperarc, here in ({(a3,a2,a1,a3)},b,d), we refer to labels and nodes collectively as "labelnodes" [in this interpretation, {(a3,a2,a1,a3)} is a complex labelnode functioning as the label of an arc with atomic nodes b and d, and b also functions as the label in a hyperarc without any nodes].

Using FIT's collection notation, a complex labelnode [including an entire DRLH] is written as a DRLH expression and a hyperarc as a TUPLE expression. For the example we get the nested collection shown below to the right of its set notation.

```
{e,                      (DRLH e
 {(f1,f2),                     (DRLH (TUPLE f1 f2)
  (f2,f2)},                          (TUPLE f2 f2))
 (b),                          (TUPLE b)
 (d,c),                        (TUPLE d c)
 ({(a3,a2,a1,a3)},             (TUPLE (DRLH (TUPLE a3 a2 a1 a3))
  b,                                  b
  d)}                                 d))
```

In our algebraic [parentheses-saving] notation complex labelnodes become N-ary "*"-expressions and hyperarcs become N-ary "."-expressions. For the example we will get the expression

```
e*((f1.f2.`)*(f2.f2.`)*\)*(b.`)*(d.c.`)*(((a3.a2.a1.a3.`)*\).b.d.`)*\
```

Now we have an equality problem similar to the one we had with ordinary graphs: The above expression should be equal to, say [the vertical layout is necessary because this expression does not fit into one line any longer]

```
 ((f1.f2.`)*(f2.f2.`)*\)
*e
*((f1.f2.`)*(f2.f2.`)*\)
*c
*(d.c.`)
*(b.`)
*(d.c.`)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*((a3.a2.a1.a3.`)*\)
*\
```

The problem can again be solved by an algebra providing axiomatic equalities for transforming such expressions into each other. Besides the usual "-"-auxiliary [for Idempotence] an auxiliary "!"-operation will be used [for ADsorption], where terms with a top-level "!" are irreducible like corresponding "-"-terms, so that the "!"-operation can be regarded as being "hidden" too.

Definition4:

```
A                      generating set [a finite set]
` IN A                 distinguished empty element [with respect to "."]
\ IN A                 distinguished empty element [with respect to "*"]

ALC04 = (M,.,*,-,!)    algebra with

M                      carrier generated by A with ".", "*", "-", "!"

. : M x M -> M         binary operation [constructing hyperarcs]
.(m1,m2) = m1.m2

* : M x M -> M         binary operation [constructing complex labelnodes]
*(m1,m2) = m1*m2

- : M -> M             unary operation [for negative elements]
-(m) = -m

! : M -> M             unary operation [for embedding elements]
!(m) = !m
```

Axiomatic operator restriction:

```
.listrestriction: m1.m2 = `   if m2 <> n.n' and m2 <> `

.flatrestriction: m1.m2 = `   if m1 = n.n' or m1 = `

*listrestriction: m1*m2 = \   if m2 <> n*n' and m2 <> \
```

Both listrestriction axioms reduce terms whose second arguments are atoms, negated, or "!"-marked, and additionally .lr applies to second arguments which are complex or empty labelnodes, while *lr applies to those which are non-empty or empty hyperarcs. Furthermore, the flatrestriction axiom is postulated for the hyperarc operator "." here,

forbidding nested hyperarcs, whereas in ALCO3 it was postulated for the graph operator "*", forbidding nested graphs, which are allowed here. Thus the <u>introduction</u> of recursively nested graphs in DR<u>L</u>Hs is achieved algebraically by simply <u>omitting</u> a flatrestriction axiom for "*". Because of the auxiliary status of "-" and "!" we have not bothered to postulate the restriction axioms -m1.m2 = ` and !m1.m2 = ` for hyperarcs having negated or "!"-marked left arguments, hence being meaningless for DR<u>L</u>Hs [although, e.g., negated elements would be useful in undirected hyperarcs as used in Berge's hypergraphs (Berge 1970)]; for the same reason we have not postulated !m = \ if m <> n.n' as a restriction axiom for "!"-arguments which are not "."-terms.

<u>Example</u>:

A = {`, \, a, b}

We derive one complex labelnode of M:

```
basic binary form                  |   N-ary/paren-sparing
-----------------------------------+-------------------------------
`                                  |   ~
\                                  |   ~
a                                  |   ~
b                                  |   ~
     b.`          a.`              |   ~
  a.(b.`)       (a.`)*\            |     a.b.`     (a.`)*\
  a.(b.`)    a*((a.`)*\)           |     a.b.`   a*(a.`)*\
(a.(b.`))*(a*((a.`)*\))            |   (a.b.`)*a*(a.`)*\
```

The heterogeneous counterpart ALCO4~ to the homogeneous DR<u>L</u>H algebra ALCO4 is four-sorted, partitioning the homogeneous carrier M into a carrier M. for hyperarcs, a carrier M* for DR<u>L</u>Hs or labelnodes [incl. atomic labelnodes], a carrier M- for negated terms, and a carrier M! for "!"-prefixed hyperarcs.

Definition4~:

```
A.                    generating set [a singleton set]
` IN A                distinguished empty element [with respect to "."]

A*                    generating set [a finite set]
\ IN A                distinguished empty element [with respect to "*"]

A- = {}               generating set [the empty set of negatives]

A! = {}               generating set [the empty set of embeds]

ALCO4~ = (M.,M*,M-,M!;.,*',*'',*''',*'''',-',-'',-'''',!)  algebra with

M., M*, M-, M!              carriers generated by A., A*, A-, A!
                            with ".", "*", "-", "!"

.     : M* x M. -> M.       binary operation [consing DRLHs to hyperarcs]

*'    : M. x M* -> M*       binary operation [consing hyperarcs to DRLHs]

*''   : M* x M* -> M*       binary operation [consing DRLHs to DRLHs]

*'''  : M- x M* -> M*       binary operation [consing negatives to DRLHs]

*'''' : M! x M* -> M*       binary operation [consing embeds to DRLHs]

-'    : M. -> M-            unary operation [for negative hyperarcs]

-''   : M* -> M-            unary operation [for negative DRLHs]

-'''' : M! -> M-            unary operation [for negative embeds]

!     : M. -> M!            unary operation [for embedding hyperarcs]
```

Axiomatic operator restriction:

`*listrestriction~: m1*m2 = \   if m2 IN A* MINUS {\}`

The omission of an operator "-'''" from ALCO4~ reflects the fact that double negations -''': M- -> M- are prohibited and keeps the numbers of primes on "*" and "-" consistent, which simplifies the formulation of [Idempotence] axiom schemes.

The .listrestriction axiom of the homogeneous version is superfluous here because the second "."-argument must conform to the sort M., which only consists of (n.n')-terms and "`"; the *listrestriction~ axiom use abbreviates four axioms for the "*"-primings. No flatrestriction axiom is needed here, because the M* sort of the hyperarc operator "." does not allow nested hyperarcs in any case. The use of the M* sort in "." also makes restriction axioms like -m1.m2 = ` and !m1.m2 = ` implicit.

In our diagram notation ALCO4~ can be depicted thus:

```
                                  *         **
                                ` *        *  *
                                  v        v  *
                 -----------------------------*---
              -'|                             *   | !
 ***************|               M.            *   |*************
*               |                             *   |             *
*                -----------------------------*---              *
*                      *                      *                 *
*                    *' *                      *                 *
*         **           *     *     **         *      **         *
*        *  *          *   \ *    *  * *''    * .   *  *        *
*        *  v          *     v    *  *        *     v  *        *
*      --*-------------*----------*--------------------*--      *
*     |  *             *          *                    *   |    *
*     |  *             *   M*     *                    *   |    *
*     |  *             *          *                    *   |    *
*      --*-------------*----------*--------------------*--      *
*        *    *        *  `       *  `                 *        *
*        *    * -''    *  *       *  *                 *        *
*        *    *         **         **                  *        *
*   *''' *    *                                  *'''' *        *
*        *    v                                        *        *
*      ---------------------           --------------------     *
*     |                     |   -'''' |                    |    *
 ****>|         M-          |<********|         M!         |<****
      |                     |         |                    |
       ---------------------           --------------------
```

The example for the homogeneous algebra ALCO4 can be transferred to the heterogeneous algebra ALCO4~ by using the generating sets A. = {`}, A* = {\, a, b}, A- = {}, and A! = {}, so that the basic binary form (a.(b.`))*'(a*''((a.`)*'\)) and its N-ary/paren-sparing short form (a.b.`)*'a*''(a.`)*'\ can be derived in M*.

As another example, let us consider an instance of self-representation. Since directed hypergraphs can be represented as directed [recursive] labelnode hypergraphs in which the labels and nodes happen to form two disjoint sets [which both happen to contain atomic elements only], the above ALCO4~ diagram can be represented as a term of the DRLH algebra ALCO4~ itself. Using the generating sets [A* contains the names of carriers, distinguished elements, and operators as quoted elements to mark them off as atoms] A. = {`} and A* = {\, "M.", "M*", "M-", "M!", ".", "*'", "*''", "*'''", "*''''", "-'", "-''", "-''''", "!", "`", "\"} we get

```
  ("`"."M.".`)
*'("-'"."M."."M-".`)
*'("*'"."M."."M*"."M*".`)
*'("!"."M."."M!".`)
*'("\"."M*".`)
*'("-''"."M*"."M-".`)
*'("*''"."M*"."M*"."M*".`)
*'("."."M*"."M."."M.".`)
*'("*'''"."M-"."M*"."M*".`)
*'("-''''"."M!"."M-".`)
*'("*''''"."M!"."M*"."M*".`)
*'\
```

For DRLHs a further generalized adsorption property, ADsorption, N-ary/N-ary with respect to "*"/"." [as indicated by capitalizing two initial letters], and an associated property, Embedd, are postulated for ".", "*", and "!". The adsorption technique of AD is completely different from the one of Ad. A hyperarc (a1,...,aI,...,aN) adsorbs a labelnode aI by generating a "!"-prefixed copy !(a1,...,aI,...,aN) of itself [AD], whose labelnodes are then [totally or partially] embedded into the DRLH [Em], enabling Idempotence to remove aI; after this, the embedding and copying steps are reversed:

```
{...,(a1,...,aI,...,aN),...,aI,...} =AD=
{...,(a1,...,aI,...,aN),!(a1,...,aI,...aN),...,aI,...} =Em=
{...,(a1,...,aI,...,aN),a1,...,aI,...aN,...,aI,...} =I=
{...,(a1,...,aI,...,aN),a1,...,aI,-aI,...aN,...,aI,...} =R=
{...,(a1,...,aI,...,aN),a1,...,aI,...aN,.......} =Em=
{...,(a1,...,aI,...,aN),!(a1,...,aI,...aN),.......} =AD=
{...,(a1,...,aI,...,aN),.......}
```

Algebraically, the new transformations of this procedure are realized by the axioms below. Because of their above foreward and backward use, required even in "directed" normalizations, AD and Em -- unlike our other axiomatic equations -- get a truly bidirectional character.

```
ADsorption: (m1.m2)*m3  = (m1.m2)*(!(m1.m2)*m3)
shorter:    (m1.m2)*m3  = (m1.m2)*!(m1.m2)*m3

Embedd1: !(m1.m2)*m3 = m1*(!m2*m3)   if m1 <> n.n' and m1 <> `
shorter: !(m1.m2)*m3 = m1*!m2*m3     if m1 <> n.n' and m1 <> `
Embedd2: !(m1.`)*m2  = m1*m2         if m1 <> n.n' and m1 <> `
```

The condition on the Em equations prevents an "erroneous" hyperarc with an embedded hyperarc from setting free its inner hyperarc, which could result, for example, in an unwanted idempotent removal of other hyperarcs, before the "erroneous" hyperarc part becomes "`"-reduced by the .flatrestriction axiom [applicable inside a hyperarc], by which it may also have been generated:

```
(a.`)*(b.c.`)*\ =.fr=
(a.(b.c.`).d.`)*(b.c.`)*\ =AD=
(a.(b.c.`).d.`)*!(a.(b.c.`).d.`)*(b.c.`)*\ =Em1=
(a.(b.c.`).d.`)*a*!((b.c.`).d.`)*(b.c.`)*\ <Em1>
(a.(b.c.`).d.`)*a*(b.c.`)*!(d.`)*(b.c.`)*\ =*I=
(a.(b.c.`).d.`)*a*(b.c.`)*-(b.c.`)*!(d.`)*(b.c.`)*\ =*R2=
(a.(b.c.`).d.`)*a*(b.c.`)*!(d.`)*-(b.c.`)*(b.c.`)*\ =*R1=
(a.(b.c.`).d.`)*a*(b.c.`)*!(d.`)*-(b.c.`)*\ =*R3=
(a.(b.c.`).d.`)*a*(b.c.`)*!(d.`)*\ <Em1>
(a.(b.c.`).d.`)*a*!((b.c.`).d.`)*\ =Em1=
(a.(b.c.`).d.`)*!(a.(b.c.`).d.`)*\ =AD=
(a.(b.c.`).d.`)*\ =.fr=
(a.`)*\
```

Not prevented is an Em1-application to a term like !(b.a)*x, because Em1 does not test the produced binding m2 = a with a <> n.n' and a <> ` [through which the .listrestriction axiom would have been applicable to (b.a), even before it became an "!"-argument], so that an "erroneous" term like b*!a*x is yielded; but then neither AD nor one of the Em equations [except Em1 backward] is applicable to the term b*!a*x or to its subterm !a*x, because these equations expect the "!"-operator to

have "."-concatenation arguments; since no other equation is applicable to a "!"-containing term either, such spurious transformations of "erroneous" terms are harmless.

The heterogeneous formulation of these axioms is as follows:

```
ADsorption~: (m1.m2)*'m3   = (m1.m2)*'(!(m1.m2)*''''m3)
shorter:     (m1.m2)*'m3   = (m1.m2)*'!(m1.m2)*''''m3

Embedd1~: !(m1.m2)*''''m3 = m1*''(!m2*''''m3)
shorter:  !(m1.m2)*''''m3 = m1*''!m2*''''m3
Embedd2~: !(m1.`)*''''m2  = m1*''m2
```

The condition on the Em equations becomes superfluous for Em~, because the sorted "."-operator in ALC04~ guarantees that m1 cannot be a hyperarc.

Directed recursive labelnode hypergraphs without contact labelnodes

ALC04[.lr,.fr,*lr,*C,*I,AD]

Directed recursive labelnode hypergraphs without contact labelnodes~

ALC04~[*lr~,*'C~,*''C~,*'*''C~,*'I~,*''I~,*''''I~,AD~]

Without loss of generality, *''''C~, *'*''''C~, and *''*''''C~, implicit in *C, are omitted here, because the commuting of embeds is never required; however, *''''I~, implicit in *I, is included, because idempotent removals of embeds -- though never strictly necessary -- shorten certain proofs.

The following example shows the normalization of the DRLH derived in the previous example.

```
Lem28:  (a.(b.`))*(a*((a.`)*\)) = (a.`)*((a.(b.`))*\)

Proof1: (a.(b.`))*(a*((a.`)*\)) =AD=
        (a.(b.`))*(!(a.(b.`))*(a*((a.`)*\))) =Em1=
        (a.(b.`))*(a*(!(b.`)*(a*((a.`)*\)))) =Em2=
        (a.(b.`))*(a*(b*(a*((a.`)*\)))) =*I=
        (a.(b.`))*(a*(-a*(b*(a*((a.`)*\))))) =*R2=
        (a.(b.`))*(a*(b*(-a*(a*((a.`)*\))))) =*R1=
        (a.(b.`))*(a*(b*(-a*((a.`)*\)))) =*R2=
        (a.(b.`))*(a*(b*((a.`)*(-a*\)))) =*R3=
        (a.(b.`))*(a*(b*((a.`)*\))) =Em2=
        (a.(b.`))*(a*(!(b.`)*((a.`)*\))) =Em1=
        (a.(b.`))*(!(a.(b.`))*((a.`)*\)) =AD=
        (a.(b.`))*((a.`)*\) =*C=
        (a.`)*((a.(b.`))*\)
```

Note the reverse order of the AD/Em uses in their second application sequence, and the resulting right-to-left reading of their equations [an ADsorption-provided "!"-copy of the hyperarc a.b.` Embedds its labelnodes into the surrounding complex labelnode, so that a can remove its duplicate idempotently, and after that !(a.b.`) is reconstructed from its labelnodes and is adsorbed again by the hyperarc original].

Since R2 can send -a over !(b.`) as well as over b, the first Em2 use and its reversal through the second Em2 use [i.e. the total embedding of the hyperarc a.b.`] is unnecessary. But the resulting proof would still need two states more than the following 10-state proof embedding the hyperarc a.` instead of a.b.` [we now use the parentheses-saving notation]:

```
Proof2: (a.b.`)*a*(a.`)*\ =*C=
          (a.b.`)*(a.`)*a*\ =AD=
          (a.b.`)*(a.`)*!(a.`)*a*\ =Em2=
          (a.b.`)*(a.`)*a*a*\ =*I=
          (a.b.`)*(a.`)*a*-a*a*\ =R1=
          (a.b.`)*(a.`)*a*-a*\ =R3=
        (a.b.`)*(a.`)*a*\ =Em2=
        (a.b.`)*(a.`)*!(a.`)*\ =AD=
        (a.b.`)*(a.`)*\ =*C=
        (a.`)*(a.b.`)*\
```

Note that the indented states 2 to 6 form a cycle, which shows that an even shorter 5-state proof is obtainable by merely omitting these indented states [the isolated labelnode a need not be removed idempotently but can directly be used as the embedded version of the hyperarc a.`].

The introductory equality problem can now be solved by the normalization chain below [parallel transformations on the parentheses-saving form will be used]. A DRLH is in normal form if no possible application of .lr, .fr, *lr, *I, and AD [incl. Em] can decrease its size and no possible *C application can add to its lexicographic order [in which all isolated labelnodes, including complex ones, precede all hyperarcs].

```
 ((f1.f2.`)*(f2.f2.`)*\)
*e
*((f1.f2.`)*(f2.f2.`)*\)
*c
*(d.c.`)
*(b.`)
*(d.c.`)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*((a3.a2.a1.a3.`)*\)
*\

=*C|*I|AD=

 e
*((f1.f2.`)*(f2.f2.`)*\)
*((f1.f2.`)*(f2.f2.`)*\)
*c
*(d.c.`)
*-(d.c.`)
*(b.`)
*(d.c.`)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*!(((a3.a2.a1.a3.`)*\).b.d.`)
*((a3.a2.a1.a3.`)*\)
*\
```

```
=*I|*C|*R2|Em1=

 e
*((f1.f2.`)*(f2.f2.`)*\)
*-((f1.f2.`)*(f2.f2.`)*\)
*((f1.f2.`)*(f2.f2.`)*\)
*(d.c.`)
*c
*(b.`)
*-(d.c.`)
*(d.c.`)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*((a3.a2.a1.a3.`)*\)
*!(b.d.`)
*((a3.a2.a1.a3.`)*\)
*\

=*R1|AD|*R1|*I=

 e
*((f1.f2.`)*(f2.f2.`)*\)
*-((f1.f2.`)*(f2.f2.`)*\)
*(d.c.`)
*!(d.c.`)
*c
*(b.`)
*-(d.c.`)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*((a3.a2.a1.a3.`)*\)
*-((a3.a2.a1.a3.`)*\)
*!(b.d.`)
*((a3.a2.a1.a3.`)*\)
*\

=*R2|Em1|*R2|*R2=

 e
*((f1.f2.`)*(f2.f2.`)*\)
*(d.c.`)
*-((f1.f2.`)*(f2.f2.`)*\)
*d
*!(c.`)
*c
*(b.`)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*-(d.c.`)
*((a3.a2.a1.a3.`)*\)
*!(b.d.`)
*-((a3.a2.a1.a3.`)*\)
*((a3.a2.a1.a3.`)*\)
*\
```

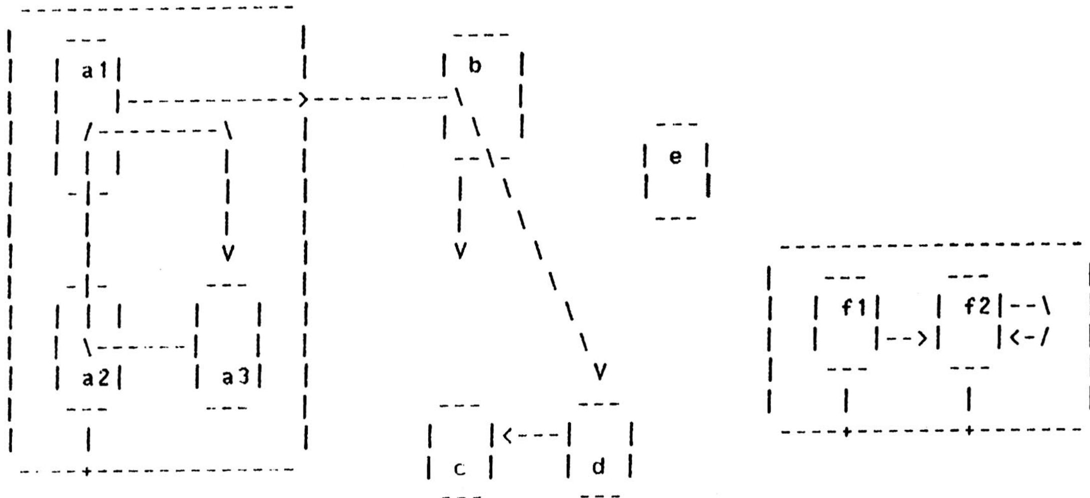
```
=*R2|Em2|Em1|*R1=

 e
*((f1.f2.`)*(f2.f2.`)*\)
*(d.c.`)
*d
*-((f1.f2.`)*(f2.f2.`)*\)
*c
*c
*(b.`)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*-(d.c.`)
*!(((a3.a2.a1.a3.`)*\).b.d.`)
*-((a3.a2.a1.a3.`)*\)
*\

=*R2|*R2|*R3=

 e
*((f1.f2.`)*(f2.f2.`)*\)
*(d.c.`)
*d
*c
*-((f1.f2.`)*(f2.f2.`)*\)
*c
*(b.`)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*!(((a3.a2.a1.a3.`)*\).b.d.`)
*-(d.c.`)
*\

=*I|*R2|AD|*R3=

 e
*((f1.f2.`)*(f2.f2.`)*\)
*(d.c.`)
*d
*c
*-c
*c
*-((f1.f2.`)*(f2.f2.`)*\)
*(b.`)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*\

=Em2|*R1|*R2=

 e
*((f1.f2.`)*(f2.f2.`)*\)
*(d.c.`)
*d
*!(c.`)
*-c
*(b.`)
*-((f1.f2.`)*(f2.f2.`)*\)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*\
```

```
=Em1|*R2|*R2=

 e
*((f1.f2.`)*(f2.f2.`)*\)
*(d.c.`)
*!(d.c.`)
*(b.`)
*-c
*(((a3.a2.a1.a3.`)*\).b.d.`)
*-((f1.f2.`)*(f2.f2.`)*\)
*\

=AD|*R2|*R3=

 e
*((f1.f2.`)*(f2.f2.`)*\)
*(d.c.`)
*(b.`)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*-c
*\

=*C|*R3=

 e
*((f1.f2.`)*(f2.f2.`)*\)
*(b.`)
*(d.c.`)
*(((a3.a2.a1.a3.`)*\).b.d.`)
*\
```

This chain could be shortened by reversing the foreward movement of the negated elements -((f1.f2.`)*(f2.f2.`)*\) and -c as soon as there is nothing more for them to remove, letting them reunite with their nearby unnegated versions [reading *I backward] instead of making them traverse the entire "*"-expression [until *R3 becomes applicable]; however, as indicated in section 4.2, such a procedure could not be used for mechanical normalization because the complete traversal serves the very purpose of systematically checking whether "there is nothing more to remove".

5.3 Similpotence and DRLHs with Contact Labelnodes

The complex labelnodes introduced in section 5.2 could participate in [hyper]arcs only as monolithic entities; there was no interface between the inner graph structure of a complex labelnode and the surrounding graph structure. We now distinguish one or more labelnodes inside a complex labelnode as its "contact labelnodes", permitting [hyper]arcs which use that complex labelnode to access its interior via some of these contact places. For example, in the introductory DRLH of section 5.2 the labelnodes a1 and a2 may be distinguished as contact labelnodes of the complex labelnode {(a3,a2,a1,a3)}, where a1 is used as the internal starting point of the arc leaving that complex label for the nodes b and d, while a2 is not actually used by any hyperarc; similarly the labelnodes f1 and f2 may be distinguished as contact labelnodes of the isolated complex labelnode {(f1,f2),(f2,f2)}:



The designation of contact labelnodes, which is done here diagrammatically by a line between the distinguished labelnode and the complex labelnode's boundary, can be done symbolically by [additionally] using the distinguished labelnode as an isolated labelnode with a special prefix, say "@". Thus the version of {(a3,a2,a1,a3)} used as a complex label can be represented as {@a1,(a3,a2,a1,a3)}, the unused version of this complex labelnode as {@a2,(a3,a2,a1,a3)}, and the isolated complex labelnode as {@f1,@f2,(f1,f2),(f2,f2)}.

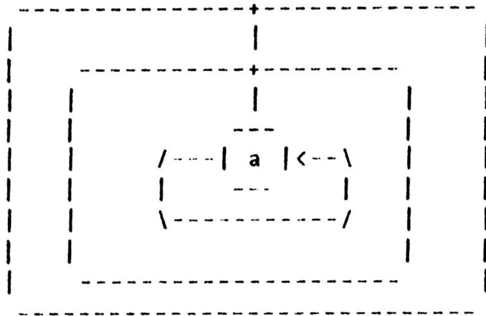
Using the collection notation in FIT [where "@" has another meaning], an "@"-prefixed contact labelnode @x can be rewritten as an explicit CONTACT expression (CONTACT x). For the example we thus get the nested collection shown below to the right of its set notation.

```
{e,                           (DRLH e
 {@a2,                              (DRLH (CONTACT a2)
  (a3,a2,a1,a3)},                         (TUPLE a3 a2 a1 a3))
 {@f1,                              (DRLH (CONTACT f1)
  @f2,                                    (CONTACT f2)
  (f1,f2),                                (TUPLE f1 f2)
  (f2,f2)},                               (TUPLE f2 f2))
 (b),                               (TUPLE b)
 (d,c),                             (TUPLE d c)
 ({@a1,                             (TUPLE (DRLH (CONTACT a1)
   (a3,a2,a1,a3)},                               (TUPLE a3 a2 a1 a3))
  b,                                       b
  d)}                                      d))
```

In our algebraic notation "@" becomes a unary operator and contact labelnodes become "@"-applications used as left arguments of "*"-applications. For the example we get the expression

```
 e
*(@a2*(a3.a2.a1.a3.`)*\)
*(@f1*@f2*(f1.f2.`)*(f2.f2.`)*\)
*(b.`)
*(d.c.`)
*((@a1*(a3.a2.a1.a3.`)*\).b.d.`)
*\
```

If a contact labelnode of a complex labelnode is itself a complex labelnode, this complex contact labelnode may again have its own contact labelnodes and so on, to any finite depth. For example,



is the diagram of a complex labelnode with the symbolic representation {∂{∂a,(a,a)}}, whose single isolated labelnode element is the complex contact labelnode {∂a,(a,a)}, which itself contains the atomic contact labelnode a and a single arc (a,a). An algebraic expression for the above diagram is ∂(∂a*(a.a.`)*\)*\.

DR<u>L</u>Hs with contact labelnodes can now also be formalized algebraically. Besides the auxiliaries "-" [for Idempotence] and "!" [for ADsorption] of ALCO4 an auxiliary operation "#" [for Similpotence] will be used, where terms with a top-level "#" are again irreducible and "#" can be regarded as another "hidden" operation.

<u>Definition5</u>:

A	generating set [a finite set]
` IN A	distinguished empty element [with respect to "."]
\ IN A	distinguished empty element [with respect to "*"]
ALCO5 = (M,.,*,#,-,!,∂)	algebra with
M	carrier generated by A with ".", "*", "#", "-", "!", "∂"
. : M x M -> M .(m1,m2) = m1.m2	binary operation [constructing hyperarcs]
* : M x M -> M *(m1,m2) = m1*m2	binary operation [constructing complex labelnodes]
# : M x M -> M #(m1,m2) = m1#m2	binary operation [merging complex labelnodes]
- : M -> M -(m) = -m	unary operation [for negative elements]
! : M -> M !(m) = !m	unary operation [for embedding elements]
∂ : M -> M ∂(m) = ∂m	unary operation [for contact labelnodes]

Axiomatic operator restriction:

```
.listrestriction:      m1.m2 = `   if m2 <> n.n' and m2 <> `

.flatrestriction:      m1.m2 = `   if m1 = n.n' or m1 = `

hypactrestriction:    @m1.m2 = `

*listrestriction:      m1*m2 = \   if m2 <> n*n' and m2 <> \

mergerestriction:      m1#m2 = \   if  (m1 <> \ and m1 <> n*n')
                                    or (m2 <> \ and m2 <> n*n')

contactrestriction:       @m = \   if m NOTIN A MINUS {`} and m <> n*n'
```

The .lr, .fr, and *lr axioms work analogously to those of ALC04, with the lr axioms now also reducing terms containing "#"/"@"-terms as second arguments. The new hypactrestriction axiom prevents hyperarcs from containing contact labelnodes [because only complex labelnodes may contain contact labelnodes and "@" -- unlike "-" and "!" -- does not have an auxiliary status], the new mergerestriction axiom forces both "#"-arguments to be empty or complex labelnodes, and the new contactrestriction axiom demands that the "@"-argument be any kind of labelnode.

Example:

A = {`, \, a, b}

We derive one complex labelnode with contact labelnodes of M:

```
basic binary form                                | N-ary/paren-sparing
-------------------------------------------------+------------------------------
\                                                | ~
a                                                | ~
b                                                | ~
@a                                               | ~
@b                                               | ~
          b*\                 a*\                | ~
     @a*(b*\)            @b*(a*\)                |     @a*b*\       @b*a*\
     @a*(b*\)           (@b*(a*\))*\             |     @a*b*\      (@b*a*\)*\
     @a*(b*\)      b*((@b*(a*\))*\)              |     @a*b*\   b*(@b*a*\)*\
    (@a*(b*\))*(b*((@b*(a*\))*\))                |    (@a*b*\)*b*(@b*a*\)*\
@b*((@a*(b*\))*(b*((@b*(a*\))*\)))               |  @b*(@a*b*\)*b*(@b*a*\)*\
```

Note that the derived complex labelnode happens to contain no hyperarcs, i.e. it is degenerated into an example of "sets with contact elements" [the embedded isolated complex labelnodes are degenerated to set elements which are sets themselves].

The heterogeneous counterpart ALC05~ to the homogeneous DRLH algebra ALC05 is six-sorted, partitioning the homogeneous carrier M into the four carriers of ALC04~ and further carriers M# and M@ for merging terms and contact labelnodes, respectively.

Definition5~:

```
A.                      generating set [a singleton set]
` IN A                  distinguished empty element [with respect to "."]

A*                      generating set [a finite set]
\ IN A                  distinguished empty element [with respect to "*"]

A- = {}                 generating set [the empty set of negatives]

A! = {}                 generating set [the empty set of embeds]

A# = {}                 generating set [the empty set of mergings]

A@ = {}                 generating set [the empty set of contacts]

ALCO5~ = (M.,M*,M-,M!,M#,M@;.,*',*'',*''',*'4'*'5',*'6',#,
                          -',-'',-'4',-'6',!,@)  algebra with

M., M*, M-, M!, M#, M@  carriers generated by A., A*, A-, A!, A#, A@
                        with ".", "*", "-", "!", "#", "@"

.     : M* x M. -> M.   binary operation [consing DRLHs to hyperarcs]

*'    : M. x M* -> M*   binary operation [consing hyperarcs to DRLHs]

*''   : M* x M* -> M*   binary operation [consing DRLHs to DRLHs]

*'''  : M- x M* -> M*   binary operation [consing negatives to DRLHs]

*'4'  : M! x M* -> M*   binary operation [consing embeds to DRLHs]

*'5'  : M# x M* -> M*   binary operation [consing mergings to DRLHs]

*'6'  : M@ x M* -> M*   binary operation [consing contacts to DRLHs]

#     : M* x M* -> M#   binary operation [merging DRLHs]

-'    : M. -> M-        unary operation [for negative hyperarcs]

-''   : M* -> M-        unary operation [for negative DRLHs]

-'4'  : M! -> M-        unary operation [for negative embeds]

-'6'  : M@ -> M-        unary operation [for negative contacts]

!     : M. -> M!        unary operation [for embedding hyperarcs]

@     : M* -> M@        unary operation [for contact labelnodes]
```

Axiomatic operator restriction:

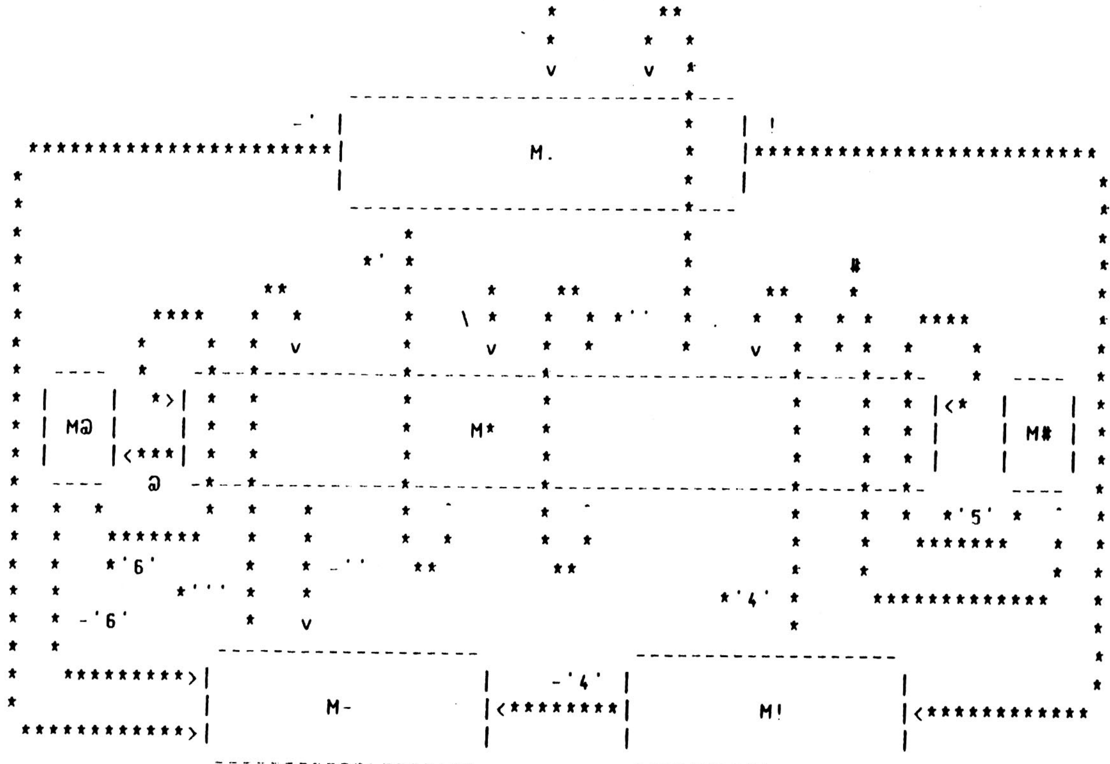
```
*listrestriction~: m1*m2 = \  if m2 IN A* MINUS {\}

mergerestriction~: m1#m2 = \  if  (m1 IN A* MINUS {\})
                               or (m2 IN A* MINUS {\})
```

The omission of an "-''''"-operator in ALCO5~ is for the reasons already explained for ALCO4~. The omission of an operator -'5': M# -> M excludes the unnecesary negation of mergings.

Again, the homogeneous .listrestriction axiom is superfluous here; the *listrestriction~ axiom use abbreviates six axioms for the "*"-primings. Also, no flatrestriction axiom is needed here. The hypactrestriction axiom has become superfluous because hyperarcs are now built from the DRLHs carrier, disjoint from the contacts carrier; the mergerestriction~ axiom is reformulated using the IN predicate, as usual; the contactrestriction axiom is now implicit in the use of M* as the domain of "∂".

[For the many-sorted generator-separated version ALCO5~$ of ALCO5~ the carrier M* would be divided into M/ and M_, so that not only would *lr~ -- as usual -- but also mr~ become superfluous, because its restriction effect would now be implicit in the domain of #: M_ x M_ -> M#.]

In our diagram notation ALCO5~ can be depicted thus:



The example for the homogeneous algebra ALCO5 can be transferred to the heterogeneous algebra ALCO5~ by using the generating sets A. = {`}, A* = {\, a, b}, A- = {}, A! = {}, A# = {}, and A∂ = {}, so that the basic binary form ∂b*'6'((∂a*'6'(b*''\))*''(b*''((∂b*'6'(a*''\))*''\))) together with the variant ∂b*'6'(∂a*'6'b*''\)*''b*''(∂b*'6'a*''\)*''\, its N-ary/paren-sparing short form, can be derived in M*.

A labelnode which has already occurred in isolation should not be duplicated through its designation as a contact labelnode, of course; instead, an isolated labelnode with a contact prefix "@" should merge not only with other "@"-occurrences of itself, as ensured by *Idempotence, but also with isolated labelnodes which are its "@"-less versions. For example, the complex labelnode {@f3,(f1,f2),(f2,f2)} should not only be equal to {@f3,@f3,(f1,f2),@f3,(f2,f2)} but also to {f3,@f3,(f1,f2),f3,(f2,f2)}. To take another example, {@b,{@a,b},{@b,a}} should not only be equal to {@b,{@a,b},@b,{@b,a}} but also to {@b,{@a,b},b,{@b,a}}, whose algebraic form was derived above. This can be accomplished by a property which is viewed here as a unary/N-ary aDsorption, but could also be regarded as a conditional Remove1 use with respect to "@" instead of to "-". [Although aD can only remove unprefixed elements <u>to its right</u>, *Commutativity -- postulated for all kinds of graphs -- can be applied as usual to arrange the required order before aD is applied.]

```
aDsorption: @m1*(m1*m2) = @m1*m2  if m1 IN A MINUS {`} or m1 = n*n'
shorter:    @m1*m1*m2   = @m1*m2  if m1 IN A MINUS {`} or m1 = n*n'
```

The condition on the aD equation prevents "erroneously" "@"-marked hyperarcs from adsorbing their unmarked versions, for example when using the empty hyperarc "`" with a "@"-mark:

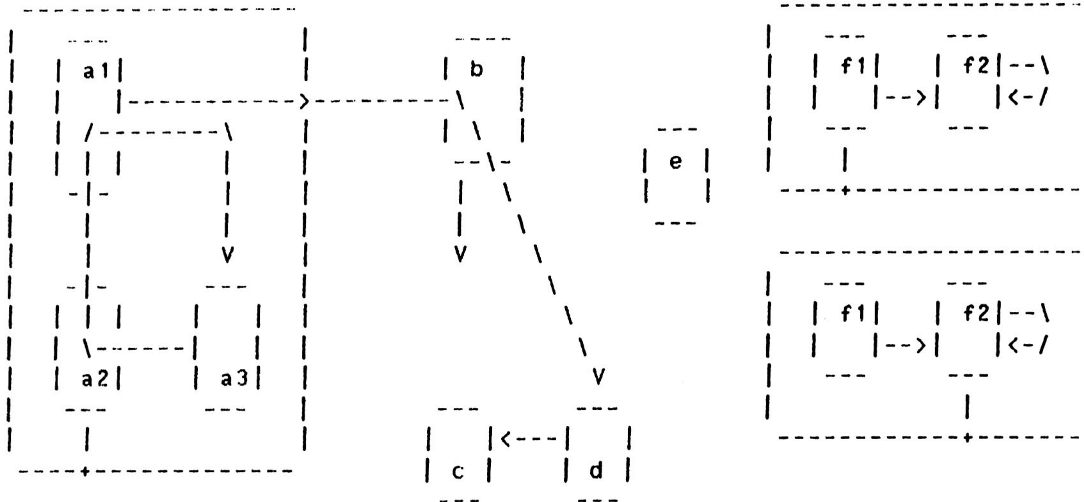
```
\*`*\ =cr=
@`*`*\ <aD>
@`*\ =cr=
\*\
```

In the heterogeneous formulation aDsorption changes thus:

```
aDsorption~: @m1*'6'(m1*''m2) = @m1*'6'm2
shorter:     @m1*'6'm1*''m2   = @m1*'6'm2
```

The condition on the aD equation has become implicit, because "@" is now only applicable to terms of the DRLHs carrier M*.

A less obvious property of DRLHs with contact labelnodes is the following: Two isolated complex labelnodes which only differ in their contact labelnodes should be merged to one complex labelnode using the union of these contact labelnodes as its contact labelnodes. For example, the two isolated complex labelnodes {@f1,(f1,f2),(f2,f2)} and {@f2,(f1,f2),(f2,f2)} should be merged to a single isolated complex labelnode {@f1,@f2,(f1,f2),(f2,f2)}. Thus the diagram



should be regarded as nothing but a distorted version of the earlier diagram in which both occurrences of the complex labelnode {..., (f1,f2),(f2,f2)} were merged. Likewise, continuing an earlier example, {∂a,b} and {∂b,a} in {∂b,{∂a,b},{∂b,a}} should be merged to {∂a,∂b}, obtaining {∂b,{∂a,∂b}}. For this a new property called "Similpotence" [a kind of "weakened" Idempotence not merging identical but "similar" expressions] together with a supplementary property Overlay is introduced.

```
Similpotence: (m1*m2)*((m3*m4)*m5) = ((m1*m2)#(m3*m4))*m5
shorter:      (m1*m2)*(m3*m4)*m5   = ((m1*m2)#(m3*m4))*m5

Overlay1: (∂m1*m2)#(m1*m3) = ∂m1*(m2#m3)
                                  if m1 IN A MINUS {`} or m1 = n*n'
Overlay2: (m1*m2)#(∂m1*m3) = ∂m1*(m2#m3)
                                  if m1 IN A MINUS {`} or m1 = n*n'
Overlay3: (m1*m2)#(m1*m3)  = m1*(m2#m3)
Overlay4: \#\              = \
```

The conditions on the Ov1 and Ov2 equations are present for analogous reasons as that on the aD equation above.

In the heterogeneous formulation these axioms can be given through the following scheme with p, q IN {1, 2, 6} and j IN {1, 2} [since p and q can independently represent three different primings, the S~ scheme represents nine axioms; since all three j occurrences must assume one of two values, the Ov3~ scheme represents two axioms]:

```
Similpotence~: (m1*'p'm2)*''((m3*'q'm4)*''m5) =
               ((m1*'p'm2)#(m3*'q'm4))*'5'm5
shorter:       (m1*'p'm2)*''(m3*'q'm4)*''m5   =
               ((m1*'p'm2)#(m3*'q'm4))*'5'm5

Overlay1~: (∂m1*'6'm2)#(m1*''m3) = ∂m1*'6'(m2#m3)
Overlay2~: (m1*''m2)#(∂m1*'6'm3) = ∂m1*'6'(m2#m3)
Overlay3~: (m1*'j'm2)#(m1*'j'm3) = m1*'j'(m2#m3)
Overlay4~: \#\                   = \
```

The conditions on the Ov1~ and Ov2~ equations are implicit for

analogous reasons as that on the aD~ equation above; the Ov4~ equation is the same as Ov4.

Of course, an attempt to use Similpotence for "non-similar" complex labelnodes will fail to produce a [completely] merged new complex labelnode:

```
(a*b*\)*(a*c*\)*\ =S=
((a*b*\)#(a*c*\))*\ =Ov3=
(a*((b*\)#(c*\)))*\
```

[Note that the size of the original term, 6+7, is larger than the size of the partially merged term, 5+6, which would cause the latter to be the normal form, according to our previous definition, if we permitted reducible terms with auxiliaries like "#" as normal forms.]

Furthermore, an inappropriate application of Ov3 instead of Ov1 or Ov2 may lead into a [harmless] "blind alley", which can only be left by "backtracking" through inverse equation applications:

```
(a*\)#(∂a*\) =aD=
 (a*\)#(∂a*a*\) =*C=
  (a*\)#(a*∂a*\) =Ov3=
   a*(\#(∂a*\)) =Ov3=
  (a*\)#(a*∂a*\) =*C=
 (a*\)#(∂a*a*\) =aD=
(a*\)#(∂a*\) =Ov2=
∂a*(\#\) =Ov4=
∂a*\
```

<u>Directed recursive labelnode hypergraphs with contact labelnodes</u>

ALCO5[.lr,.fr,hr,*lr,mr,cr,*C,*I,AD,aD,S]

<u>Directed recursive labelnode hypergraphs with contact labelnodes~</u>

```
ALCO5~[*lr~,mr~,*'C~,*''C~,*'6'C~,*'*''C~,*'*'6'C~,*''*'6'C~,
       *'I~,*''I~,*'4'I~,*'6'I~,AD~,aD~,S~]
```

Apart from the comments on the corresponding ALCO4~ quotient, it should be noted that in the present quotient mergings are neither allowed to commute [axioms *'5'C~, ... are not postulated] nor to remove each other idempotently [an axiom *'5'I~ is not postulated].

The entire normalization of the DRLH derived in an earlier example can now be presented. For a normal form it is now also required that no possible application of the new operator restriction axioms [hr, mr, or cr], aDsorption, or Similpotence [incl. Overlay] decrease its size.

```
Lem29: ∂b*((∂a*(b*\))*(b*((∂b*(a*\))*\))) = ∂b*((∂a*(∂b*\))*\)
Proof: ∂b*((∂a*(b*\))*(b*((∂b*(a*\))*\))) =*C=
       ∂b*(b*((∂a*(b*\))*((∂b*(a*\))*\))) =aD=
       ∂b*((∂a*(b*\))*((∂b*(a*\))*\)) =*C=
       ∂b*((∂a*(b*\))*((a*(∂b*\))*\)) =S=
       ∂b*(((∂a*(b*\))#(a*(∂b*\)))*\) =Ov1=
       ∂b*((∂a*((b*\)#(∂b*\)))*\) =Ov2=
       ∂b*((∂a*(∂b*(\#\))))*\) =Ov4=
       ∂b*((∂a*(∂b*\))*\)
```

It is not necessary that Similpotence [Overlay] find an *isolated* counterpart in one complex labelnode to every contact labelnode encountered in the other, as it did in the above proof for the complex labelnodes {∂a,b} and {∂b,a} after the latter was commuted to {a,∂b}: By virtue of ADsorption, *quasi-isolated* counterparts can be generated from hyperarcs for use by Similpotence [Overlay]. For this both complex labelnodes may have to be transformed, as in {∂f1,(f1,f2),(f2,f2)} and {∂f2,(f1,f2),(f2,f2)}, transformed to {∂f1,f2,(f1,f2),(f2,f2)} and {f1,∂f2,(f1,f2),(f2,f2)}, respectively; in other cases the change of only one complex labelnode may be necessary, as in {∂b,(b,c)} and {(b,c)}, where the second is transformed to {b,(b,c)}. This latter example will now be used to show how such Similpotence-preparing transformations are performed by means of ADsorption applications [the y variable permits the complex labelnodes to occur in any DR*L*H right-context].

```
Lem30: (∂b*(b.c.`)*\)*((b.c.`)*\)*y = (∂b*(b.c.`)*\)*y
Proof: (∂b*(b.c.`)*\)*((b.c.`)*\)*y =AD=
       (∂b*(b.c.`)*\)*((b.c.`)*!(b.c.`)*\)*y =Em1=
       (∂b*(b.c.`)*\)*((b.c.`)*b*!(c.`)*\)*y =*C=
       (∂b*(b.c.`)*\)*(b*(b.c.`)*!(c.`)*\)*y =AD=
       (∂b*(b.c.`)*\)*(b*(b.c.`)*!(b.c.`)*!(c.`)*\)*y =Em1=
       (∂b*(b.c.`)*\)*(b*(b.c.`)*b*!(c.`)*!(c.`)*\)*y =*I=
       (∂b*(b.c.`)*\)*(b*(b.c.`)*b*!(c.`)*-!(c.`)*!(c.`)*\)*y =*R1=
       (∂b*(b.c.`)*\)*(b*(b.c.`)*b*!(c.`)*-!(c.`)*\)*y =*R3=
       (∂b*(b.c.`)*\)*(b*(b.c.`)*b*!(c.`)*\)*y =Em1=
       (∂b*(b.c.`)*\)*(b*(b.c.`)*!(b.c.`)*\)*y =AD=
       (∂b*(b.c.`)*\)*(b*(b.c.`)*\)*y =S=
       ((∂b*(b.c.`)*\)#(b*(b.c.`)*\))*y =Ov1=
       (∂b*(((b.c.`)*\)#((b.c.`)*\)))*y =Ov3=
       (∂b*(b.c.`)*(\#\))*y =Ov4=
       (∂b*(b.c.`)*\)*y
```

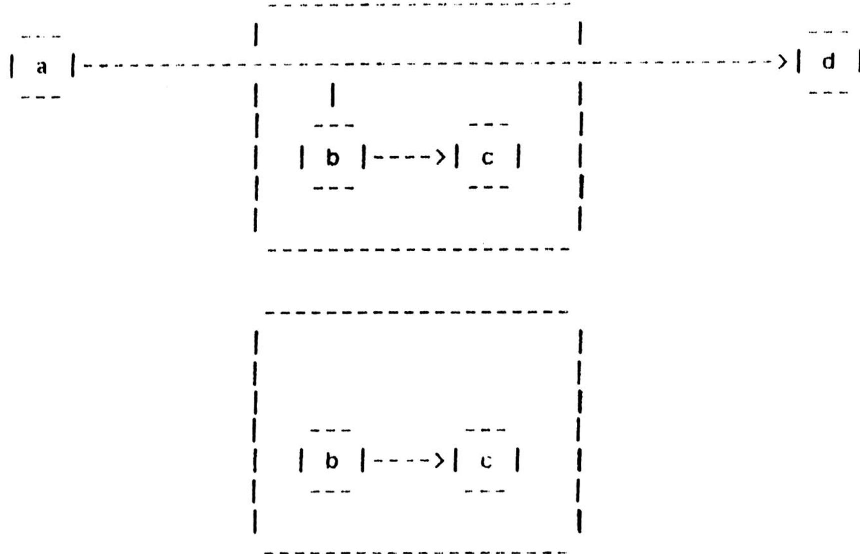
Here AD is not only applied twice for the same hyperarc, as usual, but actually three times, to generate an extra copy of the quasi-isolated labelnode for use in readsoption [alternatively, Idempotence could have been used more directly for that purpose, through Lem24]:

1. The quasi-isolated labelnode [b] is generated from the hyperarc [(b.c.`)] together with superfluous hyperarc parts [here only !(c.`)].

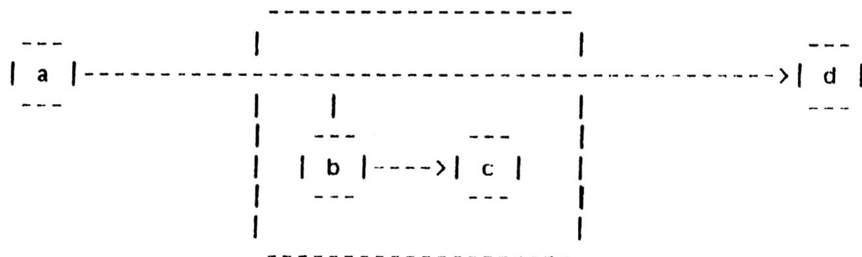
2. After the desired quasi-isolated labelnode commuted away to its destination, the generation is repeated to obtain a complete "!"-copy of the hyperarc for use in 3 [the superfluous parts thus become duplicated, but the duplicates are removed idempotently].

3. The remaining single "!"-copy is re-adsorbed by the original hyperarc.

Similpotence can be used together with ADsorption to enable a final kind of merging [this is required, for instance, to prove the sample DRLH equality in the introduction]: An isolated complex labelnode can be merged with a non-isolated complex labelnode [i.e. a labelnode used in a hyperarc] if it differs from it only in possessing merely a subset of the contact labelnodes of those possessed by the non-isolated one. For example, the DRLH



should be equivalent to the DRLH below, because the isolated complex labelnode not possessing any contact labelnode can be merged with the non-isolated complex labelnode possessing one contact labelnode.



That this is in fact the case, without requiring another axiom, can be shown as follows:

```
Lem31: (a.(∂b*(b.c.`)*\).d.`)*((b.c.`)*\)*\ = (a.(∂b*(b.c.`)*\).d.`)*\
Proof:
(a.(∂b*(b.c.`)*\).d.`)*((b.c.`)*\)*\ =AD=
(a.(∂b*(b.c.`)*\).d.`)*!(a.(∂b*(b.c.`)*\).d.`)*((b.c.`)*\)*\ =Em1=
(a.(∂b*(b.c.`)*\).d.`)*a*!((∂b*(b.c.`)*\).d.`)*((b.c.`)*\)*\ =Em1=
(a.(∂b*(b.c.`)*\).d.`)*a*(∂b*(b.c.`)*\)*!(d.`)*((b.c.`)*\)*\ =*C=
(a.(∂b*(b.c.`)*\).d.`)*a*(∂b*(b.c.`)*\)*((b.c.`)*\)*!(d.`)*\ =Lem30=
(a.(∂b*(b.c.`)*\).d.`)*a*(∂b*(b.c.`)*\)*!(d.`)*\ =Em1=
(a.(∂b*(b.c.`)*\).d.`)*a*!((∂b*(b.c.`)*\).d.`)*\ =Em1=
(a.(∂b*(b.c.`)*\).d.`)*!(a.(∂b*(b.c.`)*\).d.`)*\ =AD=
(a.(∂b*(b.c.`)*\).d.`)*\
```

Note that ADsorption is used here to generate a quasi-isolated copy of

the complex labelnode [(@b*(b.c.`)*\)] used in the hyperarc [(a.(@b*(b.c.`)*\).d.`)]. This copy is then merged with the isolated complex labelnode [((b.c.`)*\)] by means of Similpotence, in essence, as shown in Lem30. Since the quasi-isolated labelnode had a superset of the contact labelnodes possessed by the isolated one, the merge result is not different from the quasi-isolated labelnode, so that it can be re-adsorbed by the hyperarc.

Concluding our discussion of Similpotence, we note that for arbitrary complex labelnodes Similpotence is a proper generalization of Idempotence. More precisely, we make two claims:

1. There are complex labelnodes which can be merged by means of Similpotence but not by means of Idempotence [because they are not identical]. Clearly, examples can be found in all previous Similpotence lemmas, e.g. Lem30.

2. If two complex labelnodes can be merged by means of Idempotence [because they are identical], they can also be merged by means of Similpotence. To show this, we assume without loss of generality that both complex labelnodes have the form x = (x1*x2*...*xN*\) and occur adjacently in an arbitrary right-context y [this can always be arranged using the D<u>RL</u>H properties, in particular *Commutativity]. Using Idempotence they can be merged as follows:

```
(x1*x2*...*xN*\)*(x1*x2*...*xN*\)*y =I=
(x1*x2*...*xN*\)*-(x1*x2*...*xN*\)*(x1*x2*...*xN*\)*y =R1=
(x1*x2*...*xN*\)*-(x1*x2*...*xN*\)*y =I=
(x1*x2*...*xN*\)*y
```

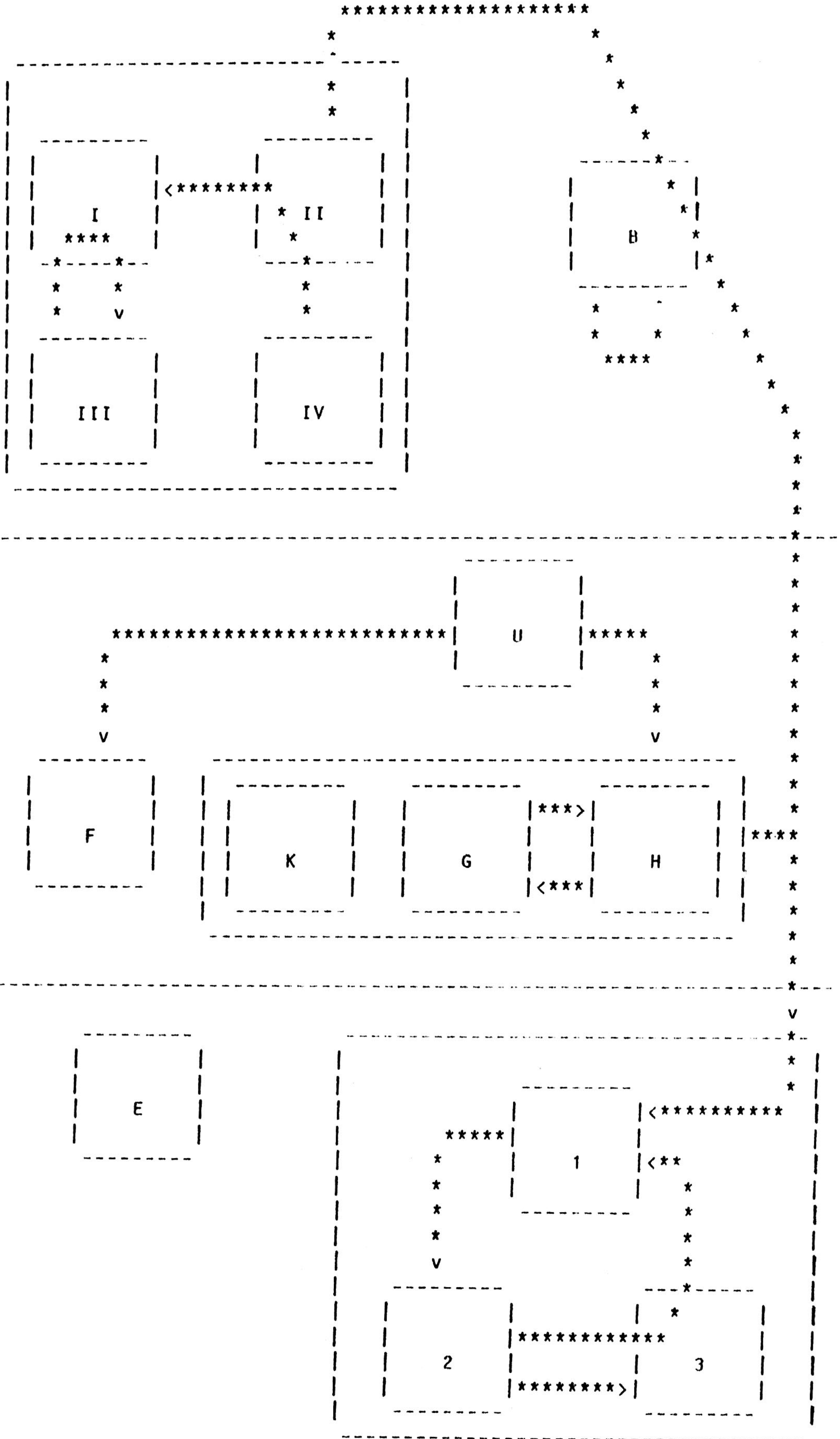
Using Similpotence, then, the complex labelnodes can be merged thus:

```
(x1*x2*...*xN*\)*(x1*x2*...*xN*\)*y =S=
((x1*x2*...*xN*\)#(x1*x2*...*xN*\))*y =Ov3=
(x1*((x2*...*xN*\)#(x2*...*xN*\)))*y =Ov3=
                  . . .
(x1*x2*...*xN*(\#\))*y =Ov4=
(x1*x2*...*xN*\)*y
```

The ". . ." elides an induction over the length l of the complex labelnode x, which can be made explicit by splitting x into (x'*x''), i.e. by reverting to the non-abbreviated basic binary form:

```
x*x*y =S= (x#x)*y =IndOv= x*y

IndOv: x#x = x

Proof:
x = \;         l(x) = 0 :                        \#\ =Ov4= \
x = x'*x'';  l(x'') = n-1 -> l(x'*x'') = n :  (x'*x'')#(x'*x'') =Ov3=
                                               x'*(x''#x'') =induct.hyp.=
                                               x'*x''
```

Finally, let us consider a larger example of equality proof in the algebra of D<u>RL</u>Hs with contact labelnodes. Starting from a redundant form, it shows how to obtain the normal form for the D<u>RL</u>H given as the following diagram, taken from (Boley 1980).


I
II
III
IV
B
U
F
K
G
H
E
1
2
3

The proof is the algebraic version of a FIT DRLH constructor application trace in (Boley 1980), whose initial expression could be rewritten as the following DRLH/TUPLE collection nesting for conciseness using "@"-prefixes instead of the previous CONTACT expressions [or the 1980 ":"-infixes] to designate contact labelnodes.

```
(DRLH
 (TUPLE
  (DRLH @II @II (TUPLE IV II I) (TUPLE III I III))
  B
  (DRLH
   @(DRLH K (TUPLE G H) K H (TUPLE H G))
   (TUPLE U F)
   (TUPLE U (DRLH K (TUPLE G H) (TUPLE H G) H G H)))
  (DRLH @1 @1 @1 (TUPLE 1 2) 1 (TUPLE 2 3 1) 2 (TUPLE 2 3) 3))
 (TUPLE B B)
 (DRLH @II (TUPLE IV II I) I (TUPLE III I III))
 E)
```

The proof will not exploit maximum parallelism but will perform only semantically related local transformations in parallel, thus making it easy to follow in spite of its greater length. The transformations will be applied in an "inside-out" manner, corresponding to the call-by-value DRLH constructor applications. As in these, we will also use the following order of normalization [FIT normalization functions are given in brackets]: 1. *Idempotence [REMEQUALS], 2. ADsorption [CLEAN], 3. *Commutativity [SORDRLH].

Lem32:

```
 ((@II*@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*K*H*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*H*G*H*\).`)
   *\)
  .(@1*@1*@1*(1.2.`)*1*(2.3.1.`)*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\

=

 E
*(B.B.`)
*((@II*(III.I.III.`)*(IV.II.I.`)*\)
  .B
  .(@(K*(G.H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
  .`)
*\
```

Proof:

```
 ((əII*əII*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(ə(K*(G.H.`)*K*H*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*H*G*H*\).`)
   *\)
  .(ə1*ə1*ə1*(1.2.`)*1*(2.3.1.`)*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(əII*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=*I|*I|*I|*I=
 ((əII*-əII*əII*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(ə(K*-K*(G.H.`)*K*H*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*H*-H*G*H*\).`)
   *\)
  .(ə1*-ə1*ə1*ə1*(1.2.`)*1*(2.3.1.`)*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(əII*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=*R1|*R2|*R2|*R1=
 ((əII*-əII*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(ə(K*(G.H.`)*-K*K*H*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*H*G*-H*H*\).`)
   *\)
  .(ə1*-ə1*ə1*(1.2.`)*1*(2.3.1.`)*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(əII*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=*R2|*R1|*R1|*R1=
 ((əII*(IV.II.I.`)*-əII*(III.I.III.`)*\)
  .B
  .(ə(K*(G.H.`)*-K*H*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*H*G*-H*\).`)
   *\)
  .(ə1*-ə1*(1.2.`)*1*(2.3.1.`)*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(əII*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=*R2|*R2|*R3|*R2=
 ((əII*(IV.II.I.`)*(III.I.III.`)*-əII*\)
  .B
  .(ə(K*(G.H.`)*H*-K*(H.G.`)*\)
   *(U.F.`)
```

```
    *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
    *\)
  .(∂1*(1.2.`)*-∂1*1*(2.3.1.`)*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(∂II*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=*R3|*R2|*R2=
 ((∂II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(∂(K*(G.H.`)*H*(H.G.`)*-K*\)
    *(U.F.`)
    *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
    *\)
  .(∂1*(1.2.`)*1*-∂1*(2.3.1.`)*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(∂II*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=*R3|*R2=
 ((∂II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(∂(K*(G.H.`)*H*(H.G.`)*\)
    *(U.F.`)
    *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
    *\)
  .(∂1*(1.2.`)*1*(2.3.1.`)*-∂1*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(∂II*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=*R2=
 ((∂II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(∂(K*(G.H.`)*H*(H.G.`)*\)
    *(U.F.`)
    *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
    *\)
  .(∂1*(1.2.`)*1*(2.3.1.`)*2*-∂1*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(∂II*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=*R2=
 ((∂II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(∂(K*(G.H.`)*H*(H.G.`)*\)
    *(U.F.`)
    *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
    *\)
  .(∂1*(1.2.`)*1*(2.3.1.`)*2*(2.3.`)*-∂1*3*\)
  .`)
*(B.B.`)
*(∂II*(IV.II.I.`)*I*(III.I.III.`)*\)
```

```
*E
*\
=*R2=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*H*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
   *\)
  .(@1*(1.2.`)*1*(2.3.1.`)*2*(2.3.`)*3*-@1*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=*R3=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*H*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
   *\)
  .(@1*(1.2.`)*1*(2.3.1.`)*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=*C=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*H*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.1.`)*1*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=AD|AD|AD|AD=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*!(G.H.`)*H*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*!(H.G.`)*H*G*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.1.`)*!(2.3.1.`)*1*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*!(IV.II.I.`)*I*(III.I.III.`)*\)
*E
*\
=Em1|Em1|Em1|Em1=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*G*!(H.`)*H*(H.G.`)*\)
```

```
    *(U.F.`)
    *(U.(K*(G.H.`)*(H.G.`)*H*!(G.`)*H*G*\).`)
    *\)
  .(@1*(1.2.`)*(2.3.1.`)*2*!(3.1.`)*1*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*!(II.I.`)*I*(III.I.III.`)*\)
*E
*\
=Em2|Em2|Em1|Em1=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*G*H*H*(H.G.`)*\)
    *(U.F.`)
    *(U.(K*(G.H.`)*(H.G.`)*H*G*H*G*\).`)
    *\)
  .(@1*(1.2.`)*(2.3.1.`)*2*3*!(1.`)*1*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*II*!(I.`)*I*(III.I.III.`)*\)
*E
*\
=Em2|Em2=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*G*H*H*(H.G.`)*\)
    *(U.F.`)
    *(U.(K*(G.H.`)*(H.G.`)*H*G*H*G*\).`)
    *\)
  .(@1*(1.2.`)*(2.3.1.`)*2*3*1*1*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*II*I*I*(III.I.III.`)*\)
*E
*\
=*I|*I|*I|*I|*I|*I|*I=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*G*H*-H*H*(H.G.`)*\)
    *(U.F.`)
    *(U.(K*(G.H.`)*(H.G.`)*H*-H*G*-G*H*G*\).`)
    *\)
  .(@1*(1.2.`)*(2.3.1.`)*2*-2*3*-3*1*-1*1*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*II*I*-I*I*(III.I.III.`)*\)
*E
*\
=*R1|*R2|*R2|*R2|*R2|*R1|*R1=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*G*H*-H*(H.G.`)*\)
    *(U.F.`)
    *(U.(K*(G.H.`)*(H.G.`)*H*G*-H*H*-G*G*\).`)
    *\)
  .(@1*(1.2.`)*(2.3.1.`)*2*3*-2*1*-3*-1*2*(2.3.`)*3*\)
  .`)
*(B.B.`)
```

```
*(@II*(IV.II.I.`)*IV*II*I*-I*(III.I.III.`)*\)
*E
*\
=*R2|*R1|*R1|*R2|*R2|*R2=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
 .B
 .(@(K*(G.H.`)*G*H*(H.G.`)*-H*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*H*G*-H*-G*\).`)
  *\)
 .(@1*(1.2.`)*(2.3.1.`)*2*3*1*-2*-3*2*-1*(2.3.`)*3*\)
 .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*II*I*(III.I.III.`)*-I*\)
*E
*\
=*R3|*R3|*R2|*R2|*R3=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
 .B
 .(@(K*(G.H.`)*G*H*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*H*G*-H*\).`)
  *\)
 .(@1*(1.2.`)*(2.3.1.`)*2*3*1*-2*2*-3*(2.3.`)*-1*3*\)
 .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*II*I*(III.I.III.`)*\)
*E
*\
=*R3|*R1|*R2|*R2=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
 .B
 .(@(K*(G.H.`)*G*H*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
  *\)
 .(@1*(1.2.`)*(2.3.1.`)*2*3*1*-2*(2.3.`)*-3*3*-1*\)
 .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*II*I*(III.I.III.`)*\)
*E
*\
=*R2|*R1|*R3=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
 .B
 .(@(K*(G.H.`)*G*H*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
  *\)
 .(@1*(1.2.`)*(2.3.1.`)*2*3*1*(2.3.`)*-2*-3*\)
 .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*II*I*(III.I.III.`)*\)
*E
*\
=*R3=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
 .B
```

```
  .(@(K*(G.H.`)*G*H*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.1.`)*2*3*1*(2.3.`)*-2*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*II*I*(III.I.III.`)*\)
*E
*\
=*R3=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*G*H*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*H*G*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.1.`)*2*3*1*(2.3.`)*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*II*I*(III.I.III.`)*\)
*E
*\
=Em2|Em2|Em2|Em2=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*G*!(H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*H*!(G.`)*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.1.`)*2*3*!(1.`)*(2.3.`)*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*II*!(I.`)*(III.I.III.`)*\)
*E
*\
=Em1|Em1|Em1|Em1=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*!(G.H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*!(H.G.`)*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.1.`)*2*!(3.1.`)*(2.3.`)*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*IV*!(II.I.`)*(III.I.III.`)*\)
*E
*\
=AD|AD|Em1|Em1=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.1.`)*!(2.3.1.`)*(2.3.`)*\)
  .`)
```

```
*(B.B.`)
*(@II*(IV.II.I.`)*!(IV.II.I.`)*(III.I.III.`)*\)
*E
*\
=AD|AD=
 ((@II*(IV.II.I.`)*(III.I.III.`)*\)
  .B
  .(@(K*(G.H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.1.`)*(2.3.`)*\)
  .`)
*(B.B.`)
*(@II*(IV.II.I.`)*(III.I.III.`)*\)
*E
*\
=*C|*C|*C=
 ((@II*(III.I.III.`)*(IV.II.I.`)*\)
  .B
  .(@(K*(G.H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
  .`)
*(B.B.`)
*(@II*(III.I.III.`)*(IV.II.I.`)*\)
*E
*\
=AD=
 ((@II*(III.I.III.`)*(IV.II.I.`)*\)
  .B
  .(@(K*(G.H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
  .`)
*!((@II*(III.I.III.`)*(IV.II.I.`)*\)
   .B
   .(@(K*(G.H.`)*(H.G.`)*\)
    *(U.F.`)
    *(U.(K*(G.H.`)*(H.G.`)*\).`)
    *\)
   .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
   .`)
*(B.B.`)
*(@II*(III.I.III.`)*(IV.II.I.`)*\)
*E
*\
=Em1=
 ((@II*(III.I.III.`)*(IV.II.I.`)*\)
  .B
  .(@(K*(G.H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*\).`)
   *\)
```

```
 .(ə1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
 .`)
*(əII*(III.I.III.`)*(IV.II.I.`)*\)
*!(B
  .(ə(K*(G.H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*\).`)
   *\)
  .(ə1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
  .`)
*(B.B.`)
*(əII*(III.I.III.`)*(IV.II.I.`)*\)
*E
*\
=*I=
 ((əII*(III.I.III.`)*(IV.II.I.`)*\)
 .B
 .(ə(K*(G.H.`)*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*\).`)
  *\)
 .(ə1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
 .`)
*(əII*(III.I.III.`)*(IV.II.I.`)*\)
*-(əII*(III.I.III.`)*(IV.II.I.`)*\)
*!(B
  .(ə(K*(G.H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*\).`)
   *\)
  .(ə1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
  .`)
*(B.B.`)
*(əII*(III.I.III.`)*(IV.II.I.`)*\)
*E
*\
=*R2=
 ((əII*(III.I.III.`)*(IV.II.I.`)*\)
 .B
 .(ə(K*(G.H.`)*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*\).`)
  *\)
 .(ə1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
 .`)
*(əII*(III.I.III.`)*(IV.II.I.`)*\)
*!(B
  .(ə(K*(G.H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*\).`)
   *\)
  .(ə1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
  .`)
*-(əII*(III.I.III.`)*(IV.II.I.`)*\)
*(B.B.`)
*(əII*(III.I.III.`)*(IV.II.I.`)*\)
*E
*\
```

```
=Em1|*R2=
 ((@II*(III.I.III.`)*(IV.II.I.`)*\)
 .B
 .(@(K*(G.H.`)*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*\).`)
  *\)
 .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
 .`)
*!((@II*(III.I.III.`)*(IV.II.I.`)*\)
  .B
  .(@(K*(G.H.`)*(H.G.`)*\)
   *(U.F.`)
   *(U.(K*(G.H.`)*(H.G.`)*\).`)
   *\)
  .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
  .`)
*(B.B.`)
*-(@II*(III.I.III.`)*(IV.II.I.`)*\)
*(@II*(III.I.III.`)*(IV.II.I.`)*\)
*E
*\
=AD|*R1=
 ((@II*(III.I.III.`)*(IV.II.I.`)*\)
 .B
 .(@(K*(G.H.`)*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*\).`)
  *\)
 .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
 .`)
*(B.B.`)
*-(@II*(III.I.III.`)*(IV.II.I.`)*\)
*E
*\
=*R2=
 ((@II*(III.I.III.`)*(IV.II.I.`)*\)
 .B
 .(@(K*(G.H.`)*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*\).`)
  *\)
 .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
 .`)
*(B.B.`)
*E
*-(@II*(III.I.III.`)*(IV.II.I.`)*\)
*\
=*R3=
 ((@II*(III.I.III.`)*(IV.II.I.`)*\)
 .B
 .(@(K*(G.H.`)*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*\).`)
  *\)
 .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
 .`)
*(B.B.`)
```

```
*E
*\
=*C=
 (B.B.`)
*((@II*(III.I.III.`)*(IV.II.I.`)*\)
 .B
 .(@(K*(G.H.`)*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*\).`)
  *\)
 .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
 .`)
*E
*\
=*C=
 (B.B.`)
*E
*((@II*(III.I.III.`)*(IV.II.I.`)*\)
 .B
 .(@(K*(G.H.`)*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*\).`)
  *\)
 .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
 .`)
*\
=*C=
 E
*(B.B.`)
*((@II*(III.I.III.`)*(IV.II.I.`)*\)
 .B
 .(@(K*(G.H.`)*(H.G.`)*\)
  *(U.F.`)
  *(U.(K*(G.H.`)*(H.G.`)*\).`)
  *\)
 .(@1*(1.2.`)*(2.3.`)*(2.3.1.`)*\)
 .`)
*\
```

6 REFERENCES

Bartels, U. & Olthoff, W. & Raulefs, P.: An expert system for implementing abstract sorting algorithms on parameterized abstract data types. Proc. GWAI-81, Bad Honnef, Informatik-Fachberichte 47, Springer-Verlag 1981, 112-123.

Beierle, C. & Voss, A.: Canonical term functors and parameterization-by-use for the specification of abstract data types. Univ. Kaiserslautern, FB Informatik, Interner Bericht 77/83, MEMO SEKI-83-07, May 1983.

Bellia, M. & Dameri, E. & Degano, P. & Levi, G. & Martelli, M.: A formal model for lazy implementations of a Prolog-compatible functional language. In: Campbell, J. [Ed.]: Implementations of PROLOG. Ellis Horwood, 1984, 309-326.

Berge, C.: Graphes et hypergraphes. Dunod, Paris 1970.

Birkhoff, G. & Lipson, J.: Heterogeneous algebras. Journal of Combinatorial Theory 8, 1970, 115-133.

Boehm, H.-P. & Fischer, H. & Raulefs, P.: CSSA: Language concepts and programming methodology. Proc. Symp. Artificial Intelligence and Programming Languages, SIGPLAN Notices 12(8), Special Issue, Aug. 1977, 100-108.

Boley, H.: Five views of FIT programming. Univ. Hamburg, Fachbereich Informatik, IFI-HH-B-57/79, Sept. 1979.

Boley, H.: Processing directed recursive labelnode hypergraphs with FIT programs. Univ. Hamburg, FB Informatik, IFI-HH-M-81/80, Sept. 1980.

Boley, H.: FIT - PROLOG: A functional/relational language comparison Univ. Kaiserslautern, FB Informatik, Interner Bericht 95/83, MEMO SEKI-83-14, Dec. 1983.

Broy, M. & Dosch, W. & Partsch, H. & Pepper, P. & Wirsing, M: Existential quantifiers in abstract data types. Proc. 6th ICALP, Graz, July 1979, Springer LNCS 71, 73-87.

Burstall, R. & Goguen, J.: Putting theories together to make specifications. Proc. 5th IJCAI-77, Cambridge, Mass., Aug. 1977, 1045-1058.

Burstall, R. & Landin, P.: Programs and their proofs: An algebraic approach. Machine Intelligence 4, 1969, 17-43.

Burstall, R. & MacQueen, D. & Sannella, D.: HOPE: An experimental applicative language. Conference Record of the 1980 LISP Conference, Stanford University, August 1980, 136-143.

Chikayama, T.: ESP - extended self-contained PROLOG - as a preliminary kernel language of fifth generation computers. New Generation Computing 1(1), 1983, 11-24.

Dilger, W. & Womann, W.: Semantic networks as abstract data types. Proc. 8th IJCAI-83, Karlsruhe, Aug. 1983, 321-324.

Dogen (Furukawa, K. & Nakajima, R. & Yonezawa, A.): Modularization and abstraction in logic programming. New Generation Computing 1(2), 1983, 169-177.

Ehrig, H. & Kreowski, H.-J. & Thatcher, J. & Wagner, F. & Wright, J.: Parameterized data types in algebraic specification languages. Proc. 7th ICALP, Noordwijkerhout, July 1980, Springer LNCS 85, 157-168.

Goguen, J. & Meseguer, J.: Equality, types, modules and generics for logic programming. Stanford University, Center for the Study of Language and Information, Report No. CSLI-84-5, March 1984.

Goguen, J. & Thatcher, J. & Wagner, E.: An initial algebra approach to the specification, correctness, and implementation of abstract data types. In: Yeh, R [Ed.]: Current trends in programming methodology. Vol. IV. Prentice-Hall, 1978, 81-149.

Graetzer, G.: Universal algebra. Van Nostrand, 1968.

Guiho, G.: Automatic programming using abstract data types. Proc. 8th IJCAI-83, Karlsruhe, Aug. 1983, 1-9.

Higgins, P.: Algebras with a scheme of operators. Mathematische Nachrichten 27, 1963, 115-132.

Klaeren, H.: A constructive method for abstract algebraic software specification. RWTH Aachen, Schriften zur Informatik & Ang. Math., Bericht Nr. 78, Juli 1982.

Loeckx, J.: Algorithmic specifications of abstrac data types. Proc. 8th ICALP, Acre, July 1981, Springer LNCS 115, 129-147.

McCarthy, J.& Talcott, C.: LISP - Programming and proving. Stanford University, book preprint, September 1980.

Morris, J.: Types are not sets. ACM Symp. Principles of Programming Languages, Boston, Mass., Oct. 1973, 120-124.

Nakashima, H. & Suzuki, N.: Data abstraction in Prolog/KR. New Generation Computing 1(1), 1983, 49-62.

Oppen, D.: Reasoning about recursively defined data structures. Stanford University, AI Laboratory, Memo AIM-314, Report No. STAN-CS-78-678, July 1978.

Raulefs, P. & Siekmann, J. & Szabo, P. & Unvericht, E.: A short survey on the state of the art in matching and unification problems. SIGSAM Bulletin 13(2), May 1979, 14-20.

Reimer, U. & Hahn, U.: A formal approach to the semantics of a frame data model. Proc. 8th IJCAI-83, Karlsruhe, Aug. 1983, 337-339.

Rulifson, J. & Derksen, J. & Waldinger, R.: QA4: A procedural calculus for intuitive reasoning. Stanford Research Institute, AI Center, Technical Note 73, Nov. 1972.

Thatcher, J. & Wagner, E. & Wright, J.: Data type specification: Parameterization and the power of specification techniques. IBM, Research Report, Computer Science, RC 7757 (#33565), 7/6/79.

Travis, L. & Honda, M. & LeBlanc, R. & Zeigler, S.: Design rationale for TELOS, a PASCAL-based AI language. Proc. Symp. Artificial Intelligence and Programming Languages, SIGPLAN Notices 12(8), Special Issue, Aug. 1977, 67-76.