Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research
Report
RR-90-12

Declarative Operations on Nets

Harold Boley

October 1990

Deutsches Forschungszentrum für Künstliche Intelligenz
GmbH

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, SEMA Group, and Siemens. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- ❑ Intelligent Engineering Systems
- ❑ Intelligent User Interfaces
- ❑ Computer Linguistics
- ❑ Programming Systems
- ❑ Deduction and Multiagent Systems
- ❑ Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Friedrich J. Wendl
Director

Declarative Operations on Nets

Harold Boley

DFKI-RR-90-12

To appear in: Fritz Lehmann (Ed.) "Semantic Networks in Artificial Intelligence",
Computers & Mathematics with Applications, Pergamon Press.

Contents

Declarative Operations on Nets

Harold Boley

Deutsches Forschungszentrum für Künstliche Intelligenz
Box 2080, D-6750 Kaiserslautern, F. R. Germany
boley@informatik.uni-kl.de

October 16, 1990

Abstract

To increase the expressiveness of knowledge representations, the graph-theoretical basis of semantic networks is reconsidered. Directed labeled graphs are generalized to directed recursive labelnode hypergraphs, which permit a most natural representation of multi-level structures and n-ary relationships. This net formalism is embedded into the relational/functional programming language RELFUN. Operations on (generalized) graphs are specified in a declarative fashion to enhance readability and maintainability. For this, nets are represented as nested RELFUN terms kept in a normal form by rules associated directly with their constructors. These rules rely on equational axioms postulated in the formal definition of the generalized graphs as a constructor algebra. Certain kinds of sharing in net diagrams are mirrored by binding common subterms to logical variables. A package of declarative transformations on net terms is developed. It includes generalized set operations, structure-reducing operations, and extended path searching. The generation of parts lists is given as an application in mechanical engineering. Finally, imperative net storage and retrieval operations are discussed.

1 Introduction

The representational paradigm of semantic networks has been explored most formally for taxonomic inheritance systems. These can be based on strict hierarchies (trees) or 'multiple-inheritance' heterarchies (directed acyclic graphs). Recently, the formal study of "cyclic definitions" in KL-ONE-like languages has again acknowledged the general-graph basis of classical semantic networks ([Neb89], [Baa90]). But even this more truthful **net** concept has representational deficiencies. Three of these are highlighted here to provide some background for the following discussion.

1) Semantic networks are used as graph-based formalisms for structuring knowledge. However, the classical directed labeled graphs (DLGs) are too simple—"flat & binary"—to capture the richness of human knowledge structures. Therefore, we will employ the generalized graph-theoretical notion of directed recursive labelnode hypergraphs (DR$\mathcal{L}$Hs), as developed in [Bol77], [Bol80], and [Bol84]. Our objective here is to obtain the greatest expressive power in a representation of knowledge by maximizing **generality**, thus minimizing representational artifacts imposed by the DLG 'syntax'. We will postpone the **recursive** and **labelnode** features to the following section. Regarding directed **hypergraphs**, Fig. 1 exemplifies the usual DLG way ternary or higher-arity relations are represented by regarding a relation r as a node linked to artificially created nodes r', r'', ... for its relationships (most semantic net systems, including KL-ONE, promote such pseudo-entities into the universe of concepts); from these, three or more artificial arcs, labeled by binary pseudo-relations $arg1$, $arg2$, $arg3$, ..., point to the arguments (such pseudo-relations are syntactic placeholders, often reinterpreted as KL-ONE-like semantic "roles"). On the other

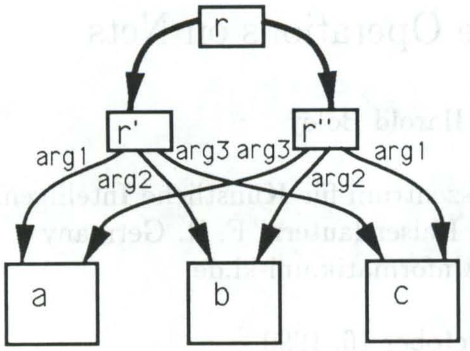

Figure 1: A graph simulating two ternary r relationships with artificial nodes r' and r''

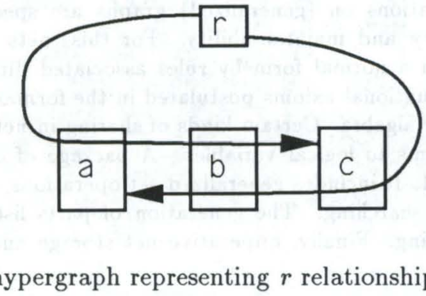

Figure 2: A hypergraph representing r relationships by arrows cutting intermediate nodes

hand, Fig. 2 shows how directed hypergraphs permit a natural representation of n-ary relations ($n \geq 3$) by directed hyperarcs or arrows starting with the relation node r, cutting the first $n-1$ argument nodes, and ending at the nth argument node (artificial nodes and arcs become superfluous because of the more powerful 'built-in' structure of DR$\mathcal{L}$Hs). Note that the DR$\mathcal{L}$H representation gracefully specializes to binary relations, while in DLG representations there is a discontinuity if normal r-labeled arcs are kept for the binary case. Just as DLGs have permitted natural binary links in ordinary semantic networks, directed hypergraphs permit natural n-ary links in our generalized nets; there is now a parallel development from ordinary KL-ONE systems to n-ary ones [Sch89].

2) The intuitive appeal of semantic nets is largely due to their pictorial, 2-dimensional (or even spatial) diagram forms. Yet 1-dimensional linear strings of symbols are often used instead of drawing large nets on paper, for representing nets as data structures, and for specifying operations on them. So we will carefully tailor such a "symbolic form" to our generalized graph notion, trying to keep the principal 2D advantage of "node sharing" by using terms with coreferential "logical variables". For example, the DR$\mathcal{L}$H in Fig. 2 will be put into the symbolic form $\lfloor(r,a,b,c),(r,c,b,a)\rfloor$, where the hyperarcs become list terms and the entire DR$\mathcal{L}$H becomes an enclosing set-like "$\lfloor\ \rfloor$"-term. The nodes a, b, and c can be shared by both hyperarcs by assigning them to variables A, B, and C via A `is` a, B `is` b, C `is` c, and then writing the symbolic DR$\mathcal{L}$H pattern $\lfloor(r,A,B,C),(r,C,B,A)\rfloor$[1]

3) Any knowledge representation formalism should, besides its 'static' expressiveness, provide a library of useful operations. Semantic nets have traditionally focused on inheritance and path-

[1] As in PROLOG, variable names will be distinguished from constants by a capital first letter (the anonymous variable being "_"); "single-assignment" variable bindings will be specified by an `is` infix.

tracing operations in DLGs. Our main goal in this article is to show that many further (DR$\mathcal{L}$H) net operations are of interest for a complete library. For example, the generalized set intersection of the previous DR$\mathcal{L}$H with $\lfloor(r, c, b, a), (s, a, b, a, c)\rfloor$ will return $\lfloor(r, c, b, a)\rfloor$. The operations are defined as RELFUN [Bol90] pattern-matching rules on a slightly modified term representation of DR$\mathcal{L}$Hs.

Two main classes of operations on nets have to be distinguished. Operators can take network pieces as input arguments, and (1) **return** (functionally) or **bind** (relationally) other pieces as output values (declarative operations), or (2) **effect** state changes in a knowledge base (imperative operations).

Research in programming languages since [Bac78] suggests that declarative, side-effect-free operators are easier to understand, maintain, and parallelize than imperative ones. Transferring this to knowledge processing, a promising approach consists in defining most operators as declarative knowledge-item transformations, and clearly separating them from the remaining imperative knowledge-base updates. Besides FP-like functional languages [Bac78] and PROLOG-like relational languages [Col83], more specific declarative tools such as graph grammars [EHK] can be used for processing semantic networks.

For DR$\mathcal{L}$H processing we will make a mostly functional use of the relational/functional language RELFUN: high-level nested-term representations of these generalized graphs become the arguments and returned values of functions. Many such declarative term-rewriting operations on DR$\mathcal{L}$Hs are defined using RELFUN's "valued clauses" as pattern-matching rules (sections 3-7); some imperative DR$\mathcal{L}$H-update operations are introduced via `assert`-like primitives (section 8).

After a derivation of DR$\mathcal{L}$Hs from list sets (section 2), our first use of RELFUN will be the normalization of algebraic DR$\mathcal{L}$H terms employing rules that generalize set-like **duplicate removal** and **canonical ordering** (section 3). Also, high-level methods of sharing common DR$\mathcal{L}$H parts using logical variables and an 'unpack' operator are given (section 4). To provide an often-needed subpackage, standard set operations are generalized to the (hyper)graph-theoretical framework (appendix A).

We will then discuss two classes of operations which critically depend on the full power of DR$\mathcal{L}$Hs: structure-reducing operations are used for analyzing complex DR$\mathcal{L}$Hs (section 5) and path searching is extended to traverse arbitrary-length **hyper**arcs and the leveled structure of **recursive** graphs (section 6).

Although our emphasis will be on such structural operations on DR$\mathcal{L}$Hs, we will also sketch principles of applying these generalized graphs to real problems from belief sharing to public transportation. As an application in the domain of mechanical engineering we discuss the generation of parts lists from DR$\mathcal{L}$H representations of workpieces (section 7). In any case, we try to illustrate all abstract concepts by concrete examples.

The conclusions will provide additional background on the DR$\mathcal{L}$H/RELFUN formalism, and compare it with related work (section 9).

2 From Sets to DLGs and DR$\mathcal{L}$Hs

Since declarative specification of transformations has been best explored for functions on—finite—lists (e.g. pure LISP or PROLOG) and sets (e.g. standard LISP/PROLOG packages), it would be nice if these data types could be used as a basis for network processing. Indeed, we can regard an arbitrary set like $\{nail, stone, scissors, paper, terminal\}$ itself as a degenerate graph consisting only of isolated nodes. Suppose we would now like to introduce the (directed) graph links $stone \rightsquigarrow scissors$, $scissors \rightsquigarrow paper$, and $paper \rightsquigarrow stone$ in order to represent the three *win* relationships of the children's game "Stone, Scissors, Paper". Fig. 3 depicts the resulting *Directed Graph* as an Euler-Venn diagram of the original set augmented by three arrows. In the symbolic representation we replace some isolated nodes by (ordered) lists, obtaining

$\{nail, (stone, scissors), (scissors, paper), (paper, stone), terminal\}$

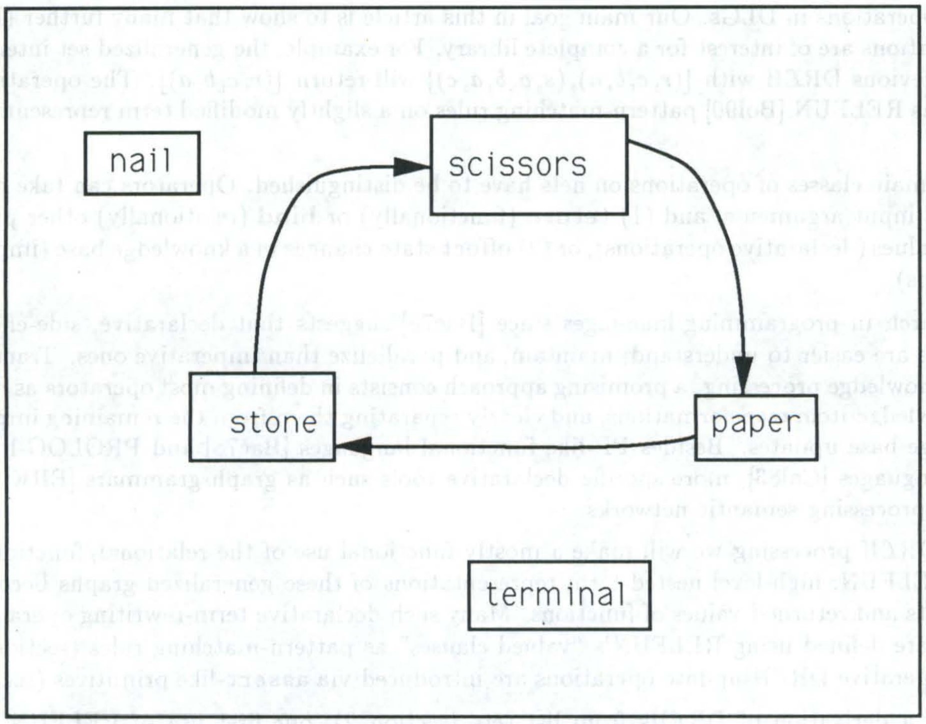

Figure 3: A directed graph with three (cyclically) linked and two isolated nodes

However, for such a heterogeneous collection of list (pair) and non-list elements it must be made explicit whether it still represents a set, keeping 'curly' brackets "{...}", or now denotes a directed graph, introducing 'floor' brackets "⌊...⌋": two collections can be different as sets,

$$\{nail, (stone, scissors), (scissors, paper), (paper, stone), terminal\} \neq$$
$$\{nail, stone, (stone, scissors), scissors, (scissors, paper), paper, (paper, stone), terminal\}$$

but identical as graphs,

$$\lfloor nail, (stone, scissors), (scissors, paper), (paper, stone), terminal \rfloor =$$
$$\lfloor nail, stone, (stone, scissors), scissors, (scissors, paper), paper, (paper, stone), terminal \rfloor$$

This is the case since, in addition to the normalization axioms of sets (in the "{...}"-representation, generalized commutativity and idempotence), graph normalization includes joining a "quasi-isolated" node x with any identical node occurring in an arc, using term-rewriting rules like $\lfloor ..., x, ..., (x, y), ... \rfloor \longrightarrow \lfloor ..., ..., (x, y), ... \rfloor$. Thus, uniqueness is maintained for isolated nodes, whereas a non-isolated node is still represented for all arrows (directed arcs) in which it occurs.

If we want to make the three special *win* relationships explicit, we can proceed to *Directed Labeled Graphs* (DLGs) by labeling the arcs with relation names or, inserting the labels as first list elements:

$\lfloor nail, (sharpen, stone, scissors), (cut, scissors, paper), (wrap, paper, stone), terminal \rfloor$

Fig. 4 gives a corresponding diagram form of DLGs in which each arrow starts at the label, cuts the first node, and ends at the second node[2].

[2]The cut-style arcs anticipate the DLG generalization to directed hypergraphs; the labels are drawn as in

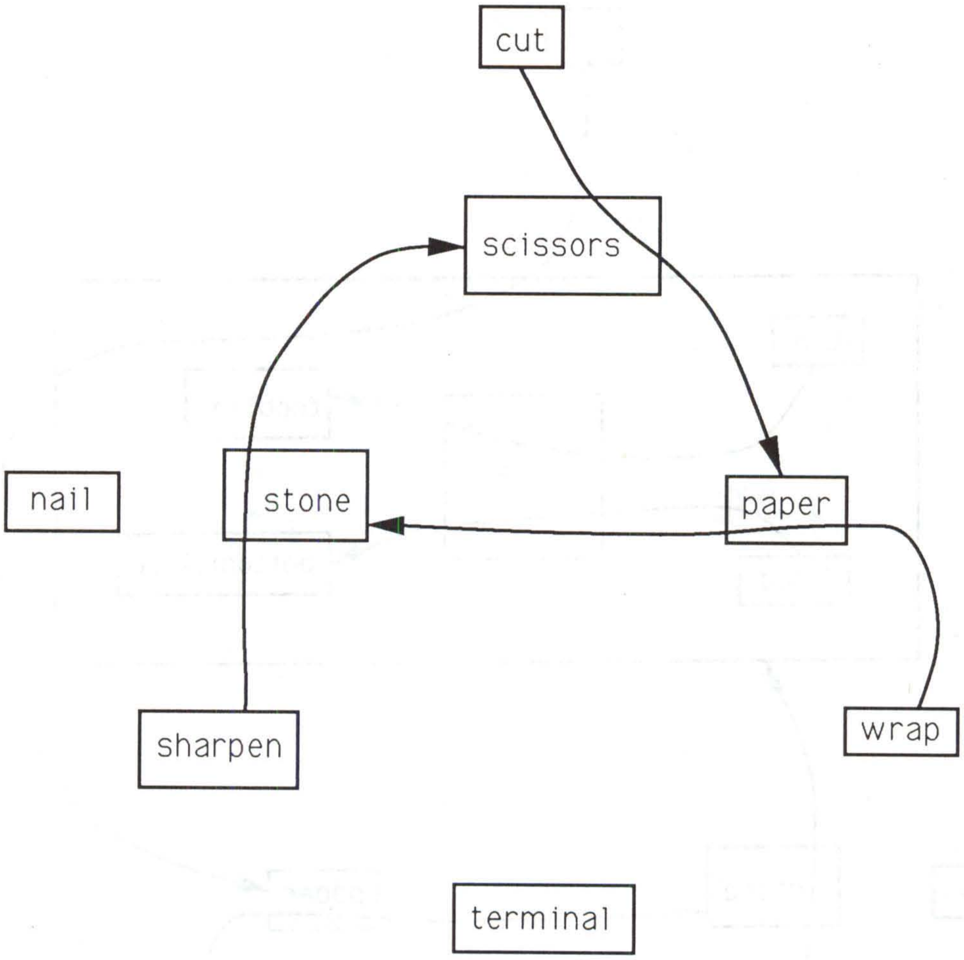

Figure 4: A DLG refinement with arc labels *sharpen*, *cut*, and *wrap* drawn like nodes

Looking at these representations of DLGs as collections of isolated nodes mixed with directed labeled arcs (lists), three graph generalizations appear very natural:

First, since set elements may again be sets, *complex nodes* can be introduced as nodes that are graphs themselves. For example, going back to our original set we can refine the elements *scissors* and *terminal* to embedded sets[3]: $\{nail, stone, \{axle, bottomblade, topblade\}, paper, \{keyboard, screen\}\}$. The new set can already be regarded as a degenerate *recursive* graph consisting only of isolated atomic and complex nodes (atoms and complexes). Besides the external arcs of the previous DLG we can also insert directed labeled arcs describing the **internal** structure of the complex nodes *scissors* ("*axle* is *fixed* at *bottomblade*", "*topblade turns* around *axle*") and *terminal* ("*keyboard* is *wired* to *screen*"), thus obtaining the *Directed Recursive Labeled Graph* ('pretty-print' indentation will be used to enhance the readability of line-exceeding linear representations):

labelnode graphs. On the other hand, Euler-Venn-like boundary lines are only kept for sublevels of recursive graphs.

[3] Since an embedded set carries no mark, we lose unrefined-element names like *scissors* and *terminal* at this point. This could be avoided, e.g., by using RELFUN variable names like `Scissors` and `Terminal` of section 4 as DR$\mathcal{L}$H labelnodes marked by the (complex) labelnodes that are their values.

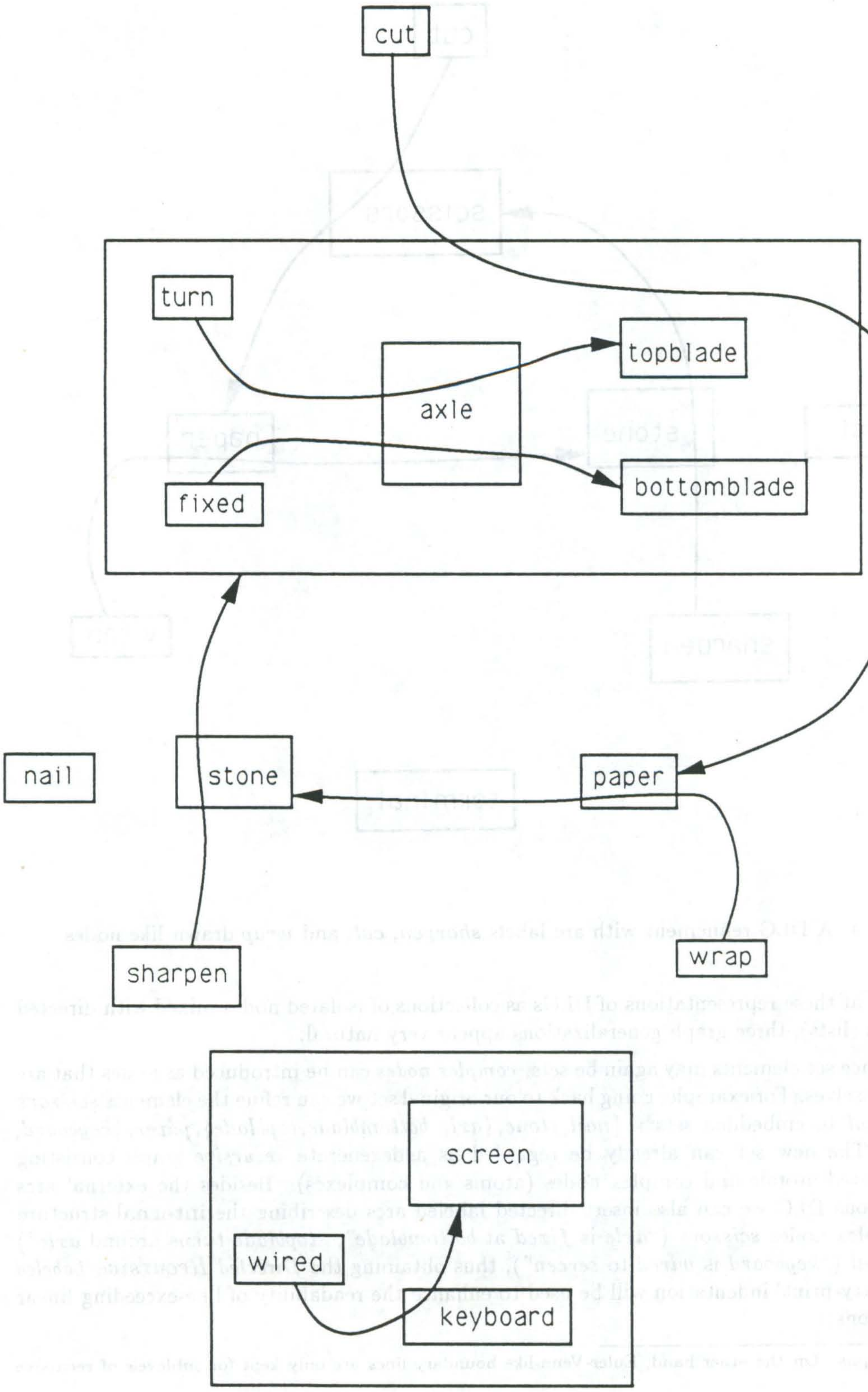

Figure 5: A directed recursive graph with *scissors* and *terminal* expanded to complex nodes

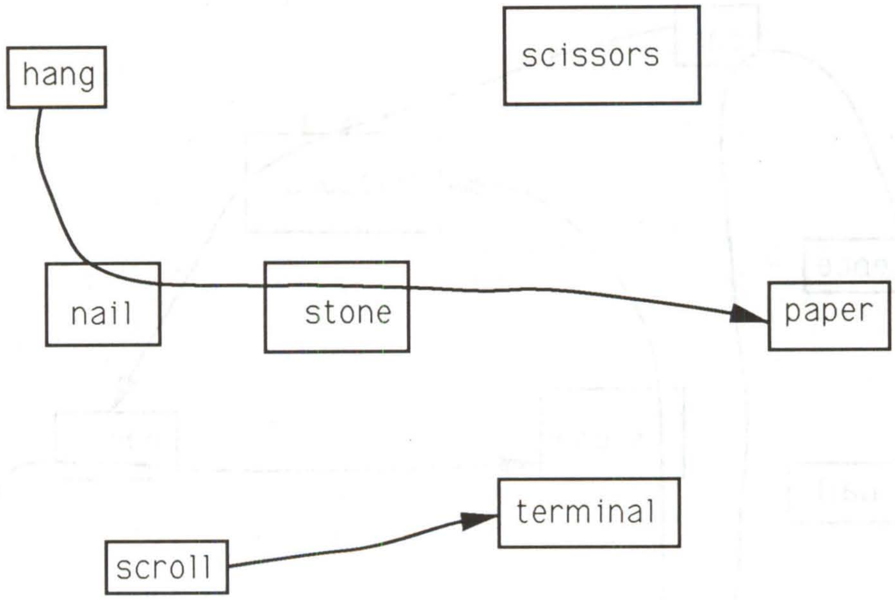

Figure 6: A directed hypergraph with *hang/scroll*-labeled hyperarcs of lengths three/one

$\lfloor nail,$
$(sharpen, stone, \lfloor (fixed, axle, bottomblade), (turn, axle, topblade) \rfloor),$
$(cut, \lfloor (fixed, axle, bottomblade), (turn, axle, topblade) \rfloor, paper),$
$(wrap, paper, stone),$
$\lfloor (wired, keyboard, screen) \rfloor \ \rfloor$

The diagram form in Fig. 5 indicates a complex node as a boundary line completely boxing in all its arrows, labels, and nodes.

Second, since lists may have $n \neq 2$ elements after the label-representing first element, *directed hyperarcs* can be introduced as arcs that link an arbitrary number of $n \geq 0$ nodes. (The degenerate case $n = 0$ corresponds to a nullary relationship like $night()$; the special case $n = 1$ permits the direct—"non-**isa**"—representation of a unary relationship like $bright(sun)$, as utilized below and discussed in section 7.) For instance, we can also structure the original set $\{nail, stone, scissors, paper, terminal\}$ by inventing a ternary *hang* relationship ("*nail* and *stone hang paper*") and a unary *scroll* relationship ("*terminal scrolls*"), obtaining the *Directed Labeled Hypergraph*:

$\lfloor (hang, nail, stone, paper), scissors, (scroll, terminal) \rfloor$

The diagram form in Fig. 6 depicts each directed hyperarc as an arrow starting from the label, cutting all intermediate nodes, and ending at the final node. (In the special case $n = 1$ the label directly points to the single node, which looks like an ordinary unlabeled arc but actually depicts a labeled length-one hyperarc.) Of course, since arcs are special hyperarcs, we could likewise have extended the DLG in Fig. 4 to a directed hypergraph, as implicit in Fig. 8.

Third, since relation names may occur not only as first list elements (labels) but also as arguments of other relationships (nodes), *labelnodes* can be introduced as uniform base objects usable as labels, nodes, or both. The earlier DLG example can thus be extended by a second-order *preference* relation between the *win* relations ("*preference* of *sharpen*ing over *wrapping*", ...), obtaining the *Directed Labelnode Graph*:

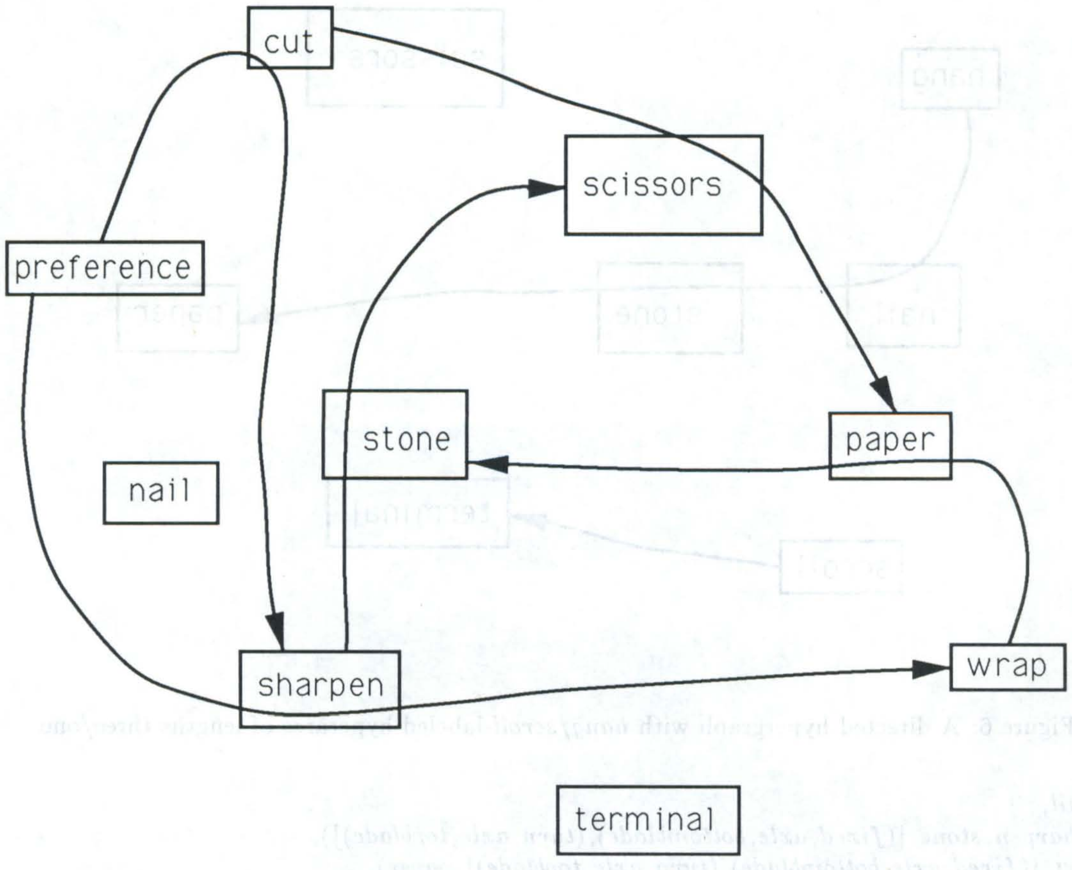

Figure 7: A directed labelnode graph with labels also used as nodes of *preference* arcs

$\lfloor$*nail*,
(*sharpen*, *stone*, *scissors*),
(*cut*, *scissors*, *paper*),
(*wrap*, *paper*, *stone*),
terminal,
(*preference*, *sharpen*, *wrap*),
(*preference*, *cut*, *sharpen*) $\rfloor$

The diagram form in Fig. 7 shows each labelnode as a box which may be used at arbitrary positions of arrows.

Bringing all three DLG generalizations together we obtain *Directed Recursive $\mathcal{L}$abelnode Hypergraphs* (DR$\mathcal{L}$Hs). For instance, this is a DR$\mathcal{L}$H combination of the previous examples with a *color screen* for the *terminal* and two new *preference* relations:

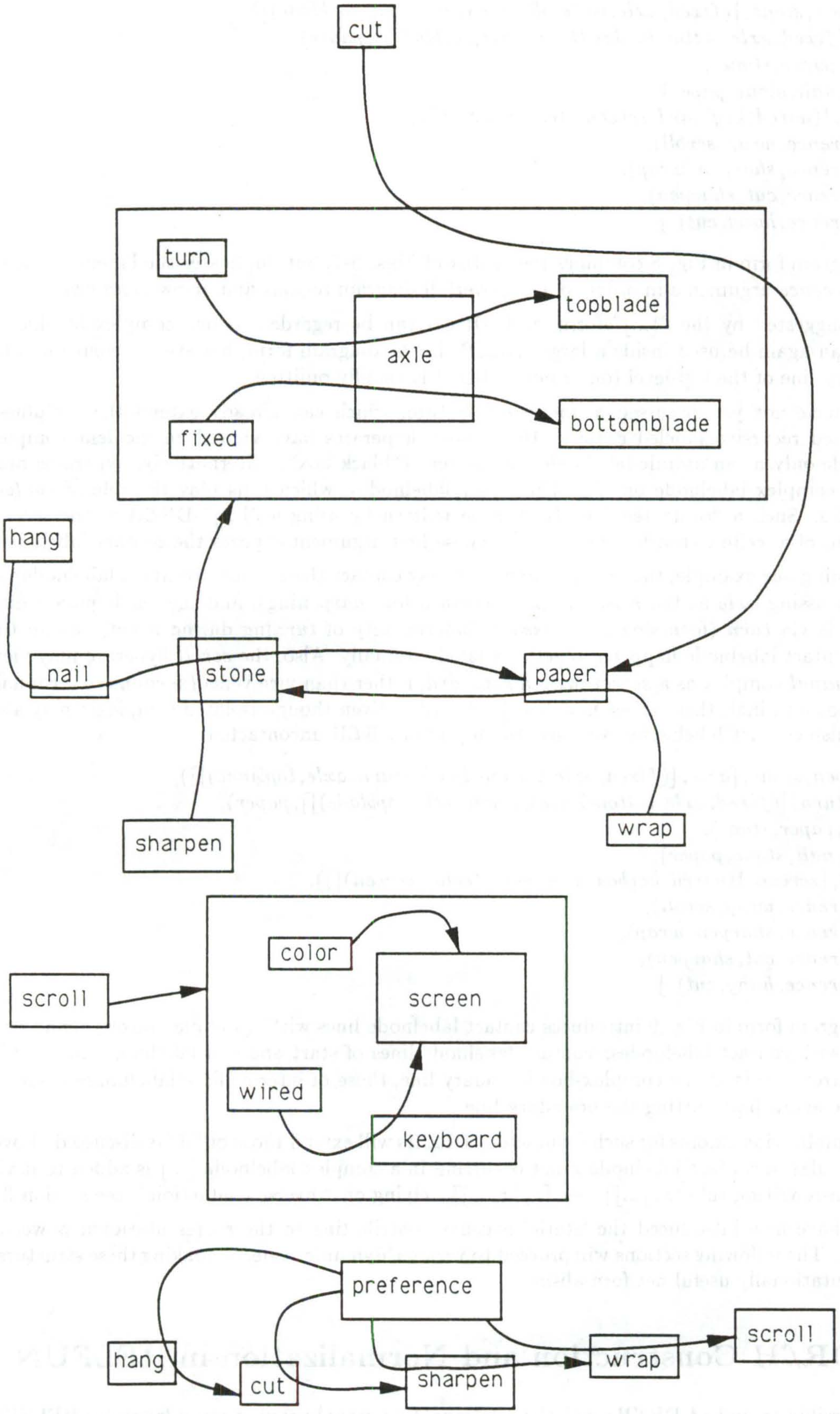

Figure 8: A DR$\mathcal{L}$H synthesizing the recursive, 'hyper', and labelnode extensions of DLGs

$\lfloor(sharpen, stone, \lfloor(fixed, axle, bottomblade), (turn, axle, topblade)\rfloor),$
$(cut, \lfloor(fixed, axle, bottomblade), (turn, axle, topblade)\rfloor, paper),$
$(wrap, paper, stone),$
$(hang, nail, stone, paper),$
$(scroll, \lfloor(wired, keyboard, screen), (color, screen)\rfloor),$
$(preference, wrap, scroll),$
$(preference, sharpen, wrap),$
$(preference, cut, sharpen),$
$(preference, hang, cut)\ \rfloor$

The diagram form in Fig. 8 combines the syntax of Figs. 5-7, but duplicates the labelnodes used as *preference* arguments in order to avoid overfull diagram regions and arrow crossings.

As suggested by the "⌊...⌋"-form, each DR$\mathcal{L}$H can be regarded as one complex labelnode, which can again be used inside a larger DR$\mathcal{L}$H. In the diagram form, however, the surrounding boundary line of the top-level (outermost) DR$\mathcal{L}$H is usually omitted.

We have not yet discussed a 'focussing' feature, which can already extend the usefulness of directed recursive labeled graphs. Up to now, hyperarcs have viewed an incident complex labelnode only as an atomic-labelnode-like entirety ("black box"); alternatively, hyperarcs may focus a complex labelnode on any of its inner labelnodes, which thus play the role of *contact labelnodes*. Such a "contacted DR$\mathcal{L}$H" will be written by using a "⌊...⌋"-DR$\mathcal{L}$H as the second argument of a 'ceiling'-bracket term "⌈...⌉" whose first argument exposes the contact labelnode.

Refining our example, the *sharpen* hyperarc may contact the *scissors* complex labelnode via *axle* (focussing *axle* as the *scissors*' part to grasp for *sharpen*ing), and the *cut* hyperarc may contact it via *turn* (focussing the *scissors*' functionality of *turn*ing during a *cut*), where the latter contact labelnode happens to act as a label internally. Also, the *scroll* hyperarc may view the *terminal* complex as a *screen* with a *keyboard*, rather than vice versa (**screens** of terminals scroll, not terminals themselves, nor their keyboards). Even though isolated complexes may also distinguish contact labelnodes, we leave the top-level DR$\mathcal{L}$H uncontacted:

$\lfloor(sharpen, stone, \lceil axle, \lfloor(fixed, axle, bottomblade), (turn, axle, topblade)\rfloor\rceil),$
$(cut, \lceil turn, \lfloor(fixed, axle, bottomblade), (turn, axle, topblade)\rfloor\rceil, paper),$
$(wrap, paper, stone),$
$(hang, nail, stone, paper),$
$(scroll, \lceil screen, \lfloor(wired, keyboard, screen), (color, screen)\rfloor\rceil),$
$(preference, wrap, scroll),$
$(preference, sharpen, wrap),$
$(preference, cut, sharpen),$
$(preference, hang, cut)\ \rfloor$

The diagram form in Fig. 9 introduces contact labelnode lines within complex boxes, connecting arrows with contact labelnodes: contact labelnode lines of start and end labelnodes have additional arrow heads at the complex-box boundary line, those of intermediate labelnodes emanate from the arrow part cutting the boundary line.

Normalization axioms for such (contacted) DR$\mathcal{L}$Hs will extend those of DLGs discussed above. In particular, a contact labelnode x not occurring in a complex labelnode $\lfloor...\rfloor$ is added to it via the term-rewriting rule $\lceil x, \lfloor...\rfloor\rceil \longrightarrow \lceil x, \lfloor x, ...\rfloor\rceil$, relying on "inverse contaction" (see section 3).

We have now introduced the 'static' features contributing to the representational power of DR$\mathcal{L}$Hs. The following sections will proceed to various 'dynamic' aspects, making these structures a computationally useful net formalism.

3 DR$\mathcal{L}$H Construction and Normalization in RELFUN

It is possible to embed DR$\mathcal{L}$Hs into the relational/functional programming language RELFUN and at the same time provide a formal, 'constructor-algebraic' DR$\mathcal{L}$H definition. First, "⌈...⌉"-,

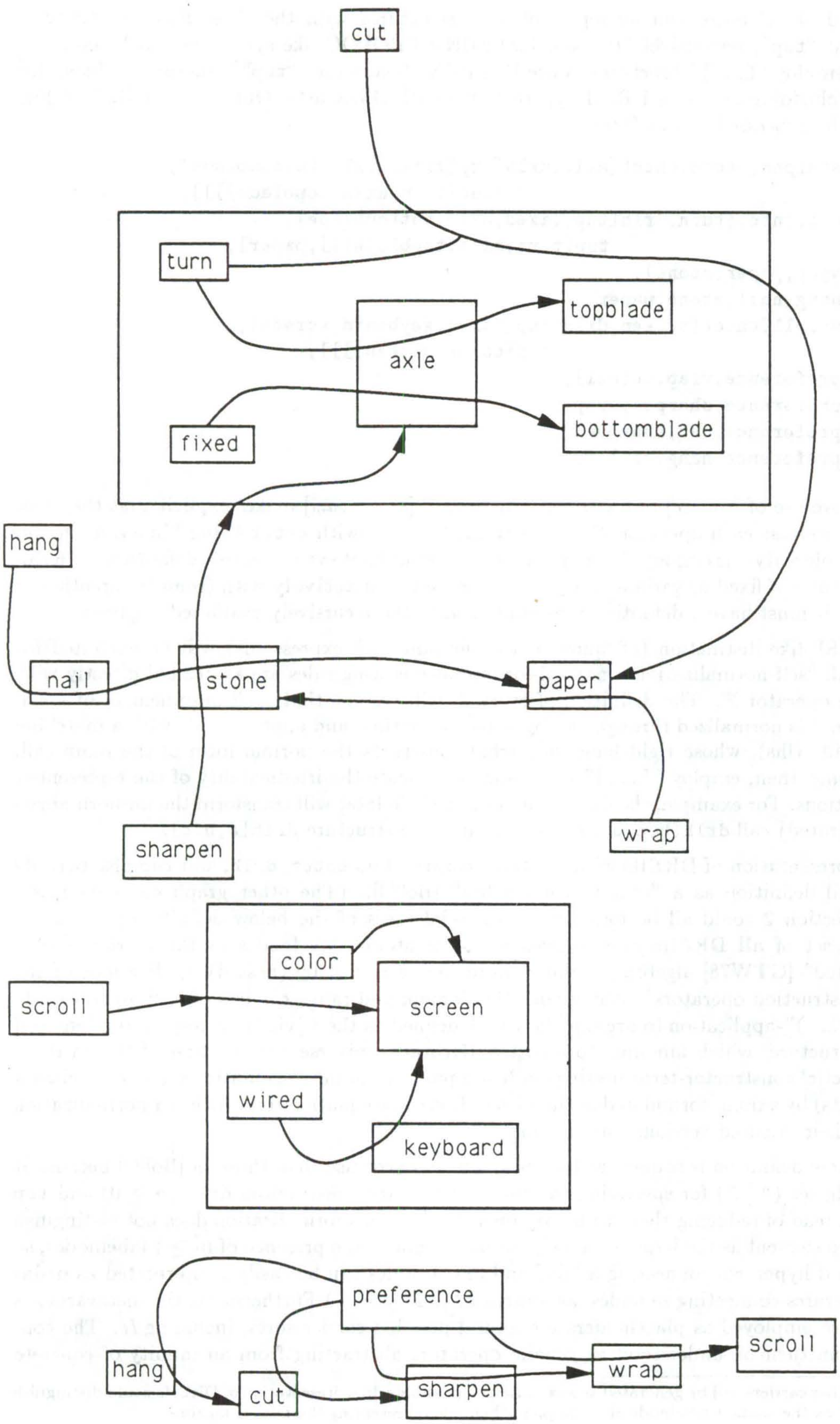

Figure 9: A refined DR$\mathcal{L}$H with double/single-contacted *scissors/terminal* complexes

"⌊...⌋"-, and "(...)"-terms can be represented as structures with the three functors "**cntct**", "**drlh**", and "**tup**", respectively (we use RELFUN's PROLOG-like syntax in which, however, structures employ "[...]"-brackets). Since RELFUN already uses "**tup**"-structures as lists, this language embedding identifies DR$\mathcal{L}$H hyperarcs with RELFUN lists. Our sample DR$\mathcal{L}$H of Fig. 9 can then be processed in this form:

```
drlh[tup[sharpen,stone,cntct[axle,drlh[tup[fixed,axle,bottomblade],
                                       tup[turn,axle,topblade]]]],
     tup[cut,cntct[turn,drlh[tup[fixed,axle,bottomblade],
                             tup[turn,axle,topblade]]],paper],
     tup[wrap,paper,stone],
     tup[hang,nail,stone,paper],
     tup[scroll,cntct[screen,drlh[tup[wired,keyboard,screen],
                                  tup[color,screen]]]],
     tup[preference,wrap,scroll],
     tup[preference,sharpen,wrap],
     tup[preference,cut,sharpen],
     tup[preference,hang,cut] ]
```

The above use of [square] brackets for structures $\mathcal{F}[a_1,\ldots,a_m]$ makes explicit that they just denote themselves: each operator $\mathcal{F} \in \{\texttt{cntct}, \texttt{drlh}, \texttt{tup}\}$—with **cntct** being binary, **drlh** and **tup** of variable arity—is employed here **passively**; it would not even require a definition. A RELFUN operator—of fixed or variable arity—can also be called **actively** with (round) parentheses; in this case it must have a definition that is applied to the recursively evaluated arguments.

This LISP-like distinction (of '**quoted**' vs. 'non-**quoted**' expressions) will be exploited for what we call "self-normalization": normal-form term-rewriting rules are associated directly with every main operator $\mathcal{F}$. The definition of every $\mathcal{F}$ will assume that each argument a_i of a call $\mathcal{F}(a_1,\ldots,a_m)$ is normalized through call-by-value evaluation, and applies a rule with a matching left-hand side (lhs), whose right-hand side (rhs) constructs the normal form of the main call. Normal forms, then, employ "[...]"-structures to indicate the irreducibility of the represented data collections. For example, the definition of **drlh** given later will transform the un-normalized (set-degenerated) call **drlh(b,c,b,a)** to the normalized structure **drlh[a,b,c]**.

Our representation of DR$\mathcal{L}$Hs with the three constructors **cntct**, **drlh**, and **tup** also permits their formal definition as a "constructor algebra" [Bol84]. (The other graph concepts introduced in section 2 could all be formalized as special cases of the below definition.) Here, we regard the set of **all** DR$\mathcal{L}$Hs over a given set $\mathcal{A}$ of atomic labelnodes as the carrier $\mathcal{U}$ of a "many-sorted" [GTW78] algebra generated from the carrier $\mathcal{A}$ by (nested) applications of the DR$\mathcal{L}$H-construction operators[4]. Along with the domain and range carriers of each operator, its (active) "(...)"-application to arguments will be defined as the trivially corresponding [passive] "[...]"-structure, which amounts to a sorted Herbrand-universe construction of the carriers. The 'syntactic' constructor-term nestings in $\mathcal{U}$ are partitioned into 'semantic' equivalence classes (or quotients) by axioms formulated as equalities. It is these equations on which our normalization rules—as their oriented versions—are relying.

The below definition is somewhat less rigid but more concise than those in [Bol84] because it employs ellipses ("...") for specifying the n-ary and m-ary constructors **drlh** ($n \geq 0$) and **tup** ($m \geq 1$) instead of reducing them to binary operators[5]. (This formalization does not distinguish the first **tup** element as the hyperarc label, but only requires the presence of $m \geq 1$ labelnodes, so that **labeled** hyperarcs connecting a label and $m-1$ nodes can be easily reinterpreted as **unlabeled** hyperarcs connecting m nodes, as illustrated in section 5.) Furthermore, the metavariables $\mathcal{L}$ and $\mathcal{P}$ are employed as placeholders for several possible carrier sorts, including $\mathcal{U}$. The constructors can then be understood as generic operators abstracting from an infinity of concrete

[4] Two further carriers will be generated as auxiliaries. A possible self-representation of DR$\mathcal{L}$Hs could distinguish the carrier $\mathcal{U}$ as the contact labelnode of a complex labelnode representing the DR$\mathcal{L}$H algebra.

[5] Also, we now rely on commutativity for preparing the application of other axioms, and axiomatize contact labelnodes via binary **cntct** structures rather than unary tags.

operators for each fixed arity and argument sort. Object variables are written as (possibly indexed) small letters, e.g. l_1, which are implicitly typed by the (meta)sort with the corresponding capital letter, e.g. $\mathcal{L}$.

Definition 1 (The Constructor Algebra of DR$\mathcal{L}$Hs)

Given a finite carrier

$\mathcal{A}$*: Atomic labelnodes*

three further carriers

$\mathcal{H}$*: Hyperarcs*
$\mathcal{U}$*: Uncontacted complex labelnodes (the set of DR$\mathcal{L}$Hs over $\mathcal{A}$)*
$\mathcal{C}$*: Contacted complex labelnodes*

are generated through mutually inductive application of three corresponding constructors ($\mathcal{L}$—'$\mathcal{L}$abelnodes'—standing for $\mathcal{A}$ or $\mathcal{U}$ or $\mathcal{C}$, and $\mathcal{P}$—'$\mathcal{P}$ieces'—for $\mathcal{L}$ or $\mathcal{H}$):

$$tup : \overbrace{\mathcal{L} \times \mathcal{L} \times \ldots \times \mathcal{L}}^{m \geq 1} \rightarrow \mathcal{H}$$
$$tup(l_1, l_2, \ldots, l_m) = tup[l_1, l_2, \ldots, l_m]$$
$$drlh : \overbrace{\mathcal{P} \times \mathcal{P} \times \ldots \times \mathcal{P}}^{n \geq 0} \rightarrow \mathcal{U}$$
$$drlh(p_1, p_2, \ldots, p_m) = drlh[p_1, p_2, \ldots, p_m]$$
$$cntct : \mathcal{L} \times \mathcal{U} \rightarrow \mathcal{C}$$
$$cntct(l, u) = cntct[l, u]$$

The following axioms are postulated for the constructor terms:

$drlh[\ldots, p, p', \ldots] = drlh[\ldots, p', p, \ldots]$ *(commutativity of* `drlh`*)*

$drlh[\ldots, p, p, \ldots] = drlh[\ldots, p, \ldots]$ *(idempotence of* `drlh`*)*

$drlh[\ldots, tup[\ldots, l, \ldots], l, \ldots] = drlh[\ldots, tup[\ldots, l, \ldots], \ldots]$ *(adsorption of labelnode by* `tup`*)*

$drlh[\ldots, cntct[l, u], u, \ldots] = drlh[\ldots, cntct[l, u], \ldots]$ *(similpotence of* `cntct` *and* `drlh`*)*

$cntct[l, drlh[l, \ldots]] = cntct[l, drlh[\ldots]]$ *(contaction of labelnode by* `cntct`*)*

All equations except the term-size-preserving first one decrease term size if read from left to right. Except for the last equation this term-size-decreasing orientation is also used for the corresponding normal-form term-rewriting rules. The reason for the inverse (term-size-increasing) use of contaction is to have all labelnodes of a DR$\mathcal{L}$H represented within the `drlh` term, restricting the role of the `cntct` term to the distinction of one of them.

Before proceeding to the RELFUN definitions of the DR$\mathcal{L}$H constructors, let us see how simple term-rewriting rules and their call patterns are specified in this language:

> A **rule** *lhs* $\longrightarrow$ *rhs* is written *lhs* `:-&` *rhs*. Here "`&`" indicates that the rhs returns a value (while the rhs of PROLOG's "`:-`" generates bindings only).
>
> A **pattern** $\mathcal{F}(a_1, \ldots, a_m, x_1, x_2, \ldots)$, with "$a_1, \ldots, a_m$" matching $m \geq 0$ fixed elements and "$x_1, x_2, \ldots$" matching a 'rest' of zero or more further elements, is written $\mathcal{F}$`(`a_1`,...,`a_m` | `X`)`. Here "`|`" indicates that the variable X binds the entire 'rest' as a single list (hyperarc) `tup[`x_1`,`x_2`,...]`. For $m = 0$, the often needed special form $\mathcal{F}(x_1, x_2, \ldots)$ looks like $\mathcal{F}$`(|`X`)`. (Generalizing LISP's dot and PROLOG's vertical bar, RELFUN's "`|`" can (1) occur in lists, arbitrary structures, or even calls and (2) follow directly after a bracket or a parenthesis.)

The definition for `tup` calls embodies the "identity" transformation of self-normalization: `tup(`y_1`,`y_2`,...)` $\longrightarrow$ `tup[`y_1`,`y_2`,...]`. In RELFUN the lhs becomes a pattern `tup(|Y)`. This uses the functor `tup` and the 'rest' variable `Y`, matching its zero or more arbitrary arguments. Similarly, the rhs becomes tup[|Y], splicing the 'rest' value back into a—now passive—tup term[6]. Together, this leads to the following RELFUN clause:

```
tup(|Y) :-& tup[|Y].
```

Thus, to construct an arc (three-element list) with—say—label 1, first node 2*1, and second node 2+1, we can evaluate tup(1,2*1,2+1), which returns tup[1,2,3].

We will often need a LISP-cons-like DR$\mathcal{L}$H constructor, which is defined to match arbitrary drlh structures in its second argument. This consdrlh function only preserves normal forms if its first argument is to simply extend the second argument by a new 'front' (head) element X:

```
consdrlh(X,drlh[|R]) :-& drlh[X|R].
```

For example, the call consdrlh(tup(1,2*1,2+1),drlh(b,c,b,a)) returns the normal form drlh[tup[1,2,3],a,b,c][7].

The definition of the central drlh constructor is done here by a kind of insertion sort with two merging functions: mergearrow for hyperarcs and mergebox for labelnodes. As a special case this includes set normalization, i.e. duplicate removal (relying on idempotence) and canonical ordering (relying on commutativity). For the general case of DR$\mathcal{L}$H normalization hyperarcs remove quasi-isolated labelnodes (relying on "adsorption" [Bol84]): in mergearrow, the hyperarc argument erases all occurrences of its labelnodes found in the top-level of the DR$\mathcal{L}$H argument; in mergebox, the labelnode argument is discarded if it is found in a hyperarc of the DR$\mathcal{L}$H argument. In the canonical ordering for DR$\mathcal{L}$Hs, hyperarcs are "less than" (to the left of) isolated labelnodes, permitting one-pass (look-ahead-free) merging even for mergebox: only after having 'survived' the prefix of tup[...] terms, need a labelnode be inserted into the suffix of isolated labelnodes. Details are given in appendix B.

The main drlh function can now be defined as alternating insertions of its hyperarc and isolated-labelnode argument fronts into its recursively normalized argument remainders. The first two clauses use a PROLOG-like 'neck' (or 'initial') cut, "!", for 'committing' callers directly after a successful lhs match[8]; since no general cut operator will be needed here, "!" is not written as the first rhs premise but is encoded into the neck operator, obtaining "!-&".

```
drlh()            !-&  drlh[].
drlh(tup[|Y]|R)   !-&  mergearrow(tup[|Y],drlh(|R)).
drlh(B|R)         :-&  mergebox(B,drlh(|R)).
```

For instance, both the calls drlh(b,2,tup(1,2,3),drlh(a,b,b,c),1,tup(2,2),4) and drlh(1,tup(1,2,3),4,drlh(c,a,b),3,tup(2,2),b,tup(1,2,3)) normalize to the structure drlh[tup[1,2,3],tup[2,2],drlh[a,b,c],4,b]. This shows that keeping DR$\mathcal{L}$Hs in normal form permits subsequent equality tests being performed in linear time ("[...]"-structures must agree character by character), just as in the special case of sets.

The definition of cntct uses the function mergebox to add the contact-labelnode argument B to the drlh argument if it is not there already (only the value of the conjunct after "&" is returned).

```
cntct(B,drlh[|R]) :- D is mergebox(B,drlh[|R]) & cntct[B,D].
```

As an illustration, cntct(a,drlh(a,b,c,a,b,c)) reduces to cntct[a,drlh[a,b,c]], while cntct(d,drlh(a,b,c,a,b,c)) rewrites to cntct[d,drlh[a,b,c,d]].

---

[6] Since Y's value must have the 'rest' form tup[y_1,y_2,...], the rhs could be simplified: it always instantiates to tup[|tup[y_1,y_2,...]], which is "|"-spliced to Y itself. In general, for any variable X = tup[x_1,x_2,...] and any functor $\mathcal{F}$, the equality $\mathcal{F}$[|X] = $\mathcal{F}$[x_1,x_2,...] holds, which for $\mathcal{F}$ = tup specializes to tup[|X] = X.

[7] After call-by-value normalization of the arguments, consdrlh(tup[1,2,3],drlh[a,b,c]) is matched by the lhs, binding R to the 'rest' tup[a,b,c]; the rhs "|"-splices drlh[tup[1,2,3]|tup[a,b,c]] to the result.

[8] Even in a declarative language this restricted cut use is beneficial for local determinism specification: it just prevents "shallow backtracking" to the remaining clauses within an operator definition.

Returning to the sample DR$\mathcal{L}$H, it should be noted that it is not completely normalized because at the bottom line the **eless** predicate called in the **mergearrow** and **mergebox** definitions of appendix B performs lexicographic comparison. For instance, because of **eless(color,wired)**, call-by-value evaluation of the *terminal* complex labelnode

```
cntct(screen,drlh(tup(wired,keyboard,screen),
                  wired,
                  color,
                  wired,
                  tup(color,screen),
                  wired,
                  tup(color,screen),
                  keyboard))
```

would return

```
cntct[screen,drlh[tup[color,screen],
                  tup[wired,keyboard,screen]]]
```

4 Labelnode Sharing

In the compact diagram forms of DR$\mathcal{L}$Hs, a single labelnode box need physically appear only once but can participate in several hyperarc arrows; if it is complex, it may also have multiple contact-labelnode views as well as overlaps with other complex-labelnode boxes. In symbolic linearizations, however, extra copies are normally made necessary for each such use of a labelnode. This is due to the fact that in the two (or three) dimensions of a diagram there are infinitely many 'directions' from which to access a box, while in the single dimension of a string or term there are only two. The general issue for semantic net formalisms here is how to represent such **sharing** of entities.

Programming languages that allow copy-free representations often do this with non-declarative constructs such as *explicit pointers*. For instance, in LISP, **rplaca**-like destructive operations could be employed to mimic directed graphs. However, the cyclic pointer structures thus created are hard to debug or even print. Similarly, LISP *property lists* can directly represent DLGs via the hashing mechanism for LISP atoms (DLG nodes). But most of the **setf-get**-like operations for their processing cause (global!) side-effects. Also, neither of these representations is easily extended to all kinds of sharing possible in DR$\mathcal{L}$Hs.

Therefore, we propose a DR$\mathcal{L}$H use of *logical variables*, PROLOG's declarative substitute for pointers, as combined with functional value returning in RELFUN[9]: like mathematical variables, these are names that can be transparently substituted with their values, in contrast to the reassignable variables of procedural programming. For the sharing of fixed (complex) labelnodes only part of the expressiveness of terms with logical variables (*non-ground terms*) is required; we only touch on the more general non-ground DR$\mathcal{L}$Hs and do not treat the issue of set (ACI) unification enhancements for the characteristic DR$\mathcal{L}$H properties such as adsorption.

Atomic labelnodes are not often worth a shared user-level representation with logical variables (most languages implement symbol tables with hashing); still there should be the possibility of writing down a long non-isolated atomic labelnode only once, subsequently using a variable in the hyperarc positions in which it occurs. If we want to share a labelnode like **very-long-atom** in this fashion, we bind a new (shorter) variable name **V** to it, calling the RELFUN **is**-primitive by **V is very-long-atom**. All occurrences of **very-long-atom** in any hyperarc structure **tup[...,very-long-atom,...,very-long-atom,...]** are then replaced by **V** occurrences, thus obtaining **tup[...,V,...,V,...]**.

[9]Since RELFUN's logical variables are implemented in LISP, there is an implicit system-level use of LISP's shared pointer structures.

Complex labelnodes can be shared similarly. Even if a complex labelnode is used with several different contact labelnodes, it is possible to share its common `drlh` subterm. For sharing the complex labelnode `drlh[...]`, a new variable name `D` is bound to it via `D is drlh[...]`. If `drlh[...]` occurs with contact labelnodes `b1`, ..., `bN`, i.e. in `cntct[b1,drlh[...]]`, ..., `cntct[bN,drlh[...]]`, the `cntct` terms are replaced by `cntct[b1,D]`, ..., `cntct[bN,D]`; occurrences of `drlh[...]` without contact labelnodes are replaced by `D` occurrences, like atoms.

As an example for atomic and complex labelnode sharing let us extract the atom `preference` as well as the `drlh` subterms of the doubly contacted *scissors* complex and the singly contacted *terminal* complex from the RELFUN form, shown in section 3, of our sample DR$\mathcal{L}$H, depicted in Fig. 9:

```
Scissors is drlh[tup[fixed,axle,bottomblade],tup[turn,axle,topblade]],
Terminal is drlh[tup[wired,keyboard,screen],tup[color,screen]],
Pref is preference &
drlh[tup[sharpen,stone,cntct[axle,Scissors]],
     tup[cut,cntct[turn,Scissors],paper],
     tup[wrap,paper,stone],
     tup[hang,nail,stone,paper],
     tup[scroll,cntct[screen,Terminal]],
     tup[Pref,wrap,scroll],
     tup[Pref,sharpen,wrap],
     tup[Pref,cut,sharpen],
     tup[Pref,hang,cut] ].
```

Of course, in the above example the three `cntct` terms with variables as second arguments could again be named by unique variables, and, finally, the top-level `drlh` term could become the value of a logical variable for use in still higher structures[10].

Moreover, each `is` call which 'sharing-abstracts' an entity to a logical variable can be transparently conjoined not only to the left (like a functional `let` expression) but also to the right (like a functional `where` expression) of the structure in which the entity occurs[11]. For instance, the (`is`-embedded) `drlh` structure

```
D is drlh[tup[1,2000000000,3],tup[2000000000,2000000000]].
```

can be shortened equivalently to the `let`-like conjunction

```
V is 2000000000, D is drlh[tup[1,V,3],tup[V,V]].
```

or to the `where`-like conjunction

```
D is drlh[tup[1,V,3],tup[V,V]],  V is 2000000000.
```

[10] The constructive character of DR$\mathcal{L}$Hs, obvious from both their diagram and symbolic forms (also captured algebraically in definition 1), prevents the 'self-containment' of complex labelnodes: infinite descending membership sequences of complex labelnodes cannot be expressed in the DR$\mathcal{L}$H formalism proper; DR$\mathcal{L}$Hs, like Zermelo-Fraenkel sets, are *well-founded*. This foundation axiom is preserved by DR$\mathcal{L}$H sharing with purely logical variables because no such variable may be bound to a term—eventually—containing this same variable (*occur-check* property). However, like most PROLOG implementations, the present RELFUN implementation omits the occur check for efficiency reasons. This could be sanctioned by reinterpreting circular bindings like `Self is drlh[tup[escape,Self,imagination]]` as "rational trees" [Col83], and the corresponding complex labelnodes as DR$\mathcal{L}$H-generalized "non-well-founded sets" [Acz88]. While these issues only arise in the RELFUN embedding of DR$\mathcal{L}$Hs, names and the ensuing circularities are unavoidable in the so-called "hierarchical graphs" [Pra69]. Our algebra similarly constructs only finite-length hyperarcs, but this could be abandoned toward finitely describable hyperarcs like `Togo is tup[long,way|Togo]`. On the other hand, well-founded infinite sets like $\{0,1,2,...\}$ lead to the well-founded DR$\mathcal{L}$H generalizations of infinite complex labelnodes like `drlh[0,1,2,...]` and infinite hyperarcs like `tup[natural,0,1,2,...]`.

[11] Both `let` and `where` are syntactic variations of *λ-application* as used in LISP; our sharing concept corresponds to *λ-abstraction*.

The naming device expanded here is a 'transient' construct employed only in the symbolic DR$\mathcal{L}$H form, mirroring physically shared diagram parts by logically shared subterms. Thus, logical variables used for the purpose of DR$\mathcal{L}$H sharing can be eliminated by back substitution. A quite different issue is the 'permanent' use of variable-like devices already in the diagram form. For instance, if the `V`-assignment is omitted entirely from the above example, `D` denotes a non-ground DR$\mathcal{L}$H, whose free variable `V` would also appear as a labelnode in the corresponding diagram form. Such non-ground DR$\mathcal{L}$Hs can be used for representing, e.g., quantified predicate-logic formulas. Thus, the diagrammatic treatment of existential quantification on the basis of Peirce's (unlabeled) "lines of identity" (see [Rob]), viewable as connected graphs composed of (undirected) "coreference links" [Sow84], can be simulated with existentially interpreted labelnode variables. For example, in [Rob] the sentence "Some pain is good" is diagrammed with a single coreference link between the concepts for **pain** and **good**; its predicate-logic form, $(\exists X)\,\mathbf{pain}(X) \wedge \mathbf{good}(X)$, leads to a non-ground DR$\mathcal{L}$H with `X` as labelnode variable, `drlh[tup[pain,X],tup[good,X]]`. [Bol77] details an alternative approach toward the DR$\mathcal{L}$H treatment of predicate logic.

While the previous kind of sharing was based on hierarchic paths (recursive levels) for abstracting entities, **overlaps** of complex labelnodes can be exploited for non-hierarchic abstraction: the common pieces of two or more overlapping complexes can be shared even though they are not generally forming a single complex. To do this, we pack them into a newly created complex labelnode; but then we must enable the original labelnodes to **unpack** it, so they can use the pieces again.

In general, the **unpack** operator, a declarative feature described in [Hew77], has a data collection as its single parameter. If it is called in a data collection of the same type (as encoded in the functor), it takes the elements of its parameter data collection out to the top-level data collection. Thus, **unpack** locally simulates associativity of a non-associative data type. Its RELFUN definition has a trivial, **tup**-like, 'context-free' clause, just for permitting its call-by-value evaluation in the collection in which it will be used:

```
unpack(Collection) :-& unpack[Collection].
```

It has also a schematizable, 'context-sensitive' clause extending the normalization definition of every collection that is to be unpackable, where the lhs pattern contains the **unpack** as a structure (as produced by the 'context-free' clause); for DR$\mathcal{L}$Hs the clause, to be positioned anywhere before the final clause, calls **uniondrlh** (cf. appendix A) in its rhs to unite **unpack**'s parameter **drlh** with the remainder **drlh** (which can be used as a structure since **uniondrlh** works by normalization):

```
drlh(unpack[drlh[|Y]]|R) !-& uniondrlh(drlh[|Y],drlh[|R]).
```

As an application of complex labelnodes and the sharing of their overlaps, let us group (episodic) knowledge into individual "belief contexts". These separate the beliefs of two or more persons from each other, but may also overlap for the "shared beliefs" of certain persons. If beliefs are represented as hyperarcs of a DR$\mathcal{L}$H database, the belief context of each person becomes a complex labelnode or DR$\mathcal{L}$H subdatabase. We will consider the (DR$\mathcal{L}$H consisting of) two overlapping complexes in Fig. 10, the first—named **JohnBeliefs**—representing the beliefs of **john**, the second—named **MaryBeliefs**—those of **mary**:

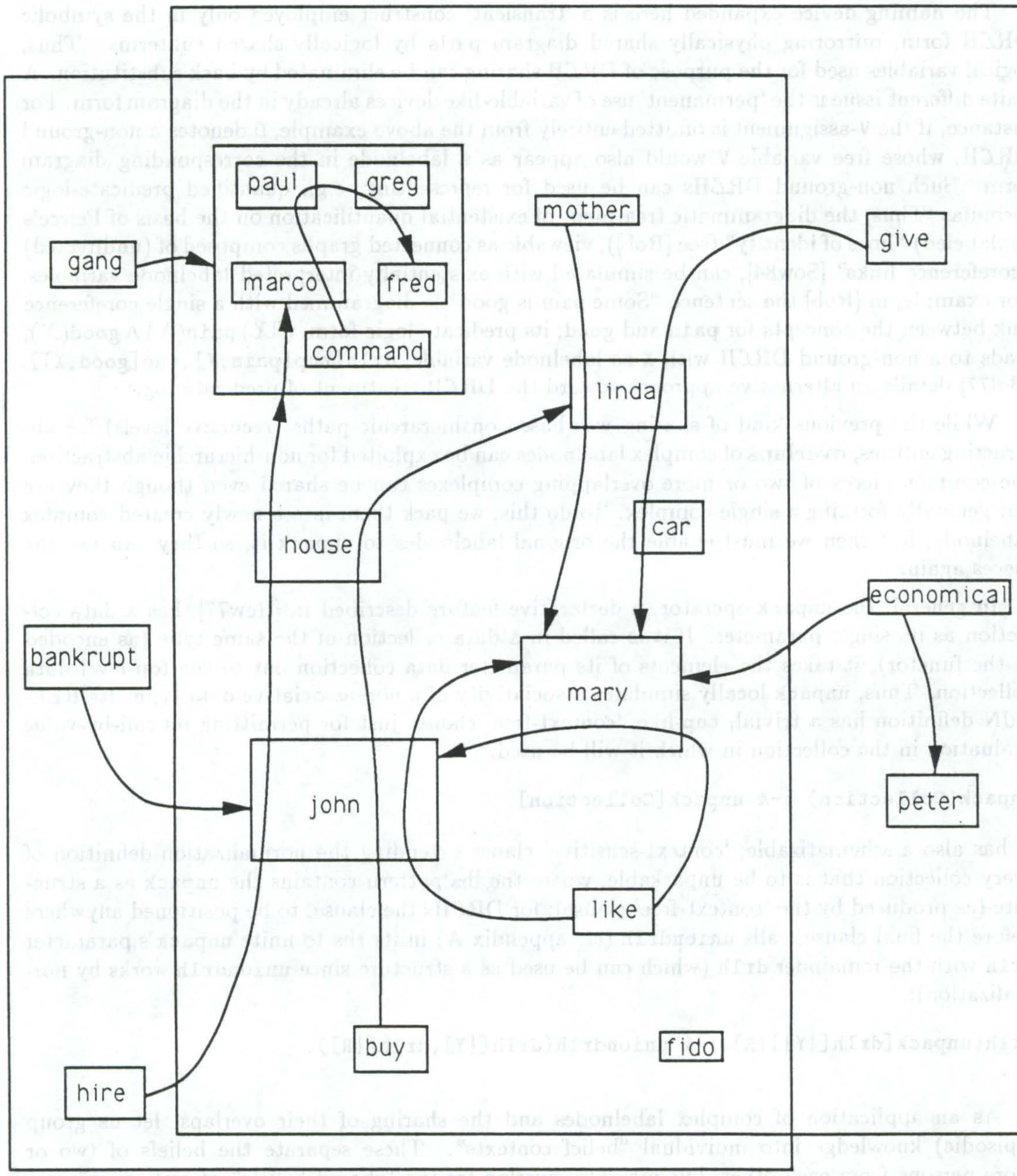

Figure 10: Two overlapping DR$\mathcal{L}$Hs or, a DR$\mathcal{L}$H with two overlapping isolated complexes

```
JohnBeliefs is
  drlh[tup[bankrupt,john],
       tup[buy,john,house,linda],
       tup[gang,drlh[tup[command,marco,paul,greg,fred]]],
       tup[hire,john,house,cntct[marco,
                                 drlh[tup[command,marco,paul,greg,fred]]]],
       tup[like,john,mary],
       tup[like,mary,john],
       tup[mother,linda,mary],
       car,
       fido],
MaryBeliefs is
  drlh[tup[buy,john,house,linda],
       tup[economical,mary],
       tup[economical,peter],
       tup[give,linda,car,mary],
       tup[like,john,mary],
       tup[like,mary,john],
       tup[mother,linda,mary],
       drlh[tup[command,marco,paul,greg,fred]],
       fido].
```

In a more abstract and maintainable version, the DR$\mathcal{L}$H of shared beliefs of `john` and `mary` is bound to a logical variable `JohnMaryShared`; this can then be united with their private belief DR$\mathcal{L}$Hs using two `unpack` calls.

```
JohnMaryShared is
    drlh[tup[buy,john,house,linda],
         tup[like,john,mary],
         tup[like,mary,john],
         tup[mother,linda,mary],
         drlh[tup[command,marco,paul,greg,fred]],
         car,
         fido],
JohnBeliefs is
  drlh(tup[bankrupt,john],
       tup[gang,drlh[tup[command,marco,paul,greg,fred]]],
       tup[hire,john,house,cntct[marco,
                                 drlh[tup[command,marco,paul,greg,fred]]]],
       unpack(JohnMaryShared)),
MaryBeliefs is
  drlh(tup[economical,mary],
       tup[economical,peter],
       tup[give,linda,car,mary],
       unpack(JohnMaryShared)).
```

If such a `drlh` call (with parentheses) containing an `unpack` call is rewritten to a `drlh` structure [with square brackets], the `unpack` is not immediately expanded but 'frozen' until the `drlh` structure becomes activated by an explicit `metacall`.

Both kinds of sharing can be combined, e.g. the above overlap-sharing example can be further abstracted by hierarchical sharing: all (contacted and uncontacted) occurrences of the complex, labelnode `drlh[tup[command,marco,paul,greg,fred]]` can be replaced by a variable MPGF to be bound to this complex using another `is` call.

5 Structure-Reducing Operations

DR$\mathcal{L}$Hs may have a rich structure consisting of both (finite but) arbitrary-length hyperarcs and arbitrary-depth labelnode nestings. For analytical purposes it is often necessary to reduce part or all of this structure, retaining only (complex) labelnodes or hyperarcs, perhaps only in a certain labelnode-nesting level. At the extreme, such reduction operations end up with the atomic labelnodes of the carrier set from which a DR$\mathcal{L}$H was built; this carrier itself constitutes a (degenerated) DR$\mathcal{L}$H. Here, we will focus on the erasure of a DR$\mathcal{L}$H's hyperarcs from the top-level (`boxes`) and from the complex labelnodes of all levels (`boxesrec`), and on the additional dissolution of these complex labelnodes (`atomicboxes`).

For exemplifying such operations, the larger DR$\mathcal{L}$H in Fig. 11 will be used, which can be regarded as a simplified representation of a city's public transportation system. Its three top-level complex labelnodes represent major transportation zones (`A`, `B`, and `C`), which are themselves interconnected by far-distance transportation lines, represented by the top-level hyperarcs (with contact labelnodes representing, e.g., main stations). Within the zones, there is a similar structure for shorter-and-shorter-distance transportation. Finally, the atomic labelnodes represent stations (or bus stops etc.). Since hyperarcs need represent nothing but transportation lines here, this use of DR$\mathcal{L}$Hs also exemplifies their reinterpretation as directed recursive **unlabeled** hypergraphs: the first element l_1 of $tup[l_1, l_2, \ldots, l_m]$ is not distinguished as the label of a hyperarc with $m-1$ nodes, but is just the first node of an unlabeled hyperarc of length m. Since most complex labelnodes are used more than once, the symbolic form of Fig. 11 employs `is` calls for hierarchical sharing:

```
A is drlh[tup[a1,a2,a3,a5,a4],tup[a4,a2,a3],tup[a7,a4],tup[a8,a7,a6]],
B is drlh[tup[b1,b2,b6],tup[b4,b2,b1,b3,b5,b6],tup[b6,cntct[b72,B7]]],
B7 is drlh[tup[b72,b71],tup[b73,b71,b72,b73]],
C is drlh[tup[c2,c1,c4,c7],
          tup[c3,c2],
          tup[cntct[c61,C6],c4,c2,c3],
          tup[c7,cntct[c65,C6]],
          c5],
C6 is drlh[tup[c61,c63,c62],tup[c62,c61],tup[c65,c63],C64],
C64 is drlh[tup[c641,c642],tup[c642,c641]] &
drlh[tup[cntct[a3,A],cntct[b1,B],cntct[c3,C]],
     tup[cntct[b6,B],d,cntct[c7,C]],
     tup[cntct[c3,C],cntct[a7,A]]].
```

The `boxes` operation reduces a DR$\mathcal{L}$H by deleting its top-level hyperarcs and keeping its labelnodes. The first clause handles a contacted DR$\mathcal{L}$H by recursion into its uncontacted version, reusing the contact labelnode for the result. The second clause returns the empty DR$\mathcal{L}$H unchanged. The third clause erases a leading hyperarc using `apptupdrlh`, which merges all labelnodes of the hyperarc into the recursion result that `boxes` obtains for the remainder DR$\mathcal{L}$H[12]. The fourth clause merges a leading labelnode into such a result.

```
boxes(cntct[B,drlh[|R]]) :-& cntct(B,boxes(drlh[|R])).
boxes(drlh[]) !-& drlh[].
boxes(drlh[tup[|Y]|R]) !-& apptupdrlh(tup[|Y],boxes(drlh[|R])).
boxes(drlh[B|R]) :-& mergebox(B,boxes(drlh[|R])).
```

For instance, the `boxes` of the transportation DR$\mathcal{L}$H, depicted in Fig. 12, consist of the contacted zone labelnodes `A`, `B`, and `C`, and the atomic station labelnode `d`, without the far-distance connections:

```
drlh[cntct[a3,A],cntct[a7,A],cntct[b1,B],cntct[b6,B],cntct[c3,C],cntct[c7,C],d]
```

[12] By applying `uniondrlh` of appendix A to the DR$\mathcal{L}$H-'converted' hyperarc, `apptupdrlh` could be defined indirectly but compactly: `apptupdrlh(tup[|Y],drlh[|R]) :-& uniondrlh(drlh(|Y),drlh[|R])`.

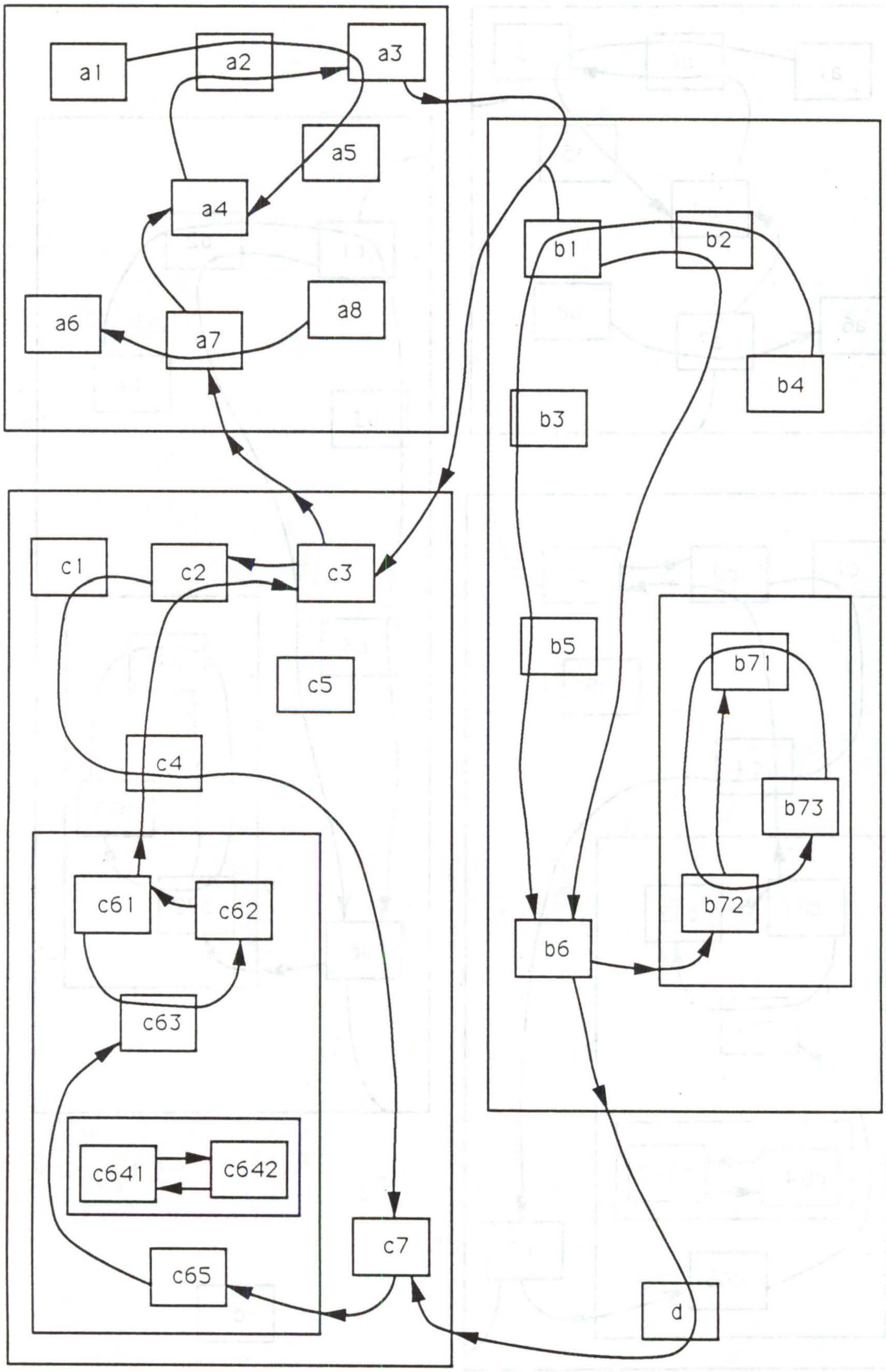

Figure 11: A DR$\mathcal{L}$H interpreted as an unlabeled transportation net

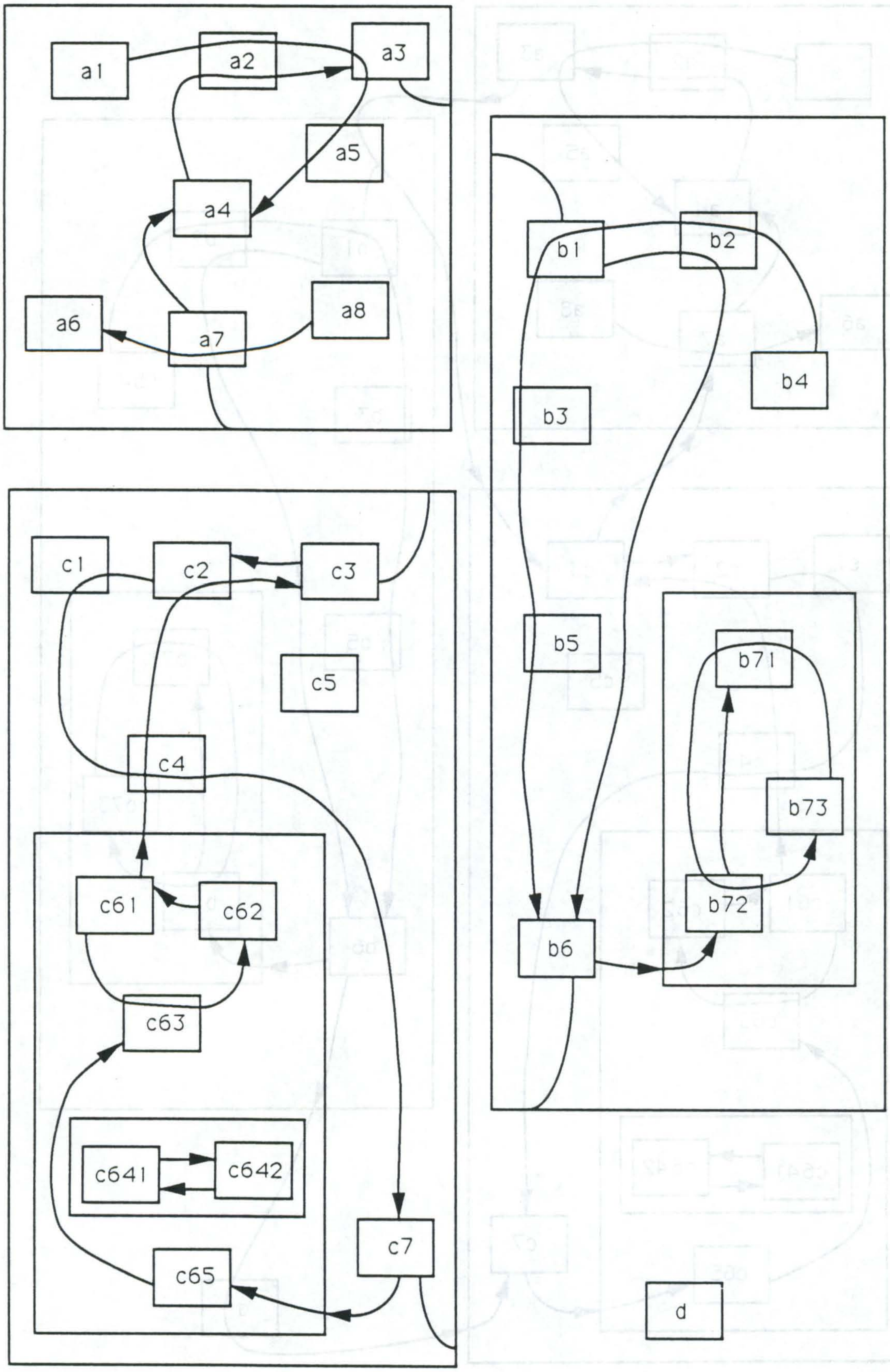

Figure 12: The **boxes** DR$\mathcal{L}$H of the transportation DR$\mathcal{L}$H

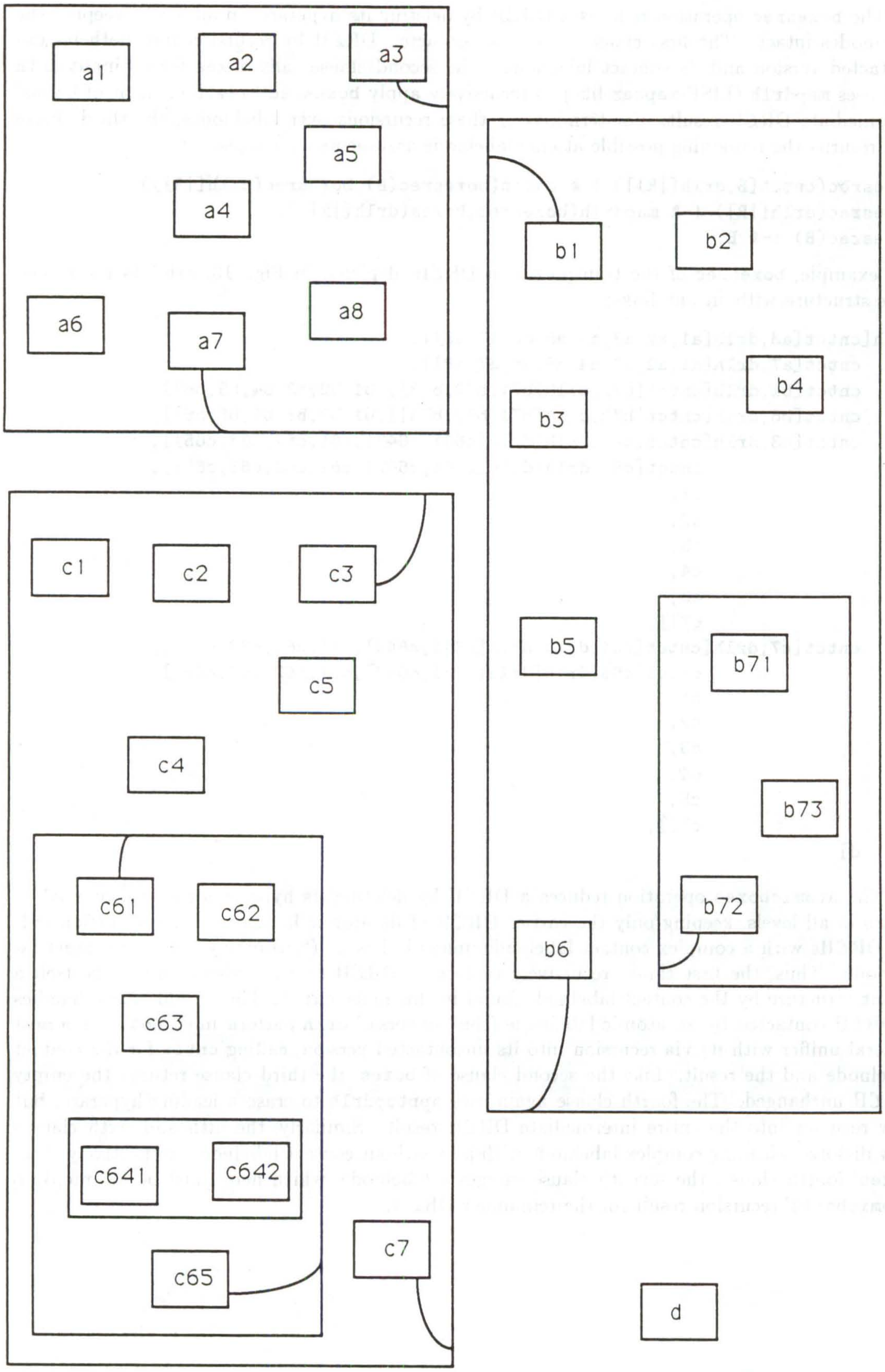

Figure 13: The **boxesrec** DR$\mathcal{L}$H of the transportation DR$\mathcal{L}$H

The **boxesrec** operation reduces a DR$\mathcal{L}$H by deleting its hyperarcs in all levels, keeping the labelnodes intact. The first clause handles a contacted DR$\mathcal{L}$H by recursion into both its uncontacted version and its contact labelnode. The second clause calls **boxes** for an input **drlh** and uses **mapdrlh** (LISP-**mapcar**-like) to recursively apply **boxesrec** to each element of **boxes**' intermediate DR$\mathcal{L}$H result. For terminating these recursions over labelnodes, the third clause just returns the remaining possible atomic-labelnode arguments unchanged.

```
boxesrec(cntct[B,drlh[|R]]) !-& cntct(boxesrec(B),boxesrec(drlh[|R])).
boxesrec(drlh[|R]) !-& mapdrlh(boxesrec,boxes(drlh[|R])).
boxesrec(B) :-& B.
```

For example, **boxesrec** of the transportation DR$\mathcal{L}$H, depicted in Fig. 13, exhibits the nested zone structure without any links:

```
drlh[cntct[a3,drlh[a1,a2,a3,a4,a5,a6,a7,a8]],
    cntct[a7,drlh[a1,a2,a3,a4,a5,a6,a7,a8]],
    cntct[b1,drlh[cntct[b72,drlh[b71,b72,b73]],b1,b2,b3,b4,b5,b6]],
    cntct[b6,drlh[cntct[b72,drlh[b71,b72,b73]],b1,b2,b3,b4,b5,b6]],
    cntct[c3,drlh[cntct[c61,drlh[drlh[c641,c642],c61,c62,c63,c65]],
                  cntct[c65,drlh[drlh[c641,c642],c61,c62,c63,c65]],
                  c1,
                  c2,
                  c3,
                  c4,
                  c5,
                  c7]],
    cntct[c7,drlh[cntct[c61,drlh[drlh[c641,c642],c61,c62,c63,c65]],
                  cntct[c65,drlh[drlh[c641,c642],c61,c62,c63,c65]],
                  c1,
                  c2,
                  c3,
                  c4,
                  c5,
                  c7]],
    d]
```

The **atomicboxes** operation reduces a DR$\mathcal{L}$H by deleting its hyperarcs and complex labelnodes in all levels, keeping only the carrier DR$\mathcal{L}$H of its atomic labelnodes; the operation fails for DR$\mathcal{L}$Hs with a complex contact labelnode unless **it** has an ('ultimately') atomic contact labelnode. Thus, the first clause recursively replaces a DR$\mathcal{L}$H contact labelnode that is itself a **cntct** structure by the contact labelnode found in this inner **cntct**. The second clause handles a DR$\mathcal{L}$H contacted by an atomic labelnode (the 'universal' **drlh** pattern must **not** have a most general unifier with it) via recursion into its uncontacted version, calling **cntct** for the contact labelnode and the result. Like the second clause of **boxes**, the third clause returns the empty DR$\mathcal{L}$H unchanged. The fourth clause again uses **apptupdrlh** to erase a leading hyperarc, but now recurses into the entire intermediate DR$\mathcal{L}$H result. Similarly, the fifth and sixth clauses now dissolve a leading complex labelnode with and without contact labelnode, respectively. Like **boxes**' fourth clause, the seventh clause merges a labelnode, which here must be atomic, into **atomicboxes**' recursion result for the remainder DR$\mathcal{L}$H.

```
atomicboxes(cntct[cntct[B,drlh[|_]],drlh[|R]]) !-&
                                    atomicboxes(cntct[B,drlh[|R]]).
atomicboxes(cntct[B,drlh[|R]]) :- not(mgu(drlh[|_],B)) &
                                    cntct(B,atomicboxes(drlh[|R])).
atomicboxes(drlh[]) !-& drlh[].
atomicboxes(drlh[tup[|Y]|R]) !-& atomicboxes(apptupdrlh(tup[|Y],drlh[|R])).
atomicboxes(drlh[cntct[B,drlh[|Y]]|R]) !-& atomicboxes(apptupdrlh(tup[|Y],
                                                           drlh[|R])).
atomicboxes(drlh[drlh[|Y]|R]) !-& atomicboxes(apptupdrlh(tup[|Y],drlh[|R])).
atomicboxes(drlh[B|R]) :-& mergebox(B,atomicboxes(drlh[|R])).
```

In the transportation example, `atomicboxes` returns the stations, without any structure left:

```
drlh[a1,a2,a3,a4,a5,a6,a7,a8,b1,b2,b3,b4,b5,b6,b71,b72,b73,c1,c2,c3,c4,c5,
                                  c61,c62,c63,c641,c642,c65,c7,d]
```

Two operations 'dual' to `boxes` and `boxesrec` perform the dissolution of a DR$\mathcal{L}$H's top-level complex labelnodes (`arrows`) and of all complex labelnodes (`arrowsrec`), altering incident hyperarcs such that a contacted complex labelnode is replaced by its contact labelnode, whereas an uncontacted one generates a failure (unlike in the earlier definitions [Bol80]). For the transportation system, having only contacted hyperarc members, these operations would show the underlying connection structure, with the (top-level) zones omitted.

The operation `atomicboxes` could then also be defined simply as the function composition `compose[arrowsrec,boxesrec]`, applying `arrowsrec` to the result of `boxesrec`. For set-degenerated normalized DR$\mathcal{L}$Hs (e.g. `drlh[drlh[drlh[a],a,b],drlh[],a,c]`) `boxesrec` acts like the identity, while `arrowsrec` hence `atomicboxes` (here returning `drlh[a,b,c]`) correspond to LISP's `flatten` for lists.

6 Searching Paths via Hyperarc Transits and Level Shifts

Path-searching is a classical non-trivial operation in semantic networks. Using DR$\mathcal{L}$Hs instead of DLGs as the graph-theoretical basis, two generalizations of legal steps in a (directed) path suggest themselves:

- **Hyperarc transits:** Starting from its first node n_1, a DLG arc $\mathtt{tup}[l, n_1, n_2]$ can step to the node n_2. Starting from any of its labelnodes a_i with[13] $i < m$, a directed labeled hyperarc $\mathtt{tup}[a_1, \ldots, a_i, a_{i+1}, \ldots, a_m]$ can step to each of the labelnodes a_j with $i + 1 \leq j \leq m$.
- **Level shifts:** For these, there is no analogy in DLGs. Starting from its contact labelnode a, a complex labelnode $\alpha = \mathtt{cntct}[a, \mathtt{drlh}[\ldots, \mathtt{tup}[..., a, ...], \ldots]]$ can step to the inner occurrence of a, shifting the path level down to the context of the `drlh` structure; vice versa, starting from an inner labelnode b also used as its contact labelnode, a complex labelnode $\beta = \mathtt{cntct}[b, \mathtt{drlh}[\ldots, \mathtt{tup}[..., b, ...], \ldots]]$ can step to the outer contact labelnode occurrence of b, shifting the path level up to the environment of the `cntct` structure.

A DR$\mathcal{L}$H path, then, is a nesting of repetitionless labelnode sequences, written here as `tup` structures: $\mathtt{tup}[start, \ldots, \alpha, \mathtt{tup}[a, ..., a_i, a_j, ..., b], \beta, \ldots, \mathtt{tup}[\ldots, \mathtt{tup}[\ldots, goal]...]]$. It begins at a top-level *start* labelnode and ends at a *goal* labelnode in any nesting level. Adjacent labelnodes a_i, a_j inside any sequence are connected by hyperarc transits. Embedded sequences are connected with adjacent contacted complex labelnodes α or β by level shifts.

These generalizations can already be discussed for single-hyperarc DR$\mathcal{L}$Hs such as the idealized depiction of human-computer interaction in Fig. 14:

[13] Labelnodes acting as labels could be excluded from paths by adding the condition $1 < i$.

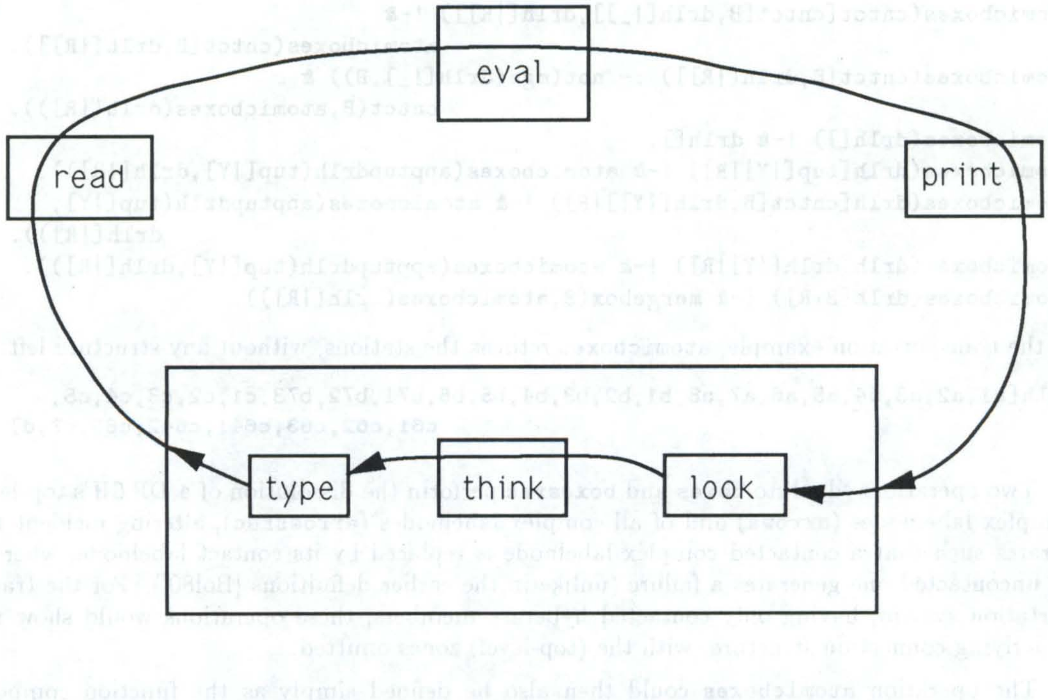

Figure 14: A DR$\mathcal{L}$H interpreted as an unlabeled `read-eval-print` loop

```
drlh[tup[cntct[type,drlh[tup[look,think,type]]],
        read,
        eval,
        print,
        cntct[look,drlh[tup[look,think,type]]]]]
```

Here, a path with one embedded sequence leads from **print** to **read**, both in the top-level:

```
tup[print,
    cntct[look,drlh[tup[look,think,type]]],
    tup[look,
        type]
    cntct[type,drlh[tup[look,think,type]]],
    read]
```

This path uses the final two labelnodes of the top-level hyperarc to step from **print** to the complex labelnode. It then shifts down into its context via the contact labelnode **look**. There, it uses the inner hyperarc to step to **type**, 'skipping' **think**. It again shifts up to the top-level environment of the complex labelnode via its contact labelnode **type**. Finally, it uses the initial two labelnodes of the top-level hyperarc to step from the complex labelnode to **read**.

Note that the directed top-level hyperarc must be used twice in this path, because we first need a later segment, then an earlier one. Of course, DLG arcs would be just "too short" for such segmentation. So, while repeated labelnodes are prohibited inside sequences of a DR$\mathcal{L}$H path, a hyperarc may participate as often as it can be divided into segments using disjoint labelnodes. A related difference between DLG and DR$\mathcal{L}$H paths arises from parallel arcs and 'transit-equivalent' hyperarcs: adjacent labelnodes in a path may be transitted by several hyperarcs that need not be parallel (anyhow impossible because of duplicate elimination in **mergearrow**) but may even cross through them via disjoint intermediate labelnodes. Thus, by specifying a DR$\mathcal{L}$H path only as labelnode sequences, we abstract from the transit-equivalent hyperarcs for each pair of adjacent labelnodes. An operation finding all hyperarc transits between a given pair of labelnodes could be used to proceed from our abstract DR$\mathcal{L}$H paths to concrete ones.

While in general DR$\mathcal{L}$Hs the tup structures representing a path are not considered as hyperarcs themselves, such a reinterpretation is applicable to hypergraphs. In this special case a DR$\mathcal{L}$H path consists only of one un-nested labelnode sequence, whose tup representation can be viewed as a single hyperarc. (After further specialization to DLGs, such an incorporation of an arbitrary path into the graph traversed becomes impossible because of its binary arcs.) For example, in the hypergraph part of the DR$\mathcal{L}$H **JohnBeliefs** in Fig. 10 there is a path from **bankrupt** to **linda**, whose **tup** representation **tup[bankrupt,john,linda]** can be reinterpreted as a hyperarc. Similarly, in **MaryBeliefs** the path **tup[linda,car]** is also viewable as a hyperarc. For the DR$\mathcal{L}$H union of **JohnBeliefs** and **MaryBeliefs** (cf. appendix A) these hyperarcs provide a shortened, **john**-less path from **bankrupt** to **car**, namely **tup[bankrupt,linda,car]**.

The main path-searching function **trav** takes a (normalized) DR$\mathcal{L}$H argument, **Net**, in which to search from the **Start** to the **Goal** argument. (Since **Start** may itself be a contacted DR$\mathcal{L}$H in whose level can be shifted immediately, **Net** may well be the empty DR$\mathcal{L}$H.) This user interface just calls the workhorse function **traverse** with the first argument **tup**-embedded and the second argument doubly **tup**-embedded: the main **tup**s represent (length-one-initialized) stacks of DR$\mathcal{L}$Hs (**Netstack**) and paths (**Pathstack**), respectively, the inner **tup**, a length-one path.

```
trav(Net,Start,Goal) :-& traverse(tup[Net],tup[tup[Start]],Goal).
```

During the search **traverse** grows the top path from right to left, with the front element always being the new **Start** labelnode from which to continue. On level-shifting down into a contacted DR$\mathcal{L}$H **cntct**[a,**drlh**[...]], its context **drlh**[...] is pushed onto **Netstack** and the length-one path of its contact labelnode **tup**[a] is pushed onto **Pathstack**. Similarly, level-shifting up from a contacted DR$\mathcal{L}$H is realized by parallel pop operations on **Netstack** and **Pathstack**. The full implementation of **traverse**, including hyperarc transits, can be found in appendix C.

As a larger example, let us consider a path through the transportation system (Fig. 11) from the top-level station **d** to the station **b73** in subzone **B7** of zone **B**:

```
tup[d,
    cntct[c7,C],
    tup[c7,
        cntct[c65,C6],
        tup[c65,c63,c62,c61],
        cntct[c61,C6],
        c3],
    cntct[c3,C],
    cntct[a7,A],
    tup[a7,a4,a3],
    cntct[a3,A],
    cntct[b1,B],
    tup[b1,
        b6,
        cntct[b72,B7],
        tup[b72,b73]]]
```

That **b73** lies two levels below the top-level can be seen at the path's ending with a nesting of three **tup** sequences[14].

[14] This shortest path is not the first one found by the **trav** function: it is not generally optimal for the **membtupall** call in the second **traverse** clause (appendix C) to choose shorter pieces of a given hyperarc before longer ones; also, the **tup** order in normalized DR$\mathcal{L}$Hs cannot be optimized for arbitrary searches. An instructive detour in **trav**'s first solution is the final subsequence **tup[b72,b71,b73]** found in **B7**. Since in this embedded DR$\mathcal{L}$H the small hyperarc **tup[b72,b71]** is lexicographically sorted before the circular hyperarc **tup[b73,b71,b72,b73]**, it is transitted first by **findarrow**; starting from **b71** in the next **traverse** recursion, the circle provides the only transits, which—since **b72** already occurs in the path—causes a direct skip to the **Goal** labelnode. Such "virtual repetitions" of labelnodes in a path could be prevented by extending **traverse**'s **not-membtup** checks to every labelnode skipped by a transit.

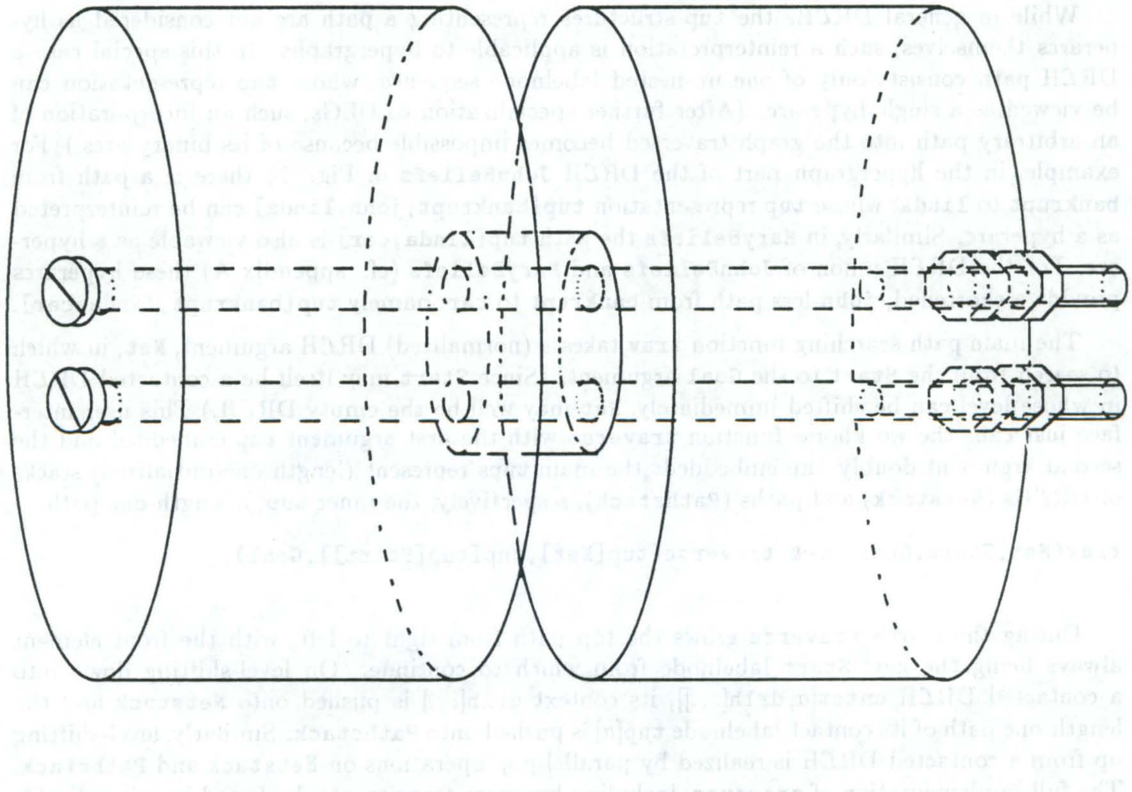

Figure 15: A 2D projection of a 'double-drum' workpiece

7 A Mechanical Engineering Application: Parts Lists

The application of DR$\mathcal{L}$Hs for representing and processing real-world knowledge will be exemplified in the domain of mechanical engineering[15]. Aspects of the meaning an engineer 'sees' in a CAD-like graphics of a workpiece (Fig. 15) can be captured by a DR$\mathcal{L}$H diagram (Fig. 16): individual subparts (drums and a disk) and connection devices (nuts and bolts) are represented as instances of abstract concepts, and their (fastening and adjacency) relationships are expressed explicitly.

Note that we use length-one hyperarcs for representing the application of unary predicates like **drum** to individuals like **dr1**; most other formalisms for semantic networks would require some auxiliary **isa**-like "∋"-link here. Also, we exploit the variable lengths of hyperarcs to obtain an 'analogical' representation in which relations like **fasten** mirror with their arguments the natural order of objects; the **adjacent** relation even has both binary and ternary occurrences, where, however, the latter can be viewed as an abbreviation for a pair of binary ones[16].

Atomic boxes or labelnodes such as **bolt** could be recursively refined to complex ones for describing objects' internal properties such as geometry, material, and function. Conversely, the entire DR$\mathcal{L}$H could be used as a single complex box in a larger workpiece representation.

[15] The Acquisition, Representation, and Compilation of such TEChnical knowledge is studied in the CIM-oriented project ARC-TEC at the German Research Center for AI (DFKI).

[16] An engineer could infer further adjacency relationships (e.g. between dr2 and nu1/nu3) with high plausibility. In AI systems such inferences would require functional knowledge about typical mechanical constructions, whose representation will not be discussed here.

Once the knowledge is diagrammed as the DR$\mathcal{L}$H in Fig. 16, its symbolic representation

```
DoubleDrum is drlh[tup[adjacent,dr2,di1,dr1],
                   tup[adjacent,nu1,nu2],
                   tup[adjacent,nu3,nu4],
                   tup[bolt,bo1],
                   tup[bolt,bo2],
                   tup[disk,di1],
                   tup[drum,dr1],
                   tup[drum,dr2],
                   tup[fasten,bo1,dr1,di1,dr2,nu1,nu2],
                   tup[fasten,bo2,dr1,di1,dr2,nu3,nu4],
                   tup[nut,nu1],
                   tup[nut,nu2],
                   tup[nut,nu3],
                   tup[nut,nu4]]
```

can be employed for performing various operations. For example, if `DoubleDrumV` is bound to a `DoubleDrum` variant with `bo1` and `bo2` exchanged throughout by `bo3` and `bo4`, respectively, the DR$\mathcal{L}$H intersection (cf. appendix A) `interdrlh(DoubleDrum,DoubleDrumV)` returns a 'loosened' version without `bolt` and `fasten` relationships: such a "maximal common subrepresentation" of two workpiece representations can be regarded as a result of their "analogy matching", useful for similarity planning. Alternatively, `boxes` or `boxesrec` (cf. section 5) show the incredient labelnodes of `DoubleDrum` (here equivalent because there are no complex boxes): this is the "domain vocabulary" to be understood when interpreting CAD graphics (a refined version is the `partslist` operation below). Finally, `trav` (cf. section 6) finds a path from `bo1` to `nu4` by changing the `fasten` hyperarc at `dr1`, `di1`, or `dr2`: for designing or diagnosing a workpiece it is important to know that and how mechanical force, thermic energy, or electric current might be transmitted between two given points.

These library operations can easily be extended by further RELFUN definitions. For instance, suppose we want to generate parts lists from workpiece nets such as `DoubleDrum`. Let each list entry simply consist of the kind of part and the number of its occurrences. This information will be represented as a binary second-order relation `card` between a concept (unary predicate) and the cardinality of its extension (number of individuals). So, in the "generalized parts list problem" a given DR$\mathcal{L}$H is to be transformed into a DLG of all arcs `tup[card,`*concept*, *n*], where *concept* acted as the label of length-one hyperarcs `tup[`*concept*, *ind*] and n is the number of labelnodes *ind* that *concept* was pointing to[17].

Our solution has the form of an operation definition `partslist`, declaratively composed of two suboperations, namely `concount` followed by `redcard`. While `concount` augments a DR$\mathcal{L}$H by isolated complex labelnodes each containing a `card` relationship, `redcard` deletes all DR$\mathcal{L}$H pieces except these `card` relationships. In `concount` we utilize the canonical ordering of DR$\mathcal{L}$H pieces, thus relying on the DR$\mathcal{L}$H being normalized.

The operation `concount` ("concept counts") distinguishes through its clauses three forms of its DR$\mathcal{L}$H argument: it may start with a length-one hyperarc, with some other hyperarc, or it may have any further form. In the first case `concount` hands the length-one hyperarc to a corecursive operation `incind` for counting the individuals of its label concept (initializing the counter with 1). In the second case the non-length-one hyperarc is constructed to the result of `concount`'s recursion into the DR$\mathcal{L}$H remainder. In the third case the DR$\mathcal{L}$H, which may start with a labelnode or may be empty, is returned unchanged.

The operation `incind` ("increment individuals") distinguishes two forms of its DR$\mathcal{L}$H argument:

[17] If the `DoubleDrum` net also included `subsumes` relationships `tup[subsumes,`*concept*, *subconcept*] between concepts and their subconcepts (e.g. `tup[subsumes,cylinder,disk]` and `tup[subsumes,cylinder,drum]`), an extended version could be defined to sum up the number of individuals pointed to by a concept, all its subconcepts, subsubconcepts, etc. In heterarchies, an individual which is, e.g., both a `disk` and a `drum` should be counted only as one `cylinder`.

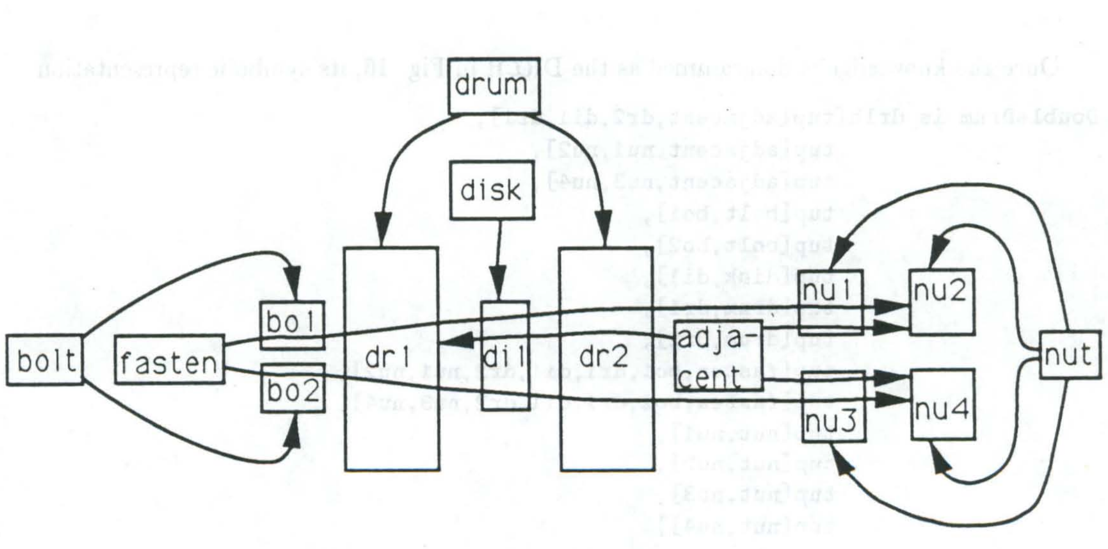

Figure 16: A DR$\mathcal{L}$H representation of the double drum

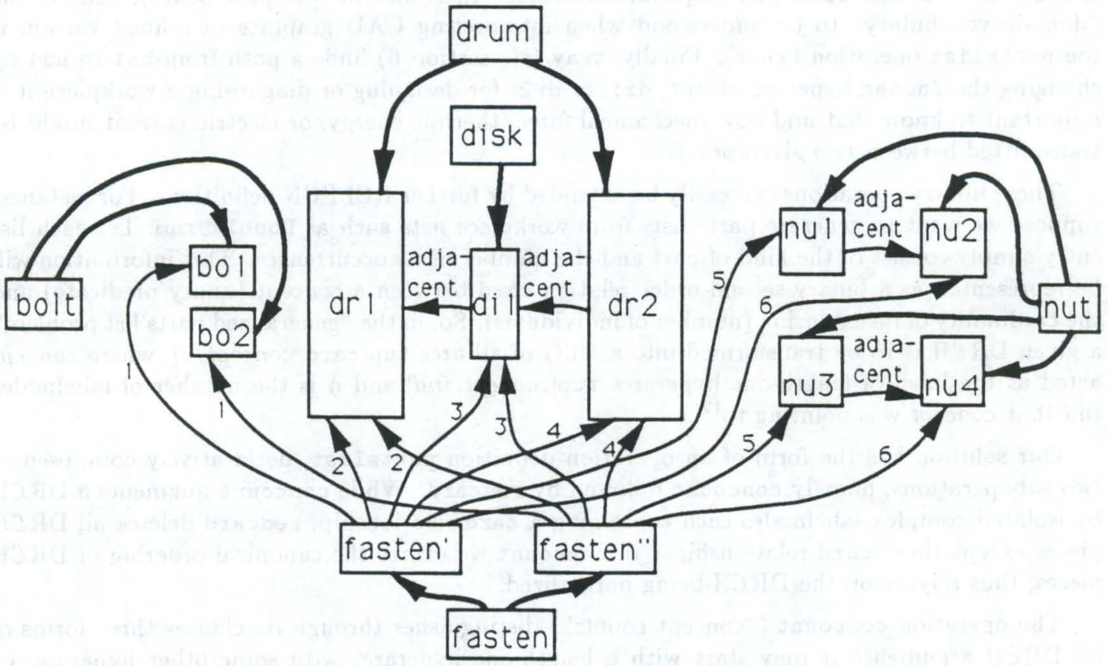

Figure 17: A DLG re-representation of the double drum

it may start with two length-one hyperarcs having the same Concept or with any length-one hyperarc. In the first case the front length-one hyperarc is constructed to the result of an incind recursion with a 1-incremented counter and the remainder DR$\mathcal{L}$H. In the second case the length-one hyperarc is constructed to the result of merging a complex box into the result of a concount corecursion with the remainder DR$\mathcal{L}$H: the complex box contains the hyperarc's label Concept and its final counter state `N` as the nodes of a new `card` arc. (The canonical ordering prevents another hyperarc occurrence having the same label `Concept`, i.e. it is a symbolic analogue to the "labelnode locality of information" in diagrams.)

```
concount(drlh[tup[Concept,Ind]|R]) !-& incind(1,drlh[tup[Concept,Ind]|R]).
concount(drlh[tup[|Y]|R]) !-& consdrlh(tup[|Y],concount(drlh[|R])).
concount(drlh[|R]) :-& drlh[|R].

incind(N,drlh[tup[Concept,Indi],tup[Concept,Indii]|R]) !-&
  consdrlh(tup[Concept,Indi],incind(1+N,drlh[tup[Concept,Indii]|R])).
incind(N,drlh[tup[Concept,Ind]|R]) :-&
  consdrlh(tup[Concept,Ind],mergebox(drlh[tup[card,Concept,N]],
                                     concount(drlh[|R]))).
```

For example, the `is` call `DoubleDrumC is concount(DoubleDrum)` is equivalent to

```
DoubleDrumC is drlh[tup[adjacent,dr2,di1,dr1],
                    .
                    tup[nut,nu4],
                    drlh[tup[card,bolt,2]],
                    drlh[tup[card,disk,1]],
                    drlh[tup[card,drum,2]],
                    drlh[tup[card,nut,4]]]
```

The operation `redcard` ("reduce to cardinalities") uses three clause patterns for its DR$\mathcal{L}$H argument: it may be the empty DR$\mathcal{L}$H, start with a complex labelnode containing a `card` arc, or start with anything else. In the first case `redcard` just returns the empty DR$\mathcal{L}$H. In the second case the complex-labelnode-extracted `card` arc is constructed to the result of `redcard`'s recursion into the DR$\mathcal{L}$H remainder. In the third case the DR$\mathcal{L}$H front is discarded and `redcard` immediately recurses into the DR$\mathcal{L}$H remainder.

```
redcard(drlh[]) !-& drlh[].
redcard(drlh[drlh[tup[card,Concept,N]]|R]) !-&
  consdrlh(tup[card,Concept,N],redcard(drlh[|R])).
redcard(drlh[_|R]) :-& redcard(drlh[|R]).
```

For example, the call `redcard(DoubleDrumC)` returns

```
drlh[tup[card,bolt,2],
     tup[card,disk,1],
     tup[card,drum,2],
     tup[card,nut,4]]
```

Now, the main operation `partslist` can be obtained simply as a `compose` of `redcard` and `concount`.

```
partslist :-& compose[redcard,concount].
```

For instance, the desired call `partslist(DoubleDrum)` is equivalent to the preceding call `redcard(DoubleDrumC)`.

Let us conclude these declarative DR$\mathcal{L}$H operations by noting that many of them have meaningful DLG specializations. It was already mentioned that the `DoubleDrum` example has no complex labelnodes and that its `adjacent` hyperarcs can be reduced to arcs. Its `fasten` hyperarcs could be simulated by introducing two "relationship nodes" `fasten'` and `fasten''` with six "role arcs" (here just ordinal numbers) pointing to the bolt, the nuts, and the parts to be connected. Similarly, the unary hyperarcs could be re-represented as "inverse-`isa` arcs" (here symbolized by heavy lines). The resulting (less succinct!) DLG in Fig. 17 is close to representations in other semantic net systems such as KL-ONE. For it, the operations `interdrlh` and

boxes/boxesrec could be used directly, producing analogous results (the latter would however also show the artificial relationship nodes). On the other hand, **trav** could not be used since role arcs would have to be traversed in both directions (this loss of a meaningful concept of **directed** paths in the standard DLG simulation of n-ary relationships is a main criticism of standard semantic networks). Finally, **partslist** could be reformulated to produce the original result (which already happened to be a DLG).

8 DR$\mathcal{L}$H Database Storage and Retrieval

In the previous sections we have treated DR$\mathcal{L}$Hs exclusively in the form of terms passed as arguments, bound to logical variables, and returned as values. For large nets, however, some more persistent DR$\mathcal{L}$H form may also be necessary, e.g. for associative storage and retrieval. As discussed in the introduction, we attempt to cleanly separate such imperative database aspects from the declarative operations.

In this section a simple representation for asserting DR$\mathcal{L}$Hs into associative RELFUN databases is given (the kind of database where operator definitions are stored). Furthermore, a standard interface between this asserted representation and the declarative external representation is sketched.

We will consider two possibilities: to assert a DR$\mathcal{L}$H as a whole, and to assert its **tup**, **cntct**, **drlh**, and atomic elements individually and regarding all these asserted DR$\mathcal{L}$H pieces in the database as implicitly constituting one DR$\mathcal{L}$H. The latter method is more general because the case of a single asserted **drlh** 'piece' corresponds to the former method.

For representing the unique normal forms of such pieces in assertions and queries, we just embed them into calls of a new unary predicate, **net**.

By virtue of call-by-value evaluation, **net** arguments that are "**(...)**"-calls of **tup**, **cntct**, or **drlh** are normalized to "**[...]**"-structures before they are seen by the main **net** call. Thus,

```
net(cntct(screen,drlh(tup(wired,keyboard,screen),wired)))
```

really means

```
net(cntct[screen,drlh[tup[wired,keyboard,screen]]])
```

In this way the user can ensure that DR$\mathcal{L}$H pieces are always normalized before database storage and retrieval.

The **net** predicate is defined by asserting **facts** only, one for the storage of each DR$\mathcal{L}$H piece. Retrieval is done by querying these facts using associative **net** patterns with named (e.g. "**Who**") or anonymous ("_") variables.

For instance, the DR$\mathcal{L}$H we called **MaryBeliefs** in section 4 (see Fig. 10) can be asserted by the following sequence of **net** facts:

```
net(tup[buy,john,house,linda]).
net(tup[economical,mary]).
net(tup[economical,peter]).
net(tup[give,linda,car,mary]).
net(tup[like,john,mary]).
net(tup[like,mary,john]).
net(tup[mother,linda,mary]).
net(drlh[tup[command,marco,paul,greg,fred]]).
net(fido).
```

Now, the query

```
net(tup[economical,Who]).
```

non-deterministically binds **Who** to **mary** or **peter**, and

```
net(drlh[tup[Label,marco|_]|_]).
```

succeeds once by binding **Label** to **command**.

If a second DR$\mathcal{L}$H, say **JohnBeliefs**, is to be asserted into the same database, "belief interference"—after loss of the original DR$\mathcal{L}$H boundaries—could be avoided by now storing both belief contexts as isolated complex labelnodes:

```
net(drlh[tup[buy,john,house,linda],
         . . .
         fido]).
net(drlh[tup[bankrupt,john],
         . . .
         fido]).
```

However, in this representation the two previous queries would involve complicated patterns. An alternative is to give the **net** predicate an extra argument, naming the 'module' in which DR$\mathcal{L}$H pieces are to be asserted, say **mb** for **MaryBeliefs** and **jb** for **JohnBeliefs** (this would also permit separate storage of shared beliefs such as those in **JohnMaryShared**, mirroring our declarative overlap sharing):

```
net(tup[buy,john,house,linda],mb).
         . . .
net(fido,mb).
net(tup[bankrupt,john],jb).
         . . .
net(fido,jb).
```

Since the module name is the second argument, an anonymous 'rest' variable would still permit a single call to retrieve not only from **any** module but also from module-less **unary net** facts. Thus, while

```
net(tup[Rel,linda|What],mb).
```

queries **mary**'s module for all relationships in which **linda** participates as the first argument,

```
net(tup[Rel,linda|What]|_).
```

queries all unary and binary **net** facts for the **linda** relationships.

Instead of letting the user make piecemeal assertions and queries, it is possible to define a standard interface to **net** facts, which takes as argument and returns as value the entire global DR$\mathcal{L}$H (we will discuss a simple version without module names). On globally asserting a DR$\mathcal{L}$H, a previously stored global DR$\mathcal{L}$H will be overwritten. Thus, the global DR$\mathcal{L}$H can be modified by retrieving it, transforming it declaratively, and storing it again.

Besides the advantage of encapsulating procedural updates to a narrow interface, this method also avoids another problem of piecemeal updates: keeping the global DR$\mathcal{L}$H in normal form. For instance, after asserting our previously normalized **cntct** structure by

```
net(cntct[screen,drlh[tup[wired,keyboard,screen]]]).
```

an attempt to assert its again normalized uncontacted **drlh** version by

```
net(drlh[tup[wired,keyboard,screen]]).
```

should add nothing to the global DR$\mathcal{L}$H because the "similpotence" property (cf. section 3 and appendix B) causes a **cntct** to swallow its **drlh**[18]. Our standard **net** interface need not deal

[18] Although an assertion operation that performs such global DR$\mathcal{L}$H normalization could be defined, it would be

with such dependencies between assertions because it stores the global DR$\mathcal{L}$H as a single self-normalizing term. Instead of the above pair of assertions we write

```
storedrlh(drlh(cntct[screen,drlh[tup[wired,keyboard,screen]]],
               drlh[tup[wired,keyboard,screen]])).
```

whose argument becomes `drlh[cntct[screen,drlh[tup[wired,keyboard,screen]]]]`, via similpotence, before it is even 'seen' by **storedrlh**.

The **storedrlh** operation is defined to **abolish** the previous **net** and then using **assertdrlh** to **assertz** each piece **X** of the given **drlh** structure as a **net** fact (as in PROLOG, **abolish** retracts all clauses of a predicate, while **assertz** adds a new last clause):

```
storedrlh(drlh[|R]) :- abolish(net), assertdrlh(drlh[|R]).
assertdrlh(drlh[])!
assertdrlh(drlh[X|R]) :- assertz(net(X)), assertdrlh(drlh[|R]).
```

The complementary, parameterless **retrievedrlh** operation calls a (PROLOGish) **bagof**, to collect all **X** for which **net(X)** holds (i.e. all DR$\mathcal{L}$H pieces) in **tup[|S]**, and returns their **drlh** normalization result **drlh(|S)**:

```
retrievedrlh() :- bagof(X,net(X),tup[|S]) & drlh(|S).
```

The composition **storedrlh(retrievedrlh())** replaces any database **net** by its globally normalized form. Also, with d being any normalized uncontacted DR$\mathcal{L}$H, the valued conjunction **storedrlh(**d**) & retrievedrlh()** replaces any database **net** by d's pieces and returns d itself.

As an example of the interplay between these standard interface operations and our declarative operations suppose that the transportation DR$\mathcal{L}$H of section 5 (see Fig. 11) was stored in the global database by a **storedrlh** call. Now, if we want to replace this global DR$\mathcal{L}$H by its labelnodes united with **drlh[tup[d,e,f],tup[f,e,d]]**, it is first retrieved by **retrievedrlh**, then transformed by the declarative **boxes** and **uniondrlh** operations, and finally stored back by **storedrlh**:

```
storedrlh(uniondrlh(boxes(retrievedrlh()),drlh[tup[d,e,f],tup[f,e,d]])).
```

It is also possible to extract a DR$\mathcal{L}$H without isolated labelnodes from a non-**net** RELFUN subdatabase of relations, exploiting a simple correspondence between DR$\mathcal{L}$H hyperarcs and RELFUN relationships (for functional clauses this would be not so easy): **tup**$[a_1, a_2, \ldots, a_m] \longleftrightarrow a_1(a_2, \ldots, a_m)$.

An operation **retrievedrlhlogic** can be defined like **retrievedrlh** but with **tup[|S]** containing a hyperarc **tup[F|R]** for any relationship **F(|R)**, where **F** is a relation variable.

```
retrievedrlhlogic() :- bagof(tup[F|R],F(|R),tup[|S]) & drlh(|S).
```

This definition can only return a finite DR$\mathcal{L}$H for a database with a finite number of (deducible) relationships, in the simplest case, a database of facts. For the well-known DATALOG database

```
likes(john,X) :- likes(X,wine).
likes(mary,wine).
```

retrievedrlhlogic() would return the **drlh** structure

```
drlh[tup[likes,john,mary],
     tup[likes,mary,wine]]
```

---

complicated by various kinds of implicit retracts. For instance, if the above net facts were asserted in reverse order, similpotence would require the contacted version to retract the uncontacted one. More frequently, on asserting a hyperarc, adsorption would enforce retracts for all its labelnodes.

The complementary operation **storedrlhlogic** could be defined to assert facts representing the (hyper)arcs of simple DR$\mathcal{L}$Hs like the above (as opposed to DR$\mathcal{L}$Hs with isolated labelnodes and complex labelnodes, which would require special treatment). Thus, the composition **storedrlhlogic(retrievedrlhlogic())** would 'extensionalize' the original DATALOG rule to the fact **likes(john,mary)**.

9 Conclusions

The goal of our DR$\mathcal{L}$H work was the development of a compact, elegant, and modular combination of three graph generalizations with interchangeable diagrammatic and symbolic notations: (1) Directed hyperarcs are introduced for the natural representation of n-ary relations. (2) Complex nodes (with optional contact nodes) are permitted for providing nested depths of description. (3) Labels are usable like nodes for obtaining higher-order capabilities. Generalizations (1)-(3) can be employed individually or in any combination, tuning the expressive power of DR$\mathcal{L}$Hs to the representation problems at hand. In our earlier DR$\mathcal{L}$H papers these generalized graphs were introduced, defined formally, implemented in FIT, applied to knowledge representation, and compared with alternative approaches. Based on the symbolic notation, the present article integrates our DR$\mathcal{L}$H work with our current RELFUN project, showing how 'logical' terms can be processed as 'analogical' graphs.

Because of (2), DR$\mathcal{L}$Hs generalize not only directed labeled graphs but also nested sets. These special cases constitute "pure structures" permitting a multitude of interpretations: set elements as well as graph nodes and arcs have been used to stand for all conceivable things, from very concrete ones to the most abstract. The study of purely structural set and graph properties—separately from their various interpretations—turned out to be rather fruitful, as it helped to discover fundamental similarities and differences between superficially incomparable systems. We have been following this same philosophy for the enriched pure structure of DR$\mathcal{L}$Hs, characterized axiomatically in definition 1. As an example consider a path through a DLG, which can be interpreted as a relational composition in a semantic net or as an activation chain in a neural net, with similar notions of (semantic or neural) distance. However, DLGs are not rich enough to refine naively drawn semantic and neural nets in order to represent their structural differences. Thus, the differentiation is often made only on the basis of their different interpretations as concepts or neurons. Using DR$\mathcal{L}$Hs, the internal structure of (assemblies of) concepts and neurons can be differentiated with complex labelnodes, and their multiple connection structures can be approximated with hyperarcs; then, the generalized path-searching and other DR$\mathcal{L}$H algorithms of this article would reveal further differences. Such finer structural tools can demonstrate why 1-to-1 mappings must be replaced by m-to-n mappings when 'implementing' concepts by neurons, quite independently from the interpretation of neuron models as biological cells (which seem to die more frequently than people forget learned concepts).

DR$\mathcal{L}$H diagrams thus are not just an alternative, graphical representation of a well-known symbolic formalism, but the formalism is itself constituting a generalized algebraic structure. Many other diagram formalisms are defined by interpreting them as the surface form of a known algebraic structure. Even semantic networks have often been formalized as a graphical version of (a subset of (first-order)) predicate logic. For instance, recent KL-ONE versions are so much viewed as subsets of predicate logic that symbolic special-purpose notations have almost supplanted the original KL-ONE diagrams. Another example is Higraphs [Har88], whose nodes always represent sets, with complex nodes representing the union of their embedded nodes, and a node-partitioning line representing the unordered Cartesian product of the partitions. In our opinion a diagram formalism should not provide overly special 'built-in' interpretations as idiosyncrasies one has to live with, but should be used—like sets or graphs—as a general basis on top of which more specialized constructs may be optionally defined. This article emphasizes dynamic versions of such constructs on the basis of DR$\mathcal{L}$Hs and RELFUN, e.g. the binary function **uniondrlh** for uniting complex labelnodes (see appendix A). However, it is also possible to introduce their static versions on the sole basis of DR$\mathcal{L}$Hs, e.g. a **union**-label for length-two hyperarcs leading from a complex labelnode to a new atomic labelnode that represents the union of its embedded labelnodes. For instance,

```
drlh[tup[larger,u1,drlh[drlh[a,b,c],drlh[d,e]]],
     tup[union,drlh[drlh[a,b,c],drlh[d,e]],u1]]
```

expresses the fact that **u1**, the five-element **union** of the two sets in $\{\{a,b,c\},\{d,e\}\}$, is **larger** than this set itself.

Our original motivation for directed hyperarcs came from Berge's definition of hypergraphs, now updated in [Ber89]. He introduced undirected hyperarcs (*edges*) as subsets of a set of nodes (*vertices*), drawing them like Euler-Venn diagrams for cardinalities greater two. Seeking diagrams for relational structures, we introduced our directed version of hyperarcs, which can cross common nodes without ambiguity. While Berge's edges are sets (unordered, without repetitions), our directed hyperarcs are tuples (ordered, with repetitions). Of course, it is possible to introduce other structures within edges, but we feel that n-tuples provide the most simple and natural concept of directedness: it is the obvious generalization of ordered pairs, i.e. directed binary arcs. If the total node ordering of our directed hyperarcs should not be desired for an application requiring a partial order, we can use complex nodes as unordered sets within ordered hyperarc tuples. For example, binary-operator precedences for simple arithmetic (in)equations can be specified by the single hyperarc **tup[prec,'^',drlh['*','/'],drlh['+','-'],drlh['=','>','<']]**. The special case of a directed arc between two complex nodes could be used to connect source and target places in Petri nets (without reifying transitions as nodes). An ('in-ordered', 'out-ordered') variant with a source and a target tuple instead of sets was called "polyedge" in [Lan69], which can be represented by introducing a hyperarc **within** both complex nodes. The further specialization to a directed arc with a set/tuple in the source only has been used to visualize signatures of many-sorted algebras [GTW78]. Later, ('in-unordered', 'out-unordered') polyedges were also called "directed hyperedges" [Har88].

The fact that most of Berge's undirected hyperarcs do not look like arcs but like set diagrams has occasionally lead to their confusion with complex nodes. However, complex nodes, unlike undirected hyperarcs, can be nested recursively and can be connected by directed hyperarcs. The most influential work with respect to recursive graphs was Pratt's definition of hierarchical graphs [Pra69]. He marked nodes with names of entire graphs, thus introducing the hierarchy (but also permitting circularities, as indicated in section 4). Wishing to avoid the necessity of names for our recursive graph generalization, we permitted the direct embedding of graphs into graphs, in both figures and formulas. The present article shows that the resulting recursive data structure, like LISP lists, permits most processing being specified as declarative operations.

Based on an algebraic view of DR$\mathcal{L}$H normalization [Bol84] and our RELFUN programming system [Bol90], the article extends work on DR$\mathcal{L}$H operations formulated as FIT programs [Bol80]. FIT permits more powerful 'parallel' patterns, which are realized, however, by breadth-first search. In RELFUN, a DR$\mathcal{L}$H pattern may contain at most one variable that matches arbitrary-size 'rests', hence matching is deterministic (we do not perform general DR$\mathcal{L}$H unification). RELFUN "rule conflicts" are handled by depth-first search, where backtracking can often be cut off immediately after the successful match of a DR$\mathcal{L}$H rule pattern. Thus, RELFUN can more easily exploit PROLOG's compilation technology for DR$\mathcal{L}$H processing (on sequential computers) than FIT. Although the RELFUN interpreter kernel is itself implemented declaratively by pure LISP functions, its efficiency was sufficient for developing the declarative DR$\mathcal{L}$H operations and processing the sample DR$\mathcal{L}$Hs of this article. However, to improve performance for larger knowledge bases, we are developing a PROLOG-WAM-like compiler for RELFUN; it currently handles the first-order RELFUN subset, allowing a 'rest' variable in hyperarc (list) patterns, which can also be used to represent such variables for **drlh** terms (arbitrary structures).

While the generality of DR$\mathcal{L}$Hs has proved to be a continuing challenge for the efficient implementation of our AI languages, as pure structures these generalized graphs do not seem to pose new complexity-theoretical problems not already arising in DLGs. Reductions of DR$\mathcal{L}$Hs to representationally equivalent DLGs produce 'larger' data structures composed of 'smaller' parts (compare Fig. 16 with Fig. 17). The increased size of the elementary DR$\mathcal{L}$H pieces thus appears to be fully compensated by the decreased total data size. For non-trivial processes such as path searching the richer domain-structuring abilities of DR$\mathcal{L}$Hs may even suggest more efficient solutions than DLG representations, e.g. by localizing the search to the relevant complex labelnodes (a potential of graph "contexts" already stressed in [PF71]), and by keeping it on mainline hyperarcs as long as possible.

The interactive construction and exploration of large DR$\mathcal{L}$H-structured knowledge 'spaces', supported by modern graphics tools, remains a task for future work. Our dual view of DR$\mathcal{L}$Hs as diagrams and terms calls for a pair of (cursor) synchronized windows, with input to (and navigation through) **each** updating **both** user presentations[19]. Automatic translation between DR$\mathcal{L}$H diagrams and terms is obviously easiest for diagrams ("with extreme labelnode copies" [Bol77]) that copy a labelnode for all its hyperarc uses, as terms do; it appears to be hardest for DR$\mathcal{L}$H diagrams "without labelnode copies".

This work should be accompanied by the development of specialized vocabularies and languages enhancing the basic DR$\mathcal{L}$H/RELFUN formalism. Our experience with the many operations defined in this article suggests that RELFUN's patterns and rules are the proper medium for specifying such extensions. A COMMON LISP implementation of RELFUN, with a LISP-like syntax, and the declarative DR$\mathcal{L}$H package are available as freeware for experimental use.

References

[Acz88] Peter Aczel. *Non-Well-Founded Sets.* Number 14 in CSLI Lecture Notes. CSLI, 1988.

[Baa90] Franz Baader. Terminological cycles in KL-ONE-based knowledge representation languages. In *Proceedings Eigth National Conference on Artificial Intelligence*, pages 621–626. AAAI, AAAI-Press / The MIT Press, 1990.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *CACM*, 21(8):613–641, August 1978.

[Ber89] Claude Berge. *Hypergraphs - Combinatorics of Finite Sets*, volume 45 of *North-Holland Mathematical Library*. North-Holland, 1989.

[Bol77] Harold Boley. Directed recursive labelnode hypergraphs: A new representation-language. *Artificial Intelligence*, 9(1):49–85, 1977.

[Bol80] Harold Boley. Processing directed recursive labelnode hypergraphs with FIT programs. Technical Report IFI-HH-M-81/80, University of Hamburg, Department of Computer Science, September 1980.

[Bol84] Harold Boley. A treatment of collection data as constructor algebras. Technical Report MEMO SEKI-84-06, University of Kaiserslautern, Department of Computer Science, October 1984.

[Bol90] Harold Boley. A relational/functional language and its compilation into the WAM. Technical Report SEKI SR-90-05, University of Kaiserslautern, Department of Computer Science, April 1990.

[Col83] A. Colmerauer. Prolog in 10 figures. In *Proc. 8th IJCAI-83, Karlsruhe*, pages 487–499, August 1983.

[EHK] Hartmut Ehrig, Annegret Habel, and H. J. Kreowski. this volume.

[19] For instance, when typing in ASCII terms the interactive tool should also extend or even restructure the diagram, like an online previewer for TEX-like text formatters. Conversely, on graphical input it should 'coadjust' the symbolic form, in analogy to what we expect from a WYSIWYG surface for (a subset of) LATEX. Synchronized graphics-and-ASCII interfaces would be useful for many other purposes such as CAD databases.

[GTW78] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology. Vol. IV*, pages 81–149. Prentice-Hall, 1978.

[Har88] David Harel. On visual formalisms. *CACM*, 31(5):514–530, May 1988.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[Lan69] Peter J. Landin. A program machine symmetric automata theory. *Machine Intelligence*, 5:99–120, 1969.

[Neb89] Bernhard Nebel. On terminological cycles. In *Preprints Workshop on Formal Aspects of Semantic Networks*, 1989. Two Harbors , CA.

[PF71] Terrence W. Pratt and Daniel P. Friedman. A language extension for graph processing and its formal semantics. *CACM*, 14(7):460–467, July 1971.

[Pra69] Terrence W. Pratt. A hierarchical graph model of the semantics of programs. In *Spring Joint Computer Conference*, pages 813–825. AFIPS Conference Proceedings, Vol. 34, 1969.

[Rob] Don D. Roberts. this volume.

[Sch89] James G. Schmolze. Terminological knowledge representation systems supporting n-ary terms. In Brachman, Levesque, and Reiter, editors, *Proc. First Int. Conference on Principles of Knowledge Representation and Reasoning, Toronto*, pages 432–443, 1989.

[Sow84] John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley Publishing Company, 1984.

A Generalizations of Standard Set Operations

Since we have derived DR$\mathcal{L}$Hs from list sets, a natural question is how to generalize the usual set operations to them (in the normalized term representation).

Particularly important is **DR$\mathcal{L}$H union**, which is employed as a function `uniondrlh` in the definition of `unpack` (cf. section 4). As in sets, the binary union operation is intimately connected with the definition of DR$\mathcal{L}$Hs. Since our definition uses the variable-arity `drlh` operator to directly construct DR$\mathcal{L}$Hs of arbitrary cardinality (cf. section 3), `uniondrlh` can be defined in terms of `drlh`: in the fourth clause, the elements `R` and `S` of the two input DR$\mathcal{L}$Hs are appended as tuples, and the elements `T` of the concatenated tuple are simply given to a `drlh` call for normalization. The first three clauses just deal with the union of contacted DR$\mathcal{L}$Hs: since only one contact labelnode is permitted in a `cntct` term, two contacted DR$\mathcal{L}$Hs can only be united if they have identical (actually, unifying) contact labelnodes `B` (first clause); if only one of the DR$\mathcal{L}$Hs is contacted, this `C` or `B` is used as the contact of the union DR$\mathcal{L}$H produced by the recursive `uniondrlh` call (second and third clause).

```
uniondrlh(cntct[B,drlh[|R]],cntct[B,drlh[|S]]) :-&
  cntct(B,uniondrlh(drlh[|R],drlh[|S])).
uniondrlh(drlh[|R],cntct[C,drlh[|S]]) :-&
  cntct(C,uniondrlh(drlh[|R],drlh[|S])).
uniondrlh(cntct[B,drlh[|R]],drlh[|S]) :-&
  cntct(B,uniondrlh(drlh[|R],drlh[|S])).
uniondrlh(drlh[|R],drlh[|S]) :-
  tup[|T] is apptup(tup[|R],tup[|S]) &
  drlh(|T).
```

Continuing the overlap example in Fig. 10 (see section 4), we can form its "belief union" by the call **uniondrlh(JohnBeliefs,MaryBeliefs)**:

```
drlh[tup[bankrupt,john],
     tup[buy,john,house,linda],
     tup[economical,mary],
     tup[economical,peter],
     tup[gang,drlh[tup[command,marco,paul,greg,fred]]],
     tup[give,linda,car,mary],
     tup[hire,john,house,cntct[marco,
                               drlh[tup[command,marco,paul,greg,fred]]]],
     tup[like,john,mary],
     tup[like,mary,john],
     tup[mother,linda,mary],
     fido]
```

The basic idea of **DR$\mathcal{L}$H intersection** is to keep not only identical elements (hyperarcs and isolated labelnodes) occurring set-like in both input DR$\mathcal{L}$Hs but also labelnodes 'producible' from a hyperarc or a contacted complex in the input DR$\mathcal{L}$Hs. For hyperarcs, 'producible' means inverse application of adsorption, $\mathtt{tup}[a_1,\ldots,a_i,\ldots,a_m] \longrightarrow a_i$, for contacted complex labelnodes, inverse application of "similpotence" (cf. section 3 and appendix B), $\mathtt{cntct}[b,\mathtt{drlh}[x_1,\ldots,x_m]] \longrightarrow \mathtt{drlh}[x_1,\ldots,x_m]$. In the **interdrlh** definition, the first three clauses again handle the obvious cases of contacted input DR$\mathcal{L}$Hs. The fourth clause returns the empty DR$\mathcal{L}$H if the first argument is empty. The fifth clause expects the first argument to begin with a hyperarc **tup[|Y]**, testing whether it is a member of the elements **S** of the second argument, viewed as a tuple: if yes, it is merged into the recursion result of **interdrlh** with a shortened first argument; otherwise, inverse adsorption is performed by merging the elements of **tup[|Y]** into the first argument, using the auxiliary **apptupdrlh** (cf. section 5), and calling **interdrlh** with the enlarged first argument. The sixth clause must deal with any labelnode **B** in the front of the first argument, checking whether it is a member of the second argument, in the sense of a predicate **membdrlh** discussed later: if yes, **B** is simply merged into the **interdrlh** result using a **B**-less first DR$\mathcal{L}$H; if no, but if **B** has the form of a contacted complex **cntct[_,drlh[|T]]**, inverse similpotence is applied by replacing **B** with its uncontacted version **drlh[|T]** in the next **interdrlh** call; otherwise, the **interdrlh** recursion omits **B** entirely.

```
interdrlh(cntct[B,drlh[|R]],cntct[C,drlh[|S]]) :-&
  if mgu(B,C)
  then cntct(B,interdrlh(drlh[|R],drlh[|S]))
  else interdrlh(drlh[|R],drlh[|S]).
interdrlh(drlh[|R],cntct[C,drlh[|S]]) :-&
  interdrlh(drlh[|R],drlh[|S]).
interdrlh(cntct[B,drlh[|R]],drlh[|S]) :-&
  interdrlh(drlh[|R],drlh[|S]).
interdrlh(drlh[],drlh[|S]) :-& drlh[].
interdrlh(drlh[tup[|Y]|R],drlh[|S]) !-&
  if membtup(tup[|Y],tup[|S])
  then mergearrow(tup[|Y],interdrlh(drlh[|R],drlh[|S]))
  else interdrlh(apptupdrlh(tup[|Y],drlh[|R]),drlh[|S]).
interdrlh(drlh[B|R],drlh[|S]) :-&
  if membdrlh(B,drlh[|S])
  then mergebox(B,interdrlh(drlh[|R],drlh[|S]))
  else if mgu(cntct[_,drlh[|T]],B)
       then interdrlh(drlh[drlh[|T]|R],drlh[|S])
       else interdrlh(drlh[|R],drlh[|S]).
```

For example, the call **interdrlh(JohnBeliefs,MaryBeliefs)** returns exactly the "maximum belief overlap" used as the variable **JohnMaryShared** in section 4:

```
drlh[tup[buy,john,house,linda],
     tup[like,john,mary],
     tup[like,mary,john],
     tup[mother,linda,mary],
     drlh[tup[command,marco,paul,greg,fred]],
     car,
     fido]
```

The operation of **DR$\mathcal{L}$H difference** has a structurally very similar definition, hence is not further discussed here.

The **subDR$\mathcal{L}$H** predicate generalizes the usual subset predicate essentially by an obvious treatment of contacted arguments (first three clauses) and by employing the labelnode membership predicate **membdrlh**, discussed below (sixth clause).

```
subdrlh(cntct[B,drlh[|R]],cntct[C,drlh[|S]]) :-&
  if mgu(B,C) then subdrlh(drlh[|R],drlh[|S]) else false.
subdrlh(drlh[|R],cntct[C,drlh[|S]]) :-& subdrlh(drlh[|R],drlh[|S]).
subdrlh(cntct[B,drlh[|R]],drlh[|S]) :-& false.
subdrlh(drlh[],drlh[|S]).
subdrlh(drlh[tup[|Y]|R],drlh[|S]) !-&
  if membtup(tup[|Y],tup[|S])
  then subdrlh(drlh[|R],drlh[|S])
  else false.
subdrlh(drlh[B|R],drlh[|S]) :-&
  if membdrlh(B,drlh[|S])
  then subdrlh(drlh[|R],drlh[|S])
  else false.
```

For example, the call **subdrlh(JohnBeliefs,MaryBeliefs)** returns **false**, whereas the call **subdrlh(JohnMaryShared,MaryBeliefs)** returns **true**.

Generalizing set membership, the **DR$\mathcal{L}$H member** predicate tests whether a labelnode occurs in a DR$\mathcal{L}$H. The first **membdrlh** clause reduces a contacted DR$\mathcal{L}$H argument to an uncontacted one. The second clause expects a DR$\mathcal{L}$H beginning with a hyperarc and returns **true** if **membarrow** (see below) can find the labelnode in it; otherwise, **membdrlh** recurses into the DR$\mathcal{L}$H remainder. While the third clause requires equality between an arbitrary labelnode and the DR$\mathcal{L}$H front, the fourth clause is satisfied with a similpotence relationship between an uncontacted complex labelnode and its contacted version at the DR$\mathcal{L}$H front (for such facts, neck cut is indicated by a "!"-suffix). The fifth clause just recurses into the DR$\mathcal{L}$H remainder, and the sixth clause returns **false** if the empty DR$\mathcal{L}$H is reached.

```
membdrlh(B,cntct[_,drlh[|R]]) :-& membdrlh(B,drlh[|R]).
membdrlh(B,drlh[tup[|Y]|R]) !-&
  if membarrow(B,tup[|Y])
  then true
  else membdrlh(B,drlh[|R]).
membdrlh(B,drlh[B|R])!
membdrlh(drlh[|S],drlh[cntct[_,drlh[|S]]|R])!
membdrlh(B,drlh[_|R]) !-& membdrlh(B,drlh[|R]).
membdrlh(B,drlh[]) :-& false.
```

For example, the second clause causes both the call **membdrlh(gang,JohnBeliefs)** and the call **membdrlh(drlh[tup[command,marco,paul,greg,fred]],JohnBeliefs)** to return **true**, using the **gang** hyperarc; if this were removed from **john**'s beliefs, **membarrow** would still cause the latter call to return **true**, using the **hire** hyperarc.

The **membarrow** predicate tests such similpotence membership of an uncontacted complex labelnode in a **tup** structure containing its contacted version (second clause). For all other argument types, truth-value computation is done exactly as in **membtup**, the normal membership predicate for tuples (remaining clauses).

```
membarrow(B,tup[B|Y])!
membarrow(drlh[|S],tup[cntct[_,drlh[|S]]|Y])!
membarrow(B,tup[_|Y]) !-& membarrow(B,tup[|Y]).
membarrow(B,tup[]) :-& false.
```

B The Hyperarc and Labelnode Merging Functions

The **mergearrow** function merges a hyperarc into a normalized DR$\mathcal{L}$H so as to produce an extended normalized DR$\mathcal{L}$H. The first clause ends recursion for an empty DR$\mathcal{L}$H argument **drlh[]**, inserting the hyperarc argument **A**. The second clause just returns a DR$\mathcal{L}$H argument **drlh[A|R]** starting with **A**. The third clause compares **A** with an arbitrary first DR$\mathcal{L}$H element **X**: if **A** is less than **X**, in the sense of the canonical DR$\mathcal{L}$H-element comparison function **eless**, **A** is constructed to the front of the DR$\mathcal{L}$H argument with all labelnodes used in **A** 'adsorbed' by an auxiliary **eatboxes**; otherwise, **A** must be greater than **X** (equality was tested by the previous clause), so **X** is constructed to the recursion result of **mergearrow** applied to **A** and the DR$\mathcal{L}$H without **X**.

```
mergearrow(A,drlh[])       !-& drlh[A].
mergearrow(A,drlh[A|R])    !-& drlh[A|R].
mergearrow(A,drlh[X|R])    :-& if eless(A,X)
                               then consdrlh(A,eatboxes(A,drlh[X|R]))
                               else consdrlh(X,mergearrow(A,drlh[|R])).
```

The function **eatboxes** leaves hyperarcs, i.e. structures **tup[|Z]**, in its DR$\mathcal{L}$H argument unchanged (second clause), but removes labelnodes, i.e. all other terms **B**, that are a **membarrow** (see end of appendix A) of its hyperarc argument **tup[|Y]** (third clause).

```
eatboxes(tup[|Y],drlh[])               !-& drlh[].
eatboxes(tup[|Y],drlh[tup[|Z]|R])  !-& consdrlh(tup[|Z],
                                              eatboxes(tup[|Y],drlh[|R])).
eatboxes(tup[|Y],drlh[B|R]) :-& if membarrow(B,tup[|Y])
                                   then eatboxes(tup[|Y],drlh[|R])
                                   else consdrlh(B,eatboxes(tup[|Y],drlh[|R])).
```

For example, the call **mergearrow(tup[1,2,3],drlh[1,4])** returns **drlh[tup[1,2,3],4]**.

The **mergebox** function merges an (isolated) labelnode into a normalized DR$\mathcal{L}$H, again producing an extended normalized DR$\mathcal{L}$H. As in **mergearrow**, the first two clauses handle emptiness and idempotence. The third clause captures one case of "similpotence" [Bol84]: a complex labelnode **drlh[|S]** without contact labelnode to be merged into a DR$\mathcal{L}$H starting with some contacted version **cntct[C,drlh[|S]]** is no longer uncontacted, i.e. becomes swallowed by returning the DR$\mathcal{L}$H unchanged. The fourth clause deals with the "adsorption" of a labelnode **B** by a hyperarc **tup[|Y]** of the DR$\mathcal{L}$H: if **membarrow** finds a **B** occurrence in **tup[|Y]**, **B** cannot be an isolated labelnode of the DR$\mathcal{L}$H, which is thus returned unchanged; otherwise, **consdrlh** puts **tup[|Y]** into the recursive **mergebox** result for **B** and the DR$\mathcal{L}$H without **tup[|Y]**. The fifth clause mainly treats commutativity: if **B** is **eless** than the DR$\mathcal{L}$H element **C**, then **B** becomes the front of the DR$\mathcal{L}$H; otherwise, **C** is constructed to the result of **mergebox** applied to **B** and the DR$\mathcal{L}$H without **C**. Also, in the **eless** branch, another case of similpotence is treated, comparable to the adsorption treatment in **mergearrow**'s third clause: if **B** has the form of a contacted complex labelnode **cntct[_,drlh[|S]]** (i.e. has a most general unifier with it), any uncontacted version **drlh[|S]** is removed from the remainder DR$\mathcal{L}$H (**removedrlh** corresponds to **eatboxes** called with a length-one **tup**); otherwise, the remainder is not changed.

```
mergebox(B,drlh[])         !-&  drlh[B].
mergebox(B,drlh[B|R])      !-&  drlh[B|R].
mergebox(drlh[|S],drlh[cntct[C,drlh[|S]]|R]) !-&  drlh[cntct[C,drlh[|S]]|R].
mergebox(B,drlh[tup[|Y]|R]) !-& if membarrow(B,tup[|Y])
                                then drlh[tup[|Y]|R]
                                else consdrlh(tup[|Y],mergebox(B,drlh[|R])).
mergebox(B,drlh[C|R])         :-& if eless(B,C)
                                then if mgu(cntct[_,drlh[|S]],B)
                                  then consdrlh(B,removedrlh(drlh[|S],
                                                             drlh[C|R]))
                                  else drlh[B,C|R]
                                else consdrlh(C,mergebox(B,drlh[|R])).
```

For example, the call `mergebox(1,drlh[tup[1,2,3],4])` returns `drlh[tup[1,2,3],4]`.

C The Traversal Function

In its first clause, **traverse** finds a complete path if the front element of the top path equals the **Goal** argument: it returns the **neststacked** **Pathstack** argument (see below) with the sequences reversed in all levels by an auxiliary **revtuprec**. The second clause performs hyperarc transits in the top DR$\mathcal{L}$H **Net** of **Netstack**, **Starting** from the front labelnode of the top path. The "transition finder" **findarrow** binds **Resttup** to $\texttt{tup}[a_{i+1},\ldots,a_m]$ for each hyperarc $\texttt{tup}[a_1,\ldots,a_i,a_{i+1},\ldots,a_m]$ in **Net** with a_i = **Start**. The non-deterministic **membtup** variant **membtupall** binds **Next** to successive elements of **Resttup**. (**membtup** and **membtupall** correspond to PROLOG **member** versions with and without neck cut, respectively.) For avoiding circles, **membtup** is used to make sure that **Next** is **not** yet in the top path. With its top path extended by each **Next** labelnode found by these three premises, **traverse** is called recursively. The third clause shifts down into the level of **cntct[B,drlh[|R]]** at the front of the top path: if the top-path remainder **tup[|Path]** does **not** have the form **tup[tup[B|_]|_]** of an immediately preceding shift-up done in the fourth clause, a shift-down is performed by recursively calling **traverse** with **drlh[|R]** pushed onto **Netstack** and **tup[B]** pushed onto **Pathstack**. The fourth clause shifts up to the level of **Net2**, the next-to-top DR$\mathcal{L}$H: if **cntct[Start,Net1]**, i.e. the top-path front used as a contact labelnode of the top net, is **not** yet a **membtup** of **tup[|Pathup]**, the next-to-top path, a shift-up is performed by recursively calling **traverse** with **Net1** popped from **Netstack** and the top path **tup[Start|Path]** popped from **Pathstack** but **tup[|Pathup]** extended to **tup[cntct[Start,Net1],tup[Start|Path]|Pathup]** (the new top path thus contains the old top path as an embedded sequence).

```
traverse(Netstack,tup[tup[Goal|Path]|Pathstack],Goal) :-&
  revtuprec(neststack(tup[tup[Goal|Path]|Pathstack])).
traverse(tup[Net|Netstack],tup[tup[Start|Path]|Pathstack],Goal) :-
  findarrow(Net,Start,Resttup),
  membtupall(Next,Resttup),
  not(membtup(Next,tup[Start|Path])) &
  traverse(tup[Net|Netstack],tup[tup[Next,Start|Path]|Pathstack],Goal).
traverse(tup[|Netstack],tup[tup[cntct[B,drlh[|R]]|Path]|Pathstack],Goal) :-
  not(mgu(tup[tup[B|_]|_],tup[|Path])) &
  traverse(tup[drlh[|R]|Netstack],
           tup[tup[B],tup[cntct[B,drlh[|R]]|Path]|Pathstack],
           Goal).
traverse(tup[Net1,Net2|Netstack],
         tup[tup[Start|Path],tup[|Pathup]|Pathstack],
         Goal) :-
  not(membtup(cntct[Start,Net1],tup[|Pathup])) &
  traverse(tup[Net2|Netstack],
           tup[tup[cntct[Start,Net1],tup[Start|Path]|Pathup]|Pathstack],
           Goal).
```

The transition finder **findarrow** tries to partition successive hyperarcs **tup[|Y]** of its DR$\mathcal{L}$H argument such that its labelnode argument **Start** precedes a non-empty hyperarc postfix to be bound to its **tup** argument. For this relational partitioning an inverse use of the PROLOG-append-like tuple-concatenation relation **appendtup** is made on **tup[|Y]** in the first clause. All leading **tup[|Y]** elements of the normalized DR$\mathcal{L}$H are recursively stripped off in the second clause.

```
findarrow(drlh[tup[|Y]|R],Start,tup[Succ|Nodes]) :-
  appendtup(tup[|_],tup[Start,Succ|Nodes],tup[|Y]).
findarrow(drlh[tup[|Y]|R],Start,Resttup) :-
  findarrow(drlh[|R],Start,Resttup).
```

The auxiliary function **neststack** recursively front-nests a **tup**-stack of tuples.

```
neststack(tup[tup[|Y]]) :-& tup[|Y].
neststack(tup[tup[|Y],tup[|Z]|Remtups]) :-&
  neststack(tup[tup[tup[|Y]|Z]|Remtups]).
```

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
Stuhlsatzenhausweg 3
6600 Saarbrücken 11
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen oder die aktuelle Liste von erhältlichen Publikationen können bezogen werden von der oben angegebenen Adresse.

DFKI Publications

The following DFKI publications or the list of currently available publications can be ordered from the above address.

DFKI Research Reports

RR-90-01
Franz Baader
Terminological Cycles in KL-ONE-based Knowledge Representation Languages
33 pages

RR-90-02
Hans-Jürgen Bürckert
A Resolution Principle for Clauses with Constraints
25 pages

RR-90-03
Andreas Dengel & Nelson M. Mattos
Integration of Document Representation, Processing and Management
18 pages

RR-90-04
Bernhard Hollunder & Werner Nutt
Subsumption Algorithms for Concept Languages
34 pages

RR-90-05
Franz Baader
A Formal Definition for the Expressive Power of Knowledge Representation Languages
22 pages

RR-90-06
Bernhard Hollunder
Hybrid Inferences in KL-ONE-based Knowledge Representation Systems
21 pages

RR-90-07
Elisabeth André, Thomas Rist
Wissensbasierte Informationspräsentation: Zwei Beiträge zum Fachgespräch Graphik und KI:
1. Ein planbasierter Ansatz zur Synthese illustrierter Dokumente
2. Wissensbasierte Perspektivenwahl für die automatische Erzeugung von 3D-Objektdarstellungen

24 pages

RR-90-08
Andreas Dengel
A Step Towards Understanding Paper Documents
25 pages

RR-90-09
Susanne Biundo
Plan Generation Using a Method of Deductive Program Synthesis
17 pages

RR-90-10
Franz Baader, Hans-Jürgen Bürckert, Bernhard Hollunder, Werner Nutt, Jörg H. Siekmann
Concept Logics
26 pages

RR-90-11
Elisabeth André, Thomas Rist
Towards a Plan-Based Synthesis of Illustrated Documents
14 pages

RR-90-12
Harold Boley
Declarative Operations on Nets
43 pages

DFKI Technical Memos

TM-89-01
Susan Holbach-Weber
Connectionist Models and Figurative Speech
27 pages

TM-90-01
Som Bandyopadhyay
Towards an Understanding of Coherence in Multimodal Discourse
18 pages

TM-90-02
Jay C. Weber
The Myth of Domain-Independent Persistence
18 pages

DFKI Documents

D-89-01
Michael H. Malburg & Rainer Bleisinger
HYPERBIS: ein betriebliches Hypermedia-Informationssystem
43 Seiten

D-90-01
DFKI Wissenschaftlich-Technischer Jahresbericht 1989
45 pages

D-90-02
Georg Seul
Logisches Programmieren mit Feature - Typen
107 Seiten

D-90-03
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
Abschlußbericht des Arbeitspaketes PROD
36 Seiten

D-90-04
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
STEP: Überblick über eine zukünftige Schnittstelle zum Produktdatenaustausch
69 Seiten

Declarative Operations on Nets

Harold Boley

RR-90-12

Research Report