Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-92-27

Integrating Bottom-up and Top-down Reasoning in CoLab

Martin Harm, Knut Hinkelmann, Thomas Labisch

September 1992

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

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

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11
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, Philips, SEMA Group Systems, Siemens and Siemens-Nixdorf. 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
- ❏ Intelligent Communication Networks
- ❏ Intelligent Cooperative Systems.

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.

Prof. Dr. Gerhard Barth
Director

Integrating Bottom-up and Top-down Reasoning in COLAB

Martin Harm, Knut Hinkelmann, Thomas Labisch

DFKI-D-92-27

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITW-8902 C4).

© Deutsches Forschungszentrum für Künstliche Intelligenz 1992

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all aplicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.

Integrating Bottom-up and Top-down Reasoning in CoLab

Martin Harm
Knut Hinkelmann
Thomas Labisch
DFKI GmbH, Postfach 2080, D-6750 Kaiserslautern
e-mail: {harm,hinkelma,labisch}@dfki.uni-kl.de

Contents

1 The Knowledge Compilation Laboratory CoLab 1

2 Rule System Overview 3

3 The CoLab Toplevel 5

4 Set-oriented bottom-up Reasoning 7
4.1 Semi-Naive Bottom-Up Reasoning . . . 8
4.1.1 The Semi-Naive Algorithm . . . 9
4.1.2 The Differential Function . . . 10
4.1.3 Rule and Fact Indexing . . . 12
4.2 Magic Set Transformation . . . 13
4.2.1 Adornment Step . . . 14
4.2.2 Generation Step . . . 15
4.2.3 Modification Step . . . 16
4.2.4 Fact Derivation . . . 16
4.3 Rule Compilation . . . 18

5 Tuple-oriented bottom-up Reasoning 19
5.1 Transformation of Rules . . . 20
5.2 Predefined Control Strategies of Forward Reasoning . . . 21
5.3 Retaining Derived Facts . . . 23
5.4 Compilation into the RFM . . . 25
5.4.1 Forward Code Area . . . 25
5.4.2 Retain Stack . . . 27

6 Conclusion 28

A Commands for the Rule Component of COLAB 29

Abstract

The knowledge compilation laboratory CoLab integrates declarative knowledge representation formalisms, providing source-to-source and source-to-code compilers of various knowledge types. Its architecture separates taxonomical and assertional knowledge. The assertional component consists of a constraint system and a rule system, which supports bottom-up and top-down reasoning of Horn clauses. Two approaches for forward reasoning have been implemented. The first set-oriented approach uses a fixpoint computation. It allows top-down verification of selected premises. Goal-directed bottom-up reasoning is achieved by a magic-set transformation of the rules with respect to a goal. The second tuple-oriented approach reasons forward to derive the consequences of an explicitly given set of facts. This is achieved by a transformation of the rules to top-down executable Horn clauses. The paper gives an overview of the various forward reasoning approaches, their compilation into an abstract machine and their integration into the CoLab shell.

Chapter 1

The Knowledge Compilation Laboratory CoLab

Declarative representations describe logically *what* the knowledge expresses without at the same time prescribing imperatively *how* it is to be used. Such a high descriptive level not only permits several uses of the same knowledge base but also enhances the readability, maintenance and parallelization of knowledge bases. Moreover, the orientation towards logic (usually, variations of first-order predicate calculus) permits a clear semantics for representation languages and eases the tough business of knowledge base verification/validation.

As a step in that direction the prototypical knowledge Compilation LABoratory COLAB [Boley *et al.*, 1991a] has been implemented on the basis of Lisp. It supports a hybrid integration of four principal representation languages, declaratively developing and extending well-known AI formalisms: a terminological language TAXON, a constraint system CONTAX and a logic-programming formalism, which itself is divided into a bottom-up and a top-down reasoning component with functions, FORWARD and RELFUN, respectively. A hybrid knowledge base can contain items from *all* subsystems. Tags indicate the type of a knowledge item and determine how it has to be processed. Dynamic cooperation of the subsystems is organized through access primitives providing an interface to the respective reasoning services.

COLAB's architecture corresponds to terminological systems like KRYPTON[Brachman *et al.*, 1983] separating taxonomic and affirmative (often called assertional) knowledge. Taxonomic knowledge is represented by intensional concept definitions which are automatically arranged in a subsumption hierarchy. Thereby inconsistencies of concept definitions can be detected. This contrasts to conventional frame-based and object-oriented expert system shells where the organization of the class hierarchy is in the responsibility of the programmer. The structure of the concept hierarchy as well as the 'content' of the concept definitions is available via access primitives to the other subformalisms. This permits more compact formulations of affirmative knowledge by referring to concepts and leads to more efficient processing if reasoning about individuals can be lifted to reasoning on the concept level. The affirmative part provides efficient reasoning with different kinds of relational or functional knowledge using tailored inference engines. For affirmative knowledge represented as net-like, non-recursive relations, called *constraint nets*, COLAB supplies constraint propagation as an efficient reasoning mechanism. Relational knowledge in the form of Horn rules is processed by *forward* and *backward* chaining. In a single query some rules can be used for top-down problem decomposition and others for bottom-up deduction. The backward component is also suited for

expressing (non-deterministic) functional dependencies.

Besides interpreting these languages for interactive knowledge base development, CoLab provides *source-to-code translators* for compiling knowledge bases down to efficient abstract machines. Also, CoLab provides *source-to-source transformers* between various knowledge types, for both user convenience and machine efficiency. For example, bidirectional rules can be split into rules specially tailored for forward and backward chaining, which then can be made more efficient in a further transformation step by additional control instructions. The magic-set transformation presented in Section 4.2 is suitable for goal-directed bottom-up reasoning. Section 5 will describe a transformation of rules to Horn clauses, which simulate forward chaining in a backward chaining system.

The following sections will describe the forward-chaining logic programming subsystem of CoLab. For a more detailed description of CoLab and its other subsystems see [Boley *et al.*, 1991a].

Chapter 2

Rule System Overview

Reasoning in rule-based and logic-programming systems can be preformed using two principal directions: while backward inference applies the rules in a top-down fashion, forward inference begins with the facts in the knowledge base, reasoning bottom-up to derive new facts. One major idea of declarative programming is the separation of logic from control shifting the responsibility for control to the execution mechanism. The programmer should care as little as possible about it. For a rule system this means, in the ideal case, that the application direction of a rule need not be visible to the programmer. Deduction rules in general have the following form:

$$P_1 \wedge P_2 \wedge \ldots \wedge P_m \rightarrow C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

The preconditions $P_1, \ldots, P_m$ on the left-hand side of the rule are literals which must be satisfied for the rule to fire. The conclusions $C_1, \ldots, C_n$ of such a *deduction rule* are facts which are true if the premises are satisfied – as opposed to production rules (cp. OPS5 [Forgy, 1981]) where the conclusion consists of operations modifying the working memory.

The rule component of CoLAB is a declarative logic programming system with Horn clauses as its basic representation scheme. Horn clauses are clauses with at most one positive literal, which is equivalent to restricting the syntax of rules to having only one conclusion:

$$\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_m \vee C \qquad \text{is equivalent to} \qquad P_1 \wedge P_2 \wedge \ldots \wedge P_m \rightarrow C$$

In the context of the paper the term *Horn rules* will be used synonymously for those non-unit *Horn clauses* that are used as deduction rules.

It should be noted that the restriction to Horn rules does not too much limit the expressive power of the rule language, compared to the general form (see above). For instance, a transformation of deduction rules with disjunctions of premises (PROLOG's ";") and conjunctions of pairwise independent conclusions to Horn clauses is straightforward and can be performed by a precompiler [Hinkelmann, 1991a].

Knowledge items in the CoLAB system are indicated by tags. The rule system can handle three types of rules:

rl: Bidirectional rules are indicated by the tag `rl` and can be used in both forward and backward direction

up: Rules indicated by the tag up can be used only bottom-up.

hn: Rules with the tag **hn** can be applied only in backward direction. The tag **hn** is an abbreviation for "hornisch" clauses and is took over from the RELFUN component.

The COLAB system is implemented in Lisp. Each knowledge item is represented as a list. Tags indicating the type of the knowledge item are the first element of this list. The second element is the conclusion of the rule and the remaining expressions are the rule's premises:

$$(<\text{tag}>\ <\text{Conclusion}>\ <\text{Premise}_1>\ \ldots\ <\text{Premise}_m>)$$

The syntax of the literals are similar to those in RELFUN. The literals of the rule are again Lisp lists: the predicate or operator name is the first element, the remaining elements are the arguments. Predicate names and constants are Lisp atoms, (universally quantified) variables are Lisp atoms starting with the underscore character "_". Lists are special terms with the functor "tup". For example the PROLOG-like rule

```
shoulder(s(X,Y),[ground(X),flank(Y)]) :- cylinder(X), ring(Y)
```

is written in COLAB as:

```
(hn (shoulder (s _x _y) (tup (ground _x) (flank _y)))
    (cylinder _x)
    (ring _y))
```

As a special feature, the bidirectional and the bottom-up rules may have more the one conclusion. The syntax of these rules is:

$(\ \{\texttt{rl} \mid \texttt{up}\}\quad (<conc_1>\ldots<conc_n>)\texttt{<-}\quad <premise_1>\ldots<premise_N>)$

The intended semantics of this rule is the same as the semantics of the following sequence of Horn rules:

$(\ \{\texttt{rl} \mid \texttt{up}\}\quad <conc_1>\quad <premise_1>\ldots<premise_N>)$

...

$(\ \{\texttt{rl} \mid \texttt{up}\}\quad <conc_N>\quad <premise_1>\ldots<premise_N>)$

The specific variety of a forward reasoning system depends on the facts which initially trigger a rule and on how the rule premises are proved. The rule system of COLAB offers two independent evaluation procedures: The first set-oriented approach (Section 4) interprets bottom-up rules using a fixpoint computation starting from all the facts in the program. The premises are verified by look-up in the fact base. A Magic-Set transformation is implemented for goal-directed reasoning. It is tightly coupled with RELFUN to achieve bidirectional reasoning. The second, tuple-oriented scheme (Section 5) reasons forward to derive the consequences of an explicitly given set of initial facts. Bottom-up and bidirectional rules are transformed to RELFUN Horn clauses, which are finally compiled into an extended RFM-System (a Warren Abstract Machine for logic programs [Warren, 1983], which is capable to handle functional clauses of RELFUN, [Boley, 1990]) with a special forward-code area. The premises of triggered rules are tested by the backward reasoning proof procedure of RELFUN.

Chapter 3

The CoLab Toplevel

The CoLab system is implemented in Common Lisp. After loading CoLab you can enter the CoLab toplevel by calling the function

```
> (colab)
```

The prompt changes to `colab>` and you can type a number of commands, which are displayed when typing a question mark "?". From the CoLab toplevel you can switch to a subsystem by typing the name of the subsystem you want to use. Thus, for instance, typing

```
colab> forward
```

switches to the rule system called FORWARD. Each subshell offers specific commands – besides the general CoLab commands. All the commands described in this paper are commands of the FORWARD-subsystem toplevel. There are five classes of commands to

- insert and remove knowledge items
- display knowledge items
- compile and transform knowledge items
- evaluate expressions
- call help and debugging facilities

To insert knowledge items into the system there is either the command `consult`, which loads a knowledge base from a file, or the commands `az` and `a0`, which assert single knowledge items. Similarly, the command `destroy` deletes a whole knowledge base, while `rx` retracts single knowledge items. The command `replace` is equivalent to a sequence of `destroy` and `consult`.

To display knowledge items on the screen there is the family of `listing` commands, which can also be spezialized to list particular (kinds of) knowledge items.

As already described in Section 2, besides the rules for bottom-up and top-down reasoning there are also so-called bidirectional rules, which can be evaluated in both directions. But since the top-down and the bottom-up reasoning system obtain their input from different

internal knowledge bases, the command `split-rules` has to be executed to split bidirectional `rl`-rules into their specialized bottom-up (up) and top-down (hn) versions.

Expressions can be evaluated either by interpretation or by an abstract machine after compilation. To select the evaluation approach the commands **`fw-inter`** and **`fw-emul`** are available. To easily recognize, which mode is effective, the prompt of the FORWARD subsystem changes between **`fwi>`** for interpreter and **`fwe>`** for emulator mode. The **`spy`** command displays detailed information of the reasoning process while evaluating an expression. This debugging facility can be switched off by typing the command **`nospy`**.

Further commands for compiling knowledge items and evaluating expressions depend on the particular evaluation strategy. The various strategies and their compilation are the subject of the remaining sections of this paper. For a detailed description of the whole set of commands see Appendix A.

To access the commands of the forward system without CoLab use the function **`fw`**. The calling convention for this function is:

(**fw** '<*command*> '<arg_1> '<arg_2> ... '<arg_n>)

This function works as if the cammand has been called directly in CoLab. The command **`rf-query`**, however, which calls the RELFUNsubsystem for the tuple-oriented forward reasoning approach (see Section 5) is treated specially, because no additional solutions can be asked for by explicit backtracking. Therefore the solutions are calculated all at once. The result is a list containing all the return values of the last argument. To fetch the values of some variables use as the last argument (`tup` <var_1> ... <var_n>). The backquotes which are used in the RELFUN subsystem have to be expanded into (`inst` ...) (see also the RELFUN manual [Boley *et al.*, 1991b]).

Chapter 4

Set-oriented bottom-up Reasoning

We will present two approaches for set-oriented bottom-up reasoning. The first one is the Semi-Naive evaluation strategy. It computes the least fixpoint of the knowledge items of a database. This evaluation is started by typing the command

```
fwi> eval
```

The derived facts are displayed together with the entire facts by the command

```
fwi> list-facts
```

But in some cases we only want to get a subset of these derived facts which satifies a certain condition, like a query. In this case there is the Magic-set transformation, which rewrites the rules of the database to avoid the application of rules and facts which are independent from the query. The Magic-set transformation is initiated by the command

```
fwi> magic-transform <goal_1> ... <goal_n>
```

The arguments <$goal_1$> ... <$goal_n$> are interpreted as a conjunction of goals. If we then apply the Semi-Naive bottom-up strategy on this new database with the command

```
fwi> magic-query <goal_1> ... <goal_n>
```

only facts, which are necessary to prove the query, are derived. It is possible to transform *and* evaluate a rule system for a particular query by using the command

```
fwi> magic-eval <goal_1> ... <goal_n>
```

The following subsections will give an introduction into the Semi-Naive evaluation and the Magic-set transformation as they are implemented in CoLab.

4.1 Semi-Naive Bottom-Up Reasoning

At the beginning of this section we will give the reader a brief overview about the used denotations, similar to [Bancilhon and Ramakrishnan, 1986]. For a detailed description of the Semi-Naive bottom-up component see [Labisch, 1991].

A *database* is an unordered set of Horn clauses. Given a database we can partition it into a set of ground unit clauses (facts) and the set of the remaining clauses (rules). The set of facts is called the *extensional* database and the set of rules is called the *intensional* database.

Different to [Bancilhon and Ramakrishnan, 1986] we have the following definitions. A *recursive predicate* is a predicate, appearing both as a premise and as a head in the rules of the intensional database, but not necessarily in the same rule. For a *recursive rule* at least one predicate of a premise is a recursive predicate. If the body of a rule contains exactly one recursive predicate, we call this rule *linear recursive*. In a *nonrecursive rule* no predicate of a premise occurs as a head of any other rule of the database. The predicates of these premises are called *nonrecursive predicates.*

Builtin predicates are also called *evaluable* predicates. The is-predicate is a special kind of builtin predicate. Its second argument is evaluated and unified with the first argument. If the first argument is a free variable its value will be the result of evaluating the second argument. Then a rule is *safe*, if all variables appearing in the head also appear in a nonevaluable premise of the body, or as the first argument of an is-term, whose second argument has not to be a variable. In CoLab a Horn rule is *bottom-up evaluable*, if it is safe.

Example: Let the intensional database contain only the following rules:

```
(rl (cylinder _name _length _radius)
    (truncone _name  _length _radius _radius))
(rl (rcone _name _length _radius)
    (truncone _name _length _radius 0))
(rl (rot-part _name)
    (rspear _name _length _radius))
(rl (rspear (c _cyl _cone) _length _radius)
    (cylinder  _cyl _length1 _radius)
    (connected _cyl _cone)
    (rcone _cone _length2 _radius)
    (is _length (+ _length1 _length2)))
(rl (price _x _y)
    (rot-part _x _y)
    (> _y 0))
```

Then there are the following notations:

- `cylinder`, `rcone`, `rot-part` and `rspear` are recursive predicates.
- `truncone` and `connected` are nonrecursive predicates, also called base predicates.
- The third and fourth rule are recursive, the third one is even linear recursive.

- The first and second rule are nonrecursive.
- All but the last rule are safe, because the variable _y in the fifth rule does not appear as argument of a premise.

There are also premise predicates in the body of a rule, which are neither defined in the FORWARD databases nor are they builtins, but appear in one of the RELFUN databases. This indicates, that we cannot prove them bottom-up. But a backward test can be made by a call to the RELFUN-Interpreter. We call this kind of predicates *backward predicates*. Examples are the `member`- and `append`-predicates. In our implementation the builtins are proved together with the backward-predicates in the top-down direction.

4.1.1 The Semi-Naive Algorithm

Semi-Naive evaluation is an improvement of Naive evaluation. Both are iterative, compiled bottom-up strategies. Their application range is the set of bottom-up evaluable rules.

The Naive evaluation computes all direct successors of the existing facts. These conclusions (derivations) are added to the fact database. In the next step again all successors of the extended fact database are evaluated.

For Naive evaluation in the first iteration step all rules whose premises are satisfied by facts F_0 of the extensional database are evaluated. The conclusions C_0 of these applied rules are added to the set of the derived facts. So we get a new set of facts $F_1 = F_0 \cup C_0$. For all further iteration steps only those rule are applied that are satisfied by facts of the set F_i and deliver conclusions C_i. Now we get $F_{i+1} = F_i \cup C_i$, and the evaluation stops, if $F_{i+1} = F_i$. It is obvious that all facts computed in iteration steps $1, \ldots, n-1$, are computed again in iteration step n. The Semi-Naive strategy tries to avoid such multiple computations.

The idea of Semi-Naive evaluation is to compute only the *new* derived facts for each iteration step. Therefore for each step n only rules with at least one premise satisfied by a *new* derived fact are applied. This method is described in the algorithm below.

Algorithm: Sєmi-Naive Evaluation *Let $\mathcal{F}$ be the set af all facts, $\mathcal{NDF}$ be the list of new facts derived in the current cycle, and $\mathcal{PDF}$ be the list fo all facts derived in the previous cycle.*

1. *Start with the initial facts $\mathcal{F}$, set $\mathcal{NDF} := \emptyset$ and $\mathcal{PDF} := \mathcal{F}$*
2. *For every clause, $H \leftarrow P_1, \ldots, P_m$ in $\mathcal{R}$ for which there is a substitution σ, such that at least one $P_i\sigma$ is in $\mathcal{PDF}$ and all $P_j\sigma$, $j \in \{1, \ldots, m\} \setminus i$, are in $\mathcal{F}$, set $\mathcal{NDF} := \mathcal{NDF} \cup \{H\sigma\}$*
3. *If $\mathcal{NDF} = \emptyset$, then stop, else set $\mathcal{PDF} := \mathcal{NDF} \setminus \mathcal{F}$, $\mathcal{F} := \mathcal{F} \cup \mathcal{PDF}$ and $\mathcal{NDF} := \emptyset$ and goto 2*

□

In practise we have to *differentiate* the rules: all rules with at least one recursive predicate in the body are replaced by new generated rules [Ullman, 1989], such that all the *new* facts

are then derived by one of the differentiated rules. For each premise G_j with a recursive predicate g_j of a rule

$$(rl \quad H \quad G_1 \quad G_2 \quad \cdots \quad G_n)$$

we get a rule

$$(rl \quad \Delta H \quad G_1 \quad G_2 \quad \cdots \quad G_{j-1} \quad \Delta G_j \quad G_{j+1} \quad \cdots \quad G_n)$$

Now the rule only can be applied for new derived facts of g_j. To lead to the differential function Δ, let us analyze the bottom-up evaluation of a recursive rule. Let the recursive predicate P appear in the head of rule

$$(rl \quad P \quad \Phi(p_1, p_2, \ldots, p_n, q_1, q_2, \ldots, q_m))$$

where Φ is a first order formula with atoms $(p_1, p_2, \ldots, p_n, q_1, q_2, \ldots, q_m)$. The p_j are recursive predicates and the q_l are base predicates, backward predicates or builtins. Let $p_j(i)$ the value of the predicate p_j in the iteration step i, i.e. the set of facts that p_j satisfies. Then in iteration step i we compute

$$\Phi(p_1(i), p_2(i), \ldots, p_n(i), q_1, q_2, \ldots, q_m)$$

In this iteration step i for each p_j a set of *new* tuples denoted by $\delta p_j(i)$ can be derived. Thus, the value of p_j at the beginning of step $i+1$ is $p_j(i+1) = p_j(i) \cup \delta p_j(i)$. Then at step $i+1$ we evaluate

$$\Phi((p_1(i) \cup \delta p_1(i)), \ldots, (p_n(i) \cup \delta p_n(i)), q_1, q_2, \ldots, q_m)$$

which, of course, recomputes all the previous derivations because of the monotony of Φ. The ideal, however, is to compute only the *new* facts, i.e. the expression:

$$\begin{aligned}
&\Delta\Phi(p_1(i), \delta p_1(i), \ldots, p_n(i), \delta p_n(i), q_1, q_2, \ldots, q_m) = \\
&\quad \Phi((p_1(i) \cup \delta p_1(i)), \ldots, (p_n(i) \cup \delta p_n(i)), q_1, q_2, \ldots, q_m) - \\
&\quad \Phi(p_1(i), p_2(i), \ldots, p_n(i), q_1, q_2, \ldots, q_m)
\end{aligned}$$

The basic principle of the Semi-Naive method is the computation of the *differential* $\Delta\Phi$ of Φ instead of the entire Φ at each step. An exact examination of $\Delta\Phi$ follows in the next section.

4.1.2 The Differential Function

In this section we will develop a differential function for rules with recursive predicates. At the beginning we inspect the case with only one recursive predicate P as a premise of a linear recursive rule. In [Ullman, 1989] a simple rewrite rule for generating $\Delta\Phi$ is presented:

$$\textbf{if} \quad \Phi(p, q) = p(X, Y), q(Y, Z) \quad \textbf{then} \quad \Delta\Phi(p, \delta p, q) = \delta p(X, Y), q(Y, Z)$$

More generally, if Φ is linear recursive, then only the recursive predicate p is replaced by δp. In the case of nonlinear recursive rules we cannot find such an easy solution. Let p and r be two recursive predicates and q a base predicate. Then the rewrite rule is:

$$\begin{aligned}
\textbf{if} \quad & \Phi(p, r, q) &=\ & p(X, Y), r(Y, Z), q(Z, W) \\
\textbf{then} \quad & \Delta\Phi(p, \delta p, r, \delta r, q) &=\ & \delta p(X, Y), r(Y, Z), q(Z, W) + \\
& & & p(X, Y), \delta r(Y, Z), q(Z, W) + \\
& & & \delta p(X, Y), \delta r(Y, Z), q(Z, W)
\end{aligned}$$

Note, that this is not an exact differential, because the three terms of the sum are not necessarily disjoint, but it is a reasonable approximation. It is obvious that the amount of rules generated by the differential function rapidly increases. For a linear recusive rule only the recursive premise p is replaced by δp, so that we get *one* new rule instead of the original one. For a rule with two recursive premises *three* new rules replace the old one and so on. Thus, the increase of generated rules is exponential:

Theorem 4.1.1 *For a rule with* **n** *recursive premises the number of new rules generated by the above differential function is* $\mathbf{2^n - 1}$.

To avoid an exponential increase, it would be favourable to develop an algorithm managing with linear increase. Indead this is done in the following. In [Labisch, 1991] the following inferences were done, that lead us after all to the same result as presented in [Balbin and Ramamohanarao, 1987]. Let us inspect here a bigger example. We have a rule

$$p :- p_1, p_2, p_3, q$$

with the recursive premises p_i and a base predicate, backward predicate or builtin q. According to the previous theorem we get *seven* new rule. We can order them into three blocks:

$$\begin{array}{llllllll}
1 & \Delta p & :- & \delta \mathbf{p_1} & , & p_2 & , & p_3 & , & q \\
\\
2 & \Delta p & :- & p_1 & , & \delta \mathbf{p_2} & , & p_3 & , & q \\
 & \Delta p & :- & \delta p_1 & , & \delta \mathbf{p_2} & , & p_3 & , & q \\
\\
3 & \Delta p & :- & p_1 & , & p_2 & , & \delta \mathbf{p_3} & , & q \\
 & \Delta p & :- & p_1 & , & \delta p_2 & , & \delta \mathbf{p_3} & , & q \\
 & \Delta p & :- & \delta p_1 & , & p_2 & , & \delta \mathbf{p_3} & , & q \\
 & \Delta p & :- & \delta p_1 & , & \delta p_2 & , & \delta \mathbf{p_3} & , & q
\end{array}$$

Different columns are marked in every block: These columns do not change inside a block; they have the value δp_i, with i being the number of the block. Those columns inside a block i with an index j, $j < i$, contain δp_j as well as p_j, while those columns with a index k, $k > i$, only contain p_k and q.

For the columns with an index smaller than the number of the block we need not to distinguish between p_j (all relations of the last step) and δp_j (the new relations of the last step), because both relations are evaluated for a rule. Therefore we introduce a new relation called *ap*. This relation contains just the sum of p and δp. Note, that we only need the sum and not the union of p_j and δp_j, because δp_j is the symmetrical difference of the new and old relations of the last step. Therefore p_j and δp_j are already disjoint. Rewriting the rules we get

$$\begin{array}{llllllll}
1 & \Delta p & :- & \delta \mathbf{p_1} & , & p_2 & , & p_3 & , & q \\
\\
2 & \Delta p & :- & ap_1 & , & \delta \mathbf{p_2} & , & p_3 & , & q \\
\\
3 & \Delta p & :- & ap_1 & , & ap_2 & , & \delta \mathbf{p_3} & , & q
\end{array}$$

From originally *seven* rules only *three* are left. So the exponential increase shrinked to a linear one. The only restriction is, that the recursive predicates have to stand at the beginning of the rule's body. For bottom-up evaluation such a reordering of the premises does not affect computation. As a final result we have

Theorem 4.1.2 *There exists a differential function that generates for a rule with* n *recursive premises only* n *new differentiated rules.*

4.1.3 Rule and Fact Indexing

For a faster access to the facts and rules of the database we will introduce an indexing scheme. With this method it is possible to find a rule which is triggered by a certain fact, without sequentialy searching the database.

The facts of the extensional database are collected by their predicate names. For every predicate of the extensional database there exists a list of tuples called the relation of the predicate.

Rules are indexed by premise predicates. Thereby we have to distinguish between recursive and nonrecursive rules. Nonrecursive rules are triggered by nonrecursive premises, while recursive rules can only be triggered by recursive premises. Nonrecursive rules with a common nonevaluable body predicate are collected into one list. This indexing only applies to nonrecursive premises. For recursive rules only recursive premises are considered. Also, the number of recursive premises (called *level of recursivity*) is needed for the differentiation of recursive rules. In nonrecursive rules this value is always zero, while in recursive rules it is greater than zero. For both types we need a criterium to decide whether two rules with different triggers are actually the same rule. If it has once been applied with the first trigger, it may not be evaluated anymore in this step. Therefore the same rule with multiple triggers has a unique number, in order to be able to decide on a former application of it.

The structure of such a rule after indexing, which is important for the differentiation, looks like the following description:

$$(Pattern \;\; (Conclusion \;\; Body \;\; Level) \;\; Number)$$

where the single components are described below.

Pattern: the trigger of the rule

Conclusion: the head of the rule

Body: the remaining body of the rule without the trigger in the order of

1. remaining recursive premises
2. remaining nonrecursive premises
3. builtins and backward-predicates

Level: the number of all recursive premises of the rule

Number: the unique number of the rule

For example the rule

```
(rl (rspear (c _cyl _cone) _length _radius)
    (cylinder  _cyl _length1 _radius)
    (connected _cyl _cone)
    (rcone _cone _length2 _radius)
    (is _length (+ _length1 _length2)))
```

with conclusion **rspear**, recursive predicates **cylinder** and **rcone**, nonrecursive predicate **connected** and builtin **is** is first stored as a list under the property *rules* of the symbol **cylinder**

```
Pattern:        (cylinder _cyl _length1 _radius)
  Conclusion:   (rspear (c _cyl _cone) _length _radius)
  Body:         ((rcone _cone _length2 _radius)
                 (connected _cyl _cone)
                 (is _length (+ _length1 _length2)))
  Level:        2
Number:         1
```

and as a second list, now under the property *rules* of the symbol **rcone**

```
Pattern:        (rcone _cone _length2 _radius)
  Conclusion:   (rspear (c _cyl _cone) _length _radius)
  Body:         ((cylinder _cyl _length1 _radius)
                 (connected _cyl _cone)
                 (is _length (+ _length1 _length2)))
  Level:        2
Number:         1
```

The variable-binding which we get by matching a fact against the pattern is propagated to the premises of the rule, so that every premise has to be proved only once. It should be noted that all the premises and the conclusion are stored only once. Multiple occurences are shared by links.

The last optimization treats conclusions which are ground (containing no variables). We only have to evaluate them once. As soon as the conclusion is derived, all occurences of the rule can be found by the unique *number* and then be deleted, because this rule cannot deliver new facts anymore.

4.2 Magic Set Transformation

Magic Sets optimize the bottom-up evaluation by simulating the sideway passing of bindings for goal-directed reasoning à la PROLOG [Beeri and Ramakrishnan, 1991]. Depending on the goals new rules are introduced. This cuts down the number of potentially relevant facts and rules. The application domain is the set of bottom-up evaluable rules. In our framework we choose the generalized version of Magic Sets, that can also handle rules with more than one recursive predicate without a big loss of efficiency. In the original transformation there was no improvement over Semi-Naive evaluation in this case. The transformation of generalized Magic Sets can be applied to any program with at least one given query.

This method can be broken down into three essential steps. In the next section we describe the *adornment step*, in which the relationship between a bound argument in the rule head and the bindings in the rule body is made explicit. The *generation step*, in which the adorned program is used to generate the magic rules that simulate the top-down evaluation process, follows afterwards. At last we have a *modification step*, in which the adorned rules are modified by the magic rules generated in the previous step.

This is similar to the description in [Naqvi and Tsur, 1989], but we do not use the original Magic Sets method but the generalized version of it, so that there are some essential differences in the following.

4.2.1 Adornment Step

An adornment *a* is denoted by a string that denotes the binding status (free or bound) of each of the arguments of the predicate, e.g. `(sg-bf _x _y)` indicates that the first argument is bound and the second one is free. A *distinguished* argument is recursively defined by the following rules:

1. it is bound by an adornment *a* or
2. it is a constant or
3. it appears in a base predicate occurence that has a distinguished argument.
4. it appears in a derived predicate occurence that has a distinguished argument.

With the last point we leave the original method of Magic Sets. In the generalized version we are passing information through derived predicates as well. This kind of information passing is well known as *sideaway information passing (sip)* [Beeri and Ramakrishnan, 1991]. Thus, the sources of bindings are

1. the constants and
2. the bindings in the head of the rule

These bindings are propagated through the base *and* derived predicates. If we consider each distinguished argument to be bound, this defines an adornment for each derived literal on the right. The adorned rule is obtained by replacing each derived literal by its adorned version.

Example: Let the database be:

```
(rl (cylinder _name _length _radius)
    (truncone _name  _length _radius _radius))
(rl (ring _name _radius1 _radius2)
    (truncone _name 0 _radius1 _radius2))
(rl (rcone _name _length _radius)
    (truncone _name _length _radius 0))
(rl (lcone _name _length _radius)
    (truncone _name _length 0 _radius))
(rl (lshoulder (c _ring _cyl) _length _tradius _bradius)
    (ring _ring _tradius _bradius)
    (connected _ring _cyl)
    (cylinder _cyl _length _bradius)
    (> _tradius _bradius))
(rl (rspear (c _cyl _cone) _length _radius)
    (connected _cyl _cone)
```

```
        (cylinder _cyl _length1 _radius)
        (rcone _cone _length2 _radius)
        (is _length (+ _length1 _length2)))
```

and the goal

```
(rspear _a _b 2)
```

Then there is the adorned goal (query) (rspear-ffb _a _b 2) and an adorned database:

```
(rl (cylinder-ffb _name _length _radius)
    (truncone _name  _length _radius _radius))
(rl (rcone-bfb _name _length _radius)
    (truncone _name _length _radius 0))
(rl (rspear-ffb (c _cyl _cone) _length _radius)
    (cylinder-ffb  _cyl _length1 _radius)
    (connected _cyl _cone)
    (rcone-bfb _cone _length2 _radius)
    (is _length (+ _length1 _length2)))
```

Note that no adornment for the ring, lshoulder and lcone predicates are generated. This indicates that they are not needed for evaluation and therefore do not appear in the adorned database. This cuts down the number of relevant rules.

4.2.2 Generation Step

In the second step of the process the adorned program is used for the generation of the magic rules. For each of the adorned predicates of the body of an adorned rule do the following:

1. eliminate all other following predicates in the body
2. replace the predicate name p-a with magic.p-a, where a is the adornment of the predicate, and erase the nondistinguished variables
3. eliminate all predicates in the remaining body that do not contain distinguished variables occuring also in the derived magic predicate above
4. replace the rule head h-a with magic.h-a, deleting all nondistinguished variables in the transformed head
5. take the derived predicate as the head and the transformed head as the first literal of the rule's body, giving the generated magic rule

If there is no adorned predicate in the body of a rule, no magic rule will be generated. An improvement is, not going always back to the head and collecting all involved premises but only collecting the really needed premises in order to generate a safe magic rule.

Eaxmple (continued): In our example we get no magic rules for the first two nonrecursive rules, but one magic rule for each of the two adorned predicates of the third rule's body. The second generated magic rule is an improved (minimal) magic rule, in which the redundant transformed head literal is deleted.

```
(rl (magic.cylinder-ffb  _radius)
    (magic.rspear-ffb _radius))
(rl (magic.rcone-bfb _cone _radius)
    (magic.cylinder-ffb _radius)
    (connected _cyl _cone))
```

4.2.3 Modification Step

Finally an adorned rule is modified as follows: For each adorned rule with head `(h-a X)`, where the `X` denotes the list of distinguished variables, append `(magic.h-a X)` to the rule's body. In our example this is

```
(rl (cylinder-ffb _name _length _radius)
    (magic.cylinder-ffb _radius)
    (truncone _name  _length _radius _radius))
(rl (rcone-bfb _name _length _radius)
    (magic.rcone-bfb _name _radius)
    (truncone _name _length _radius 0))
(rl (rspear-ffb (c _cyl _cone) _length _radius)
    (magic.rspear-ffb _radius)
    (cylinder-ffb  _cyl _length1 _radius)
    (connected _cyl _cone)
    (rcone-bfb _cone _length2 _radius)
    (is _length (+ _length1 _length2)))
```

We also create a *seed* for the magic predicates, which is a fact, obtained from the query. The seed provides an initial value for the magic predicate corresponding to the query predicate. Here it is:

```
(hn (magic.rspear-ffb 2))
```

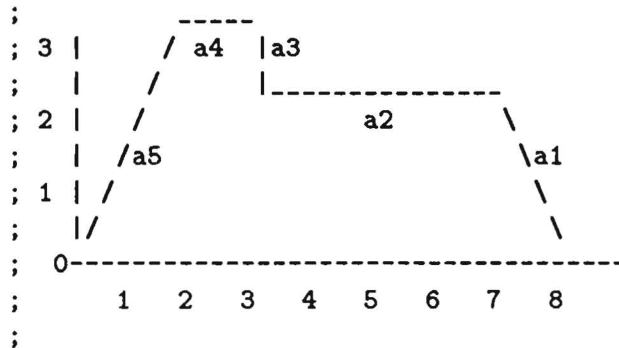
4.2.4 Fact Derivation

After transformation we have a complete magic database:

```
(hn (magic.rspear-ffb 2))
(rl (magic.cylinder-ffb  _radius)
    (magic.rspear-ffb _radius))
(rl (magic.rcone-bfb _cone _radius)
    (magic.cylinder-ffb _radius)
    (connected _cyl _cone))
```

```
(rl (cylinder-ffb _name _length _radius)
    (magic.cylinder-ffb _radius)
    (truncone _name  _length _radius _radius))
(rl (rcone-bfb _name _length _radius)
    (magic.rcone-bfb _name _radius)
    (truncone _name _length _radius 0))
(rl (rspear-ffb (c _cyl _cone) _length _radius)
    (magic.rspear-ffb _radius)
    (cylinder-ffb  _cyl _length1 _radius)
    (connected _cyl _cone)
    (rcone-bfb _cone _length2 _radius)
    (is _length (+ _length1 _length2)))
```

All predicates that are not adorned or builtin predicates are nonrecursive predicates. These predicates are collected in a special fact database:



```
;
;            -----
; 3 |       / a4  |a3
;   |     /       |---------------
; 2 |    /              a2       \
;   |  /a5                        \a1
; 1 | /                            \
;   |/                              \
;  0-----------------------------------
;      1   2   3   4   5   6   7   8
;
```

```
(hn (truncone a1 1 2 0))
(hn (truncone a2 4 2 2))
(hn (truncone a3 0 3 2))
(hn (truncone a4 1 3 3))
(hn (truncone a5 2 0 3))
(hn (connected a5 a4))
(hn (connected a4 a3))
(hn (connected a3 a2))
(hn (connected a2 a1))
```

During bottom-up evaluation of our magic database with these facts we have at the beginning, in a 0th step, only the magic seed:

```
0 (magic.rspear-ffb 2)
```

In a first step we can derive only:

```
1 (magic.cylinder-ffb 2)
```

Two facts are derived in a second step:

```
2 (magic.rcone-bfb a1 2)
  (cylinder-ffb a2 5 2)
```

In the third step we get one fact:

```
3 (rcone-bfb a1 1 2)
```

Also in the fourth step:

```
4 (rspear-ffb (c a2 a1) 5 2)
```

After the fifth step, when no new facts are derived, the bottom-up evalution stops with five derived facts in this case.

Otherwise, applying only Semi-Naive bottom-up evaluation without Magic Sets transformatoin we would get:

```
(ring a3)
(cylinder a2)
(cylinder a4)
(lcone a5 2 3)
(rcone a1 1 2)
(rspear (c a2 a1) 5 2)
(lspear (c a5 a4) 3 3)
(lshoulder (c a3 a2) 4 3 2)
```

Here the number of new facts is eight and thereby greater than the number of facts derived by applying Magic Sets transformation, although the used knowledge base is very small. The gap between these two evaluation methods increases with the number of rules and facts of the database.

4.3 Rule Compilation

An Abstract Machine for efficient execution of this set-oriented forward reasoning strategy has been implemented (FAM - Forward Abstract Machine [Falter, 1992]). It is developed from the RETE pattern match algorithm with some special features. In particular, it can access the RELFUN's relational-functional machine to evaluate top-down premises.

Executing the command

```
fwi> rule-compile
```

will compile all bottom-up rules and magic rules into the FAM. After switching to emulator mode, evaluation of the rules either by applying the **eval** or the **magic-query** command will force the execution to be performed by the FAM.

Chapter 5

Tuple-oriented bottom-up Reasoning

The tuple-oriented forward reasoning approach is implemented by a horizontal transformation of the Horn rules to backward reasoning Horn clauses of RELFUN, thus performing forward reasoning in a backward reasoning system [Hinkelmann, 1991a]. While the set-oriented forward reasoning approach described in Section 4.1 computes all the consequences of the whole knowledge base by a fixpoint operation, the objective of the tuple-oriented approach presented in this chapter is to compute *only the derivations of an explicitly given set of facts.* Another difference to the set-oriented approach of Section 4.1 is that the premises of the rules are proved by SLD-resolution instead of a simple look-up in the fact base.

The calling convention for forward evaluation is

(*<strat_name>*{*<fact>*|*<list_of_facts>*}*<inference_pattern>*)

with

<strat_name> the name of the control strategy. The predefined strategies are described in Section 5.2

<fact> a passive RELFUN-structure, representing the initial fact

<list_of_facts> a structure of the form (tup *<fact*$_1$*>* ... *<fact*$_m$*>*)

<inference_pattern> is either a variable, or a passive RELFUN-structure

As the result the consequences of the initial facts which are unifiable with *<inference_pattern>* are returned. This means that there is a derivation path from at least one of the initial facts to the result facts.

There is a command `rf-query` to evaluate queries in the RELFUN subsystem, which do have forward chaining expressions as (sub)goals. For example:

```
fwi> rf-query (bf-enum '(parent john peter) _res)
```

will enumerate all the consequences of the fact `(parent john peter)`, which are then bound to `_res`.

5.1 Transformation of Rules

The horizontal compilation presented in this chapter takes a set of Horn rules $\mathcal{P} = \{C_1, \ldots, C_n\}$ and produces a set of RELFUN clauses $\mathcal{FP} = \{C'_1, \ldots, C'_m\}$, which are the corresponding clauses of $\mathcal{P}$ for forward reasoning (see below). This transformation is equivalent to the partial evaluation of a forward reasoning meta-interpreter as described in [Hinkelmann, 1991b]. To activate this horizontal compilation there is a command

```
fwi> fw-transform
```

Every rule

({rl | up} A B_1 ... B_m)

is translated into a *sequence* of forward RELFUN clauses following this pattern:

(hn (forward B_1 A) B_2 ... B_m (retain 'A))
(hn (forward B_2 A) B_1 B_3 ... B_m (retain 'A))
...
(hn (forward B_m A) B_1 ... B_{m-1} (retain 'A)

The clauses have the following intended semantics:

> "If the actual fact is unifiable with B_i with most general unifier σ, then prove the remaining premises $B_1\sigma, \ldots, B_{i-1}\sigma, B_{i+1}\sigma, \ldots, B_m\sigma$. If they are satisfied giving substitution $\tau \geq \sigma$, *retain* the conclusion $A\tau$ for further reasoning."

A goal (forward B_i A) succeeds, if A is a one-step derivation of B_i. Thus, applying a forward clause corresponds to a one-step forward execution of the original Horn rule, triggered by B_i. retain is a built-in operator asserting the derived fact if it has not already been derived in a previous step.

Because forward evaluation of a Horn clause can be triggered by a fact unifying any premise of the clause, for every premise $B_1, \ldots, B_m$ of the original clause a forward clause is generated. This is an important difference to Yamamoto's and Tanaka's translation for production rules [Yamamoto and Tanaka, 1986], where only goal-directed forward reasoning is supported.

The command

```
fwi> list-forward
```

shows all these forward clauses on the screen. An optional argument is a pattern which unifies the head of the listed clause (see Section A).

The transformation of rules with more than one conclusion into Horn rules is called "hornification". The tuple-oriented bottom-up reasoning handles the multiple conclusions directly, so no hornification of these rules is necessary.
A clause

({rl | up} $A_1 \ldots A_n$<-$B_1 \ldots B_m$)

is transformed to the *sequence* of forward clauses

(hn (forward B_1 _conc) $B_2 \ldots B_m$ (member _conc (tup $A_1 \ldots A_n$)) (retain _conc))
...
(hn (forward B_m _conc) $B_1 \ldots B_{m-1}$ (member _conc (tup $A_1 \ldots A_n$)) (retain _conc))

The advantage of this transformation is that multiple proves of the premises are avoided. To use the bidirectional rules as backward rules, hornification is necessary.

Example: The given rules

```
(rl (p _x) (q _x _y)  <-  (r1 _x _z) (r2 _z _y))
```

are horizontally transformed into the forward clauses

```
(hn  (forward (r1 _x _z) _conc)
     (r2 _z _y)
     (member _conc (tup (p _x) (q _x _y)))
     (retain _conc))
(hn  (forward (r2 _z _y) _conc)
     (r1 _x _z)
     (member _conc (tup (p _x) (q _x _y)))
     (retain _conc))
```

and splitted into the hornified RELFUN clauses

```
(hn  (p _x _y)
     (r1 _x _z)
     (r2 _z _y)
(hn  (q _x _z)
     (r1 _x _z)
     (r2 _z _y)
```

5.2 Predefined Control Strategies of Forward Reasoning

There is an explicit control strategy necessary to derive all the possible deductions of one or more facts which calls the corresponding forward rules and administer the results. To keep the solutions consistent in a given knowledge base, the triggers used within the forward reasoning must be correct in this knowledge base. So the strategy has to prove first the triggers and then to continue with all the proved facts.

Each forward clause of Section 5.1 corresponds to the application of one rule in forward direction. To control the application of these clauses two basic control strategies for the forward-chaining system are already defined:

depth-first reasoning: the deduction continues with the *most recently derived* fact, for which there are applicable rules.

breadth-first reasoning: this algorithm *first* derives *all* the possible one-step consequences of each fact, before it triggers rules with a new one.

Both strategies have been implemented to enumerate their solutions and to return the derivations all at once. The strategy (**bf-enum** *<fact>* *<inference_pattern>*) enumerates all the consequences of *<fact>* using breadth-first search, unifying the result with *<inference_pattern>*. The strategie **df-enum** is similar, but uses depth-first search.

The strategies df-all and bf-all are functions, which have as values the list of all consequences of the initial facts,which are unifiable with *<inference_pattern>* as values. The *<inference_pattern>* remains unbound.

Example: depth-first strategy, enumerating the consequences of a list of facts

```
(ft (df-enum (tup _Fact | _Rest) _Inference)
    (fc-initialize)
    (satisfied '(tup _Fact | _Rest))
    (df-elist '(tup _Fact | _Rest) _Inference))
(ft (df-enum _x _y)
    (reset-retain)
    unknown)

(ft (df-elist (tup _Fact | _Rest) _Inference)
    (df-one _Fact _Inference))
(ft (df-elist (tup _First | _Rest) _Inference)
    (df-elist _Rest _Inference))

(ft (df-one _Fact _Inference)
    (forward _Fact _Conclusion)
    (df-one-more _Conclusion _Inference))

(ft (df-one-more _Conclusion _Conclusion) _Conclusion)
(ft (df-one-more _Conclusion _Next)
    (df-one _Conclusion _Next))

(hn (satisfied (tup)))
(hn (satisfied (tup _Fact | _Rest))
    _Fact
    (satisfied _Rest))

(hn (nou _x _x) ! unknown)
(hn (nou _x _y))
```

Example: Given the rules

```
(rl (ancestor _X _Y) (parent _X _Y))
(rl (ancestor _X _Y) (parent _X _Z) (ancestor _Z _Y))
```

and the following list of facts

```
(hn (parent s1 s2))
(hn (parent s2 s3))
(hn (parent s3 s4))
(hn (parent s4 s5))
(hn (parent s5 s6))
```

asking the query

```
fwi> rf-query (df-enum (parent s3 _X) _Result)
```

will compute all the ancestors of s3:

```
  _Result = (ancestor s3 s4)
fwi> more
  _Result = (ancestor s3 s5)
fwi> more
  _Result = (ancestor s3 s6)
fwi> more
 unknown
```

Because the strategies are specified in the high-level RELFUN language, it is possible for the user of the system to define his own reasoning strategies. With the command

fwi> replace-strategies <*filename*>

the predefined strategies are destroyed and the clauses in file <*filename*> are consulted into the strategy database. The actually loaded strategies are displayed when typing the command

```
fwi> list-strategies
```

5.3 Retaining Derived Facts

Derived facts in horizontally compiled forward rules could be retained by assertion [Hinkelmann, 1991a]. Such assertions are rather inefficient because program code itself is altered dynamically. Information about derived facts can be held more compactly at machine level.

To record the derived facts there is an extra database organized as a stack wich is called *retain stack*. Some predefined predicates and built-in operators are responsible for the insertion and access of derived facts. The operator `retain` which occurs as last premise in every forward clause (see Section 5.1) has to push the new derived facts on the retain stack. To accept a derived fact, it must be ensured that it is not subsumed by any structure already existing on the stack. This subsumption test is done by a special machine [Oltzen, 1992]. If this subsumption test fails, the derived fact is pushed onto the retain stack.

The `retain` predicate is defined as:

- `(retain _fact)`: The functionality is described above. The basic implementation of this operator is:

```
(ft  (retain _fact)
     (not-r-subsumed _fact)
     (push-fact-retain _fact)
     _fact)
```

The predicates `not-r-subsumed` and `push-fact-retain` are built-in operators:

- `(not-r-subsumed _fact)`: Tests whether `_fact` is subsumed by any element of the *retain stack* or not. Subsumption is a "*one-side*" unification:

  > "Let $(p_1 \ldots)$ and $(p_2 \ldots)$ be terms then $(p_1 \ldots)$ subsumes $(p_2 \ldots)$ if there exists a substitution σ such that $(p_1 \ldots)\sigma = (p_2 \ldots)$. "

- `(push-fact-retain _fact)`: Copies the whole structure `_fact` on top of the retain stack so that there are no references into the heap.

To retrieve facts from the retain stack there are two possibilities: retrieving the most recently derived fact (called actual node) and retrieving a fact as a trigger for a forward clause. Facts that have not been selected as a trigger for a forward clause are called open nodes.

- `(get-actual-node)`: This built-in function returns the most recently derived fact from the retain stack.
- `(actual-node _inference)`: The access procedure to the most recently derived fact.

```
(hn  (actual-node _inference)
     (is _inference (get-actual-node)))
```

The following primitives to access open nodes are specialized for breadth-first reasoning. Since depth-first reasoning corresponds to the underlying RELFUN system, no extra administration of open nodes is necessary.

- `(get-open-node)`: Returns the first open node. An open node is an entry on the retain stack which has not been used to trigger a forward rule.
- `(next-open-node)`: For breadth-first reasoning this function simply sets the open node to the next entry on the retain stack following the last open node. This instruction fails if there is none. For more sophisticated search strategies like best-first search this function has to be redefined.
- `(not-open-node-at-end)`: Test whether there are any open nodes left on the retain stack.
- `(open-node _inference)`: The access procedure to the last open node. The basic version maps simply to:

```
(hn  (open-node _Fact)
     (next-open-node)
     (is _Fact (get-open-node)))
(hn  (open-node _Fact)
     (not-open-node-at-end)
     (open-node _Fact))
```

The following two operators are neccessary to handle recursive calls of forward chaining.

- `(fc-initialize)`: Initializes a new retain stack.
- `(reset-retain)`: Removes the last retain stack.

5.4 Compilation into the RFM

After source-to-source transformation of Horn rules $\mathcal{P}$ into a forward clause program $\mathcal{FP}$, the clauses of $\mathcal{FP}$ can be compiled into RFM code by using the command

```
fwi> fw-compile
```

In emulator mode a query

fwe> rf-query <*query*>

will be evaluated in the RFM.

The RFM (Relational Functional Machine), a variation of Warren's Abstract Machine (WAM) for RELFUN, is the target for compiling RELFUN code. A direct compilation of $\mathcal{FP}$ would be rather inefficient, because all clauses of $\mathcal{FP}$ have the same predicate `forward`; this means that there is one large procedure with with costly search for an applicable rule.

To overcome this deficiency and to handle the retain instruction as described above (Section 5.3) some modifications of the RFM are suggested: first, the code area is split into the usual backward code area and a new special forward code area for `forward` clauses; second, a new stack for derived facts, called *retain stack*, is introduced (see Fig. 5.1).

5.4.1 Forward Code Area

The clauses obtained by horizontal transformation have one fundamental drawback: they are represented with a single predicate `forward`. After compilation there is one large procedure for all the forward clauses. Access is just supported by indexing on the first argument's functor. A special code area for forward clauses can make this `forward` predicate implicit and clauses with the same trigger can be grouped together into one procedure. The principle of this special forward code can be explained as follows. A forward clause

(hn (`forward` B_1 A) $B_2 \ldots B_m$) (`retain` 'A))

is applied, if the actual fact is unifiable with B_1. Then the remaining premises $B_2 \ldots B_m$ are tested. If they are satisfied the conclusion A is retained. This can be achieved in principle by the following clause:

(hn B_1 $B_2 \ldots B_m$ (`retain` 'A))

The head B_1 is the trigger of the forward clause. Please note that this clause is not applied to prove B_1 but to derive A, if B_1 is already known. The advantage of this approach is that the single forward procedure is decomposed into one procedure for each trigger predicate. By applying the indexing techniques of the RFM [Sintek and Stein, 1992], the applicable clauses can be found efficiently.

The access to the forward code area is done with the RFM builtin `forward` which has the same semantics as the interpreted `forward` clause so that no difference between the interpreted and compiled program occurs.

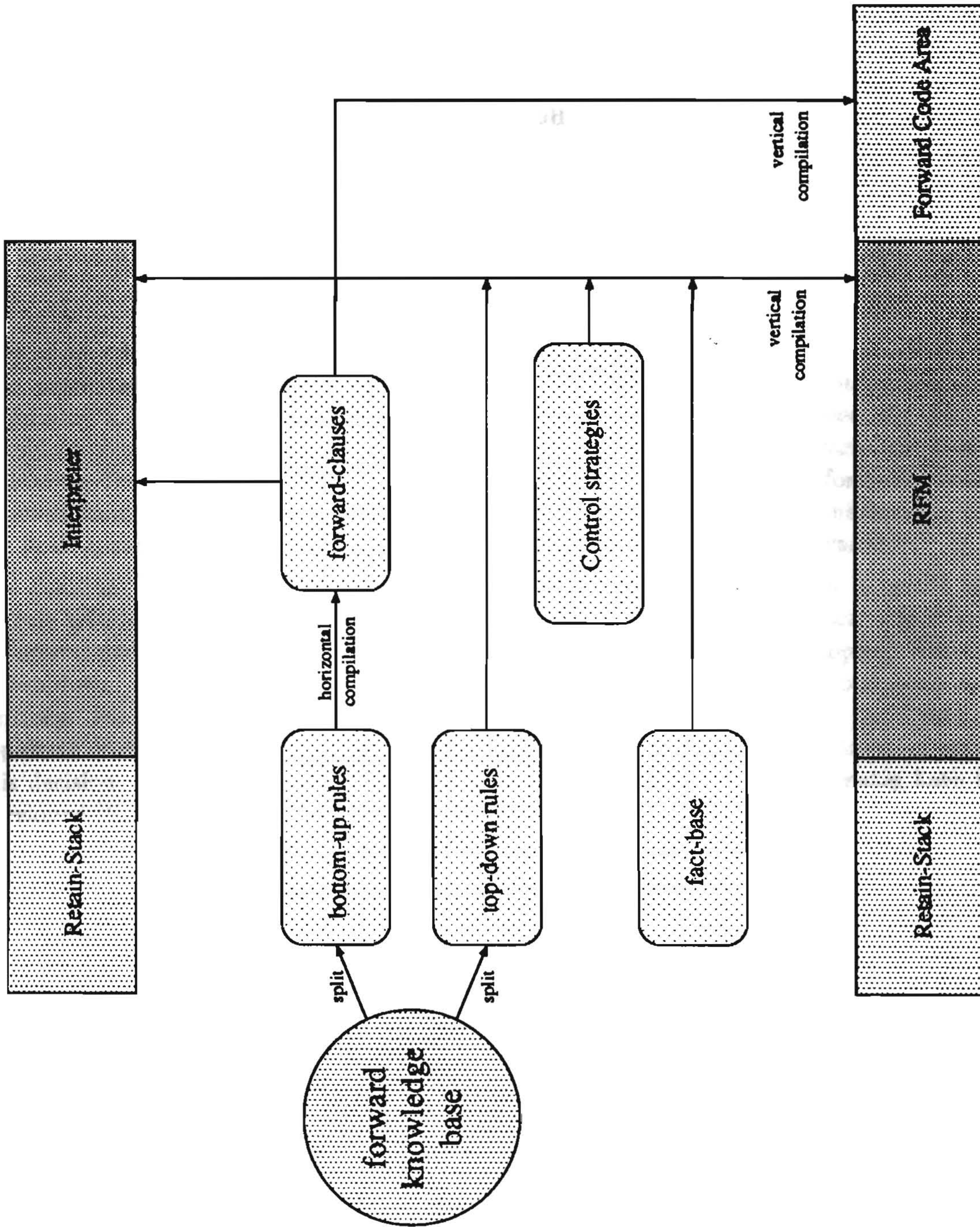


Figure 5.1: Forward Reasoning Architecture of the Tuple-oriented Approach

5.4.2 Retain Stack

As described in Section 5.3 derived clauses are recorded in a special retain stack. This stack is also necessary in the RFM to avoid assertions into the code area. The built-in operators to manage the retain stack are also available at RFM level.

The values on the retain stack are more persistant than values on the stack or the heap. The retain stack will not be changed by backtracking, because the environment does not contain any information about the retain stack. But, while values on stack and heap are destroyed by backtracking, no reference from the retain stack to any other memory cell is permitted. This is why a derived fact is *copied* onto the retain stack.

Chapter 6

Conclusion

Two approaches for integrating bottom-up and top-down reasoning of logic programs have been presented. The first, set-oriented approach views the bottom-up reasoning component as primary and provides access to the top-down system for testing premises of rules. The second, tuple-oriented approach implements bottom-up reasoning by rule transformation and evaluating the resulting clauses in a top-down manner. The rule transformation can be achieved from partial evaluation of a meta-interpreter as presented in [Hinkelmann, 1992]. For both approaches interpretative and compilative implementations are available.

The decision whether rules should be applied bottom-up or top-down can be made either on rule level or on strategy level. A rule-level decision means that for each rule it has to be determined before the reasoning process starts, whether it should be applied bottom-up or top-down. On the other hand, the application direction of a rule can also depend on the strategy. For instance, the tuple-oriented bottom-up approach applies the same rules also top-down to test premises of triggered rules. A further goal of the project is to automatize this decision. As a first step several criteria have been set up and cost estimates have been developed [Hintze, 1992].

The rule component is part of the declarative knowledge representation system COLAB. The relational-functional language RELFUN proves the top-down queries of the rule component and is the basis run-time system for the tuple-oriented bottom-up reasoning approach. This interface also gives access to the functional part of RELFUN. The terminological reasoning system TAXON allows to establish taxonomies of strutured terms, which can be accessed during rule application. This integration has been prototypically realized for the tuple-oriented approach. Currently it is going to be extended also for the set-oriented bottom-up reasoning. A combination between terminological and rule-based reasoning for abstraction processes is described in [Hanschke and Hinkelmann, 1992]. Possible interactions between the constraint system CONTAX and the rule system still have to be explored.

Appendix A

Commands for the Rule Component of COLAB

The bottom-up reasoning component is included in the CoLab system. When CoLab has been loaded into Lisp, calling the function `(colab)` starts the CoLab toplevel shell, which is organized as a read-eval-print loop. A number of commands are available, which are listed by typing a question mark. A special group of commands are the CoLab system commands **`contax`**, **`forward`**, **`relfun`**, and **`taxon`**, which switch to the corresponding component. Each such subshell offers – besides the general CoLab commands – specific commands for querying, listing and managing knowledge items. In the following the commands of the bottom-up subshell `forward` will be described.

az:

Format: **`az`** `<clause>`

Options: <clause> a COLAB clause

Effect: The rule <clause> will be inserted at the end of one of the (possibly empty) FORWARD databases **`*rule-database*`** (containing the bidirectional rules with tag `rl` and the forward rules with tag up) or **`*fact-base*`** (containing only facts) where the tags **`fact`** and **`attrterm`** are substituted by the tag **hn** in order to provide this database for the RELFUN-Interpeter.

see also: **consult, destroy, replace**

a0:

Format: `a0 <clause>`

Options: <clause> a COLAB-clause

Effect: The rule <clause> will be inserted in front of one of the (possibly empty) FORWARD databases **`*rule-database*`** (cointaining the bidirectional rules with tag `rl` and the forward rules with tag up) or **`*fact-base*`** (containing only facts) where the tags **`fact`** and **`attrterm`** are substituted by the tag **hn** in order to provide this database for the RELFUN-Interpeter.

see also: **consult, destroy, replace**

compile-strategies:

Format: **`compile-strategies`**

Options: none

Effect: The actual loaded strategies are vertically compiled.

see also: **consult-strategies, replace-strategies, fw-compile**

consult:

Format: `consult <filename>`

Options: <filename> a stringified or normal pathname

Effect: Global consulting function. Loading a hybrid database from file <filename> at the end of the (possibly empty) databases in COLAB depending on the tags of the knowledge items of the file. In FORWARD there are `*rule-database*` and `*factbase*`. If no extension is provided, COLAB extends the filename with `".rf"`.

see also: **consult-facts, consult-rules, consult-strategies, destroy, replace**

consult-facts:

Format: `consult-facts <filename>`

Options: <filename> a stringified or normal pathname

Effect: Loading all the facts of a hybrid database from file <filename> at the end of the (possibly empty) database `*factbase*`. If no extension is provided, COLAB extends the filename with `".rf"`.

see also: **consult, destroy, replace**

consult-rules:

Format: `consult-rules <filename>`

Options: <filename> a stringified or normal pathname

Effect: Loading all the rl- and up- rules of a hybrid database from file <filename> at the end of the (possibly empty) database `*rule-database*`. If no extension is provided, COLAB extends the filename with `".rf"`.

see also: **consult, destroy, replace**

consult-strategies:

Format: `consult-strategies <filename>`

Options: <filename> a stringified or normal pathname

Effect: Loading strategies for tuple-oriented forward chaining from file <filename> at the end of the database `*fc-strategies*`. If no extension is provided, COLAB extends the filename with `".rf"`.

see also: **consult, destroy, replace**

destroy:

Format: `destroy`

Options: none

Effect: Global destroying function. Destroy all existing COLAB databases.

see also: **consult, destroy-facts, destroy-magic, destroy-rules, replace**

destroy-facts:

Format: `destroy-facts`

Options: none

Effect: Destroy the databases `*factbase*` and `*derived-factbase*`.

see also: **consult, destroy, replace**

destroy-magic:

Format: `destroy-magic`

Options: none

Effect: Destroy the magic databases `*magic-rules*` and `*magic-seeds*`.

see also: **destroy, magic-transform**

destroy-rules:

Format: `destroy-rules`

Options: none

Effect: Destroy all databases with rules in FORWARD. Their names are `*rule-database*`, `*up-rulebase*`, `*hn-rulebase*`, `*forward-rules*` and `*magic-rules*`.

see also: **consult, destroy, replace**

eval:

Format: `eval`

Options: none

Effect: Semi-naive bottom-up evaluation is performed on the databases `*up-rulebase*` and `*factbase*`. The derived facts are added to the database `*derived-factbase*`.

see also: **magic-eval, magic-query**

fw-compile:

Format: `fw-compile` <op>

Options: <op> an operator

Effect: If an operator is given only the matching rules of `*rule-database*` are regarded otherwise all rules of `*rule-database*` are splitted. The resulting up-rules will be horizontally and vertically compiled into the extended WAM. The resulting horn-rules and the `*factbase*` are vertically compiled.

see also: **fw-compile-facts, fw-compile-rules,fw-transform, hornify-up.**

fw-compile-facts:

Format: `fw-compile-facts` <op>

Options: <op> an operator

Effect: If an operator is given only the matching facts of `*factbase*` are regarded otherwise all facts of `*factbase*` are vertically compiled into the extended WAM.

see also: **fw-compile, fw-compile-rules**

fw-compile-rules:

Format: `fw-compile-rules` <op>

Options: <op> an operator

Effect: If an operator is given only the matching rules of `*rule-database*` are regarded otherwise all rules of `*rule-database*` are splitted. The resulting up-rules will be horizontally and vertically compiled into the extended WAM. The resulting horn-rules are vertically compiled.

see also: **fw-compile, fw-compile-facts,fw-transform, hornify**.

fw-emul:

Format: `fw-emul`

Options: none

Effect: With this command you are entering the emulator mode of FORWARD and the prompt changes to `fwe>`.

see also: **fw-inter**

fw-inter:

Format: `fw-inter`

Options: none

Effect: With this command you are leaving the emulator mode of FORWARD, you return to the interpreter mode and the prompt changes to `fwi>`.

see also: **fw-emul**

fw-transform:

Format: `fw-transform`

Options: none.

Effect: All rules of the database `*up-rulebase*` will be horizontally compiled into the database `*forward-rules*`.

see also: **fw-compile, fw-compile-facts, fw-compile-rules**

hornify-up:

Format: `hornify-up`

Options: none.

Effect: All rules of `*up-rulebase*` will be hornified. Hornified means that a rule with more than one conclusion is splitted into several rules, each with one conclusion.

see also: **fw-compile, hornify**

l:

Format: `l`
`l` <op>
`l` <pat>

Options: <op> an operator,
<pat> a head pattern

Effect: Shows the knowledge items of the databases `*rule-database*`, `*factbase*` and `*derived-factbase*`. If no argument is given, the whole databases will be printed on the terminal. If an operator <op> is given, only those knowledge items in the databases will be printed which use this operator as their procedure name. If a pattern is given, only those knowledge items in the databases will be printed whose head matches the pattern <pat>.

see also: **list-facts**, **list-forward**, **list-magic**, **list-rules**, **list-strategies**, **consult**

list-facts:

Format: `list-facts`
`list-facts <op>`
`list-facts <pat>`

Options: <op> an operator,
<pat> a head pattern

Effect: Shows the knowledge items of the databases `*factbase*` and `*derived-factbase*`. If no argument is given, both databases will be printed on the terminal. If an operator <op> is given, only those knowledge items in the database will be printed which use this operator as their procedure name. If a pattern is given, only those knowledge items in the database will be printed whose head matches the pattern <pat>.

see also: **consult**, **consult-facts**, **listing**

list-forward:

Format: `list-forward`
`list-forward <pat>`

Options: <pat> a head pattern

Effect: Shows the knowledge items of the database `*forward-rules*`. If no argument is given, the whole database will be printed on the terminal. If a pattern is given, only those knowledge items in the database will be printed whose head matches the pattern <pat>.

see also: **consult**, **fw-transform**

listing:

Format: `listing`
`listing <op>`
`listing <pat>`

Options: <op> an operator,
<pat> a head pattern

Effect: Global listing function. Shows the knowledge items of all databases in COLAB. If no argument is given, the all databases will be printed on the terminal. If an operator <op> is given, only those knowledge items in the databases will be printed which use this operator as their procedure name. If a pattern is given, only those knowledge items in the databases will be printed whose head matches the pattern <pat>.

see also: **l**, **list-facts**, **list-forward**, **list-magic**, **list-rules**, **list-strategies**, **consult**

list-magic:

Format: `list-magic`
`list-magic <op>`
`list-magic <pat>`

Options: <op> an operator,
<pat> a head pattern

Effect: Shows the knowledge items of the databases `*magic-rules*` and `*magic-seeds*`. If no argument is given, both databases will be printed on the terminal. If an operator <op> is given, only those knowledge items in the databases will be printed which use this operator as their procedure name. If a pattern is given, only those knowledge items in the databases will be printed whose head matches the pattern <pat>.

see also: **consult, listing, magic-eval, magic-transform**

list-rules:

Format: `list-rules`
`list-rules <op>`
`list-rules <pat>`

Options: <op> an operator,
<pat> a head pattern

Effect: Shows the knowledge items of the database `*rule-database*`. If no argument is given, the whole database will be printed on the terminal. If an operator <op> is given, only those knowledge items in the database will be printed which use this operator as their procedure name. If a pattern is given, only those knowledge items in the database will be printed whose head matches the pattern <pat>.

see also: **consult, consult-rules, listing**

list-strategies:

Format: `list-strategies`
`list-strategies <op>`
`list-strategies <pat>`

Options: <op> an operator,
<pat> a head pattern

Effect: Shows the knowledge items of the databases `*fc-strategies*`. If no argument is given, the whole database will be printed on the terminal. If an operator <op> is given, only those knowledge items in the database will be printed which use this operator as their procedure name. If a pattern is given, only those knowledge items in the database will be printed whose head matches the pattern <pat>.

see also: **consult-strategies**

magic-eval:

Format: `magic-eval <goal1 > ... <goaln >`

Options: <goal1 > ... <goal$n$ > the initial seeds for magic sets evaluation.

Effect: First magic sets transformation and then semi-naive evaluation is performed wrt to the given goals. This command is an abbreviation of the sequence of commands **magic-transform, magic-query**, where the arguments of both commands are the same.

see also: **eval, magic-query, magic-transform**

magic-query:

Format: `magic-query` <goal1 > ... <goal*n* >

Options: <goal1 > ... <goal*n* > the initial seeds for magic sets evaluation.

Effect: The databases `*magic-rules*`, `*factbase*` and `*derived-factbase*` will be evaluated wrt the given goals using the semi-naive bottom-up strategy. If no goals are given, the database `*magic-seeds*` (all seeds of transformation) is taken for evaluation.

see also: **eval, magic-eval, magic-transform**

magic-transform:

Format: `magic-transform` <goal1 > ... <goal*n* >

Options: <goal1 > ... <goal*n* > the initial seeds for magic sets transformation.

Effect: The database `*up-rulebase*` will be used to generate new rules wrt to the given goals applying the Magic Sets Transformation method. The new rules are stored in the database `*magic-rules*` and the transformed seeds in `*magic-seeds*`.

see also: **eval, magic-eval, magic-query**

nospy:

Format: nospy

Options: none

Effect: With this FORWARD command you leave the trace mode.

see also: **spy**

replace:

Format: `replace <filename>`

Options: <filename> a stringified or normal pathname

Effect: Global replacing function. Replacing the (possibly empty) databases in COLAB depending on the tags of the knowledge items of the file with the contents of the file <filename>. In FORWARD there are the `*rule-database*` and `*factbase*`. If no extension is provided, COLAB extends the filename with `".rf"`.

see also: **consult, destroy, replace-facts, replace-rules, replace-strategies**

replace-facts:

Format: `replace-facts <filename>`

Options: <filename> a stringified or normal pathname

Effect: Replacing the (possibly empty) database `*factbase*` with the facts of the file <filename> and destroying the database `*derived-factbase*`. If no extension is provided, COLAB extends the filename with `".rf"`.

see also: **consult, destroy, replace**

replace-rules:

Format: `replace-rules <filename>`

Options: <filename> a stringified or normal pathname

Effect: Replacing the (possibly empty) database `*rule-database*` in COLAB depending on the tags of the rules with the rl- and up-rules of a hybrid database of file <filename>. If no extension is provided, COLAB extends the filename with `".rf"`.

see also: **consult, destroy, replace**

replace-strategies:

Format: `replace-strategies <filename>`

Options: <filename> a stringified or normal pathname

Effect: Replacing the database `*fc-strategies*` with the strategies for forward chaining from file <filename>. If no argument is given the default strategies are consulted. If no extension is provided, COLAB extends the filename with `".rf"`.

see also: **consult, destroy, replace**

rf-query:

Format: `rf-query <query>`

Options: <query> a query to RELFUN

Effect: The specified <query> is evaluated either by thr RELFUN-interpreter or emulator depending on the operating mode in FORWARD. In interpreter mode you are able to retrieve alternative solutions by typing the command `more` directly when the answer of the query is displayed, except the system indicates that there are no more solutions by showing the item `unknown`.

see also: **magic-query**

rule-compile:

Format: `rule-compile`

Options: none

Effect: Compiles the magic rules (`*magic-rules*`,`*magic-seeds*`), the bottom-up rules (`*up-rulbase*`) and the facts (`*factbase*`) into the FAM.

see also: **eval, magic-query, magic-eval, magic-transform,**

rx:

Format: `rx <clause>`

Options: <clause> a COLAB knowledge item

Effect: The <clause> will be removed from `*rule-database*`, `*up-rulebase*`, `*hn-rulebase*` or `*factbase*` depending on its tag.

see also: **consult, destroy, replace**

split-rules:

Format: **split-rules**

Options: none

Effect: All up-clauses of the database *rule-database* are copied to the *up-rulebase*, and all rl-clauses are copied to the database *hn-rulebase* by substituting the rl-tags to hn-tags in order to provide this database for the RELFUN-Interpreter.

see also: **hornify-up**

spy:

Format: **spy**

Options: none

Effect: This activates the tracer of the FORWARD system. After entering the trace mode for semi-naive bottom-up evaluation first the indexing of the facts and rules is shown. Then all new derived facts are shown at each step until no more facts can be derived. For Magic Sets transformation also the adorned database, the magic rules and magic seeds are shown to the user.

see also: **eval, magic-eval, magic-query, magic-transform, nospy**

Bibliography

[Balbin and Ramamohanarao, 1987] I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Programming*, 4:259–262, 1987.

[Bancilhon and Ramakrishnan, 1986] Francois Bancilhon and Raghu Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the ACM SIGMOD Conference*, pages 16–52. ACM, 1986.

[Beeri and Ramakrishnan, 1991] Catriel Beeri and Raghu Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10:255–299, October 1991.

[Boley *et al.*, 1991a] H. Boley, P. Hanschke, K. Hinkelmann, and M. Meyer. COLAB: A Hybrid Knowledge Compilation Laboratory. Presented at 3rd International Workshop on Data, Expert Knowledge and Decisions: Using Knowledge to Transform Data into Information for Decision Support, Reisensburg, Germany, September 1991.

[Boley *et al.*, 1991b] Harold Boley, Klaus Elsbernd, Hans-Guenther Hein, and Thomas Krause. RFM Manual: Compiling RELFUN into the Relational/Functional Machine. Document D-91-03, DFKI GmbH, 1991.

[Boley, 1990] Harold Boley. A Relational/Functional Language and Its Compilation into the WAM. SEKI Report SR-90-05, Universität Kaiserslautern, Fachbereich Informatik, April 1990.

[Brachman *et al.*, 1983] Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. KRYPTON: A functional approach to knowledge representation. *IEEE Computer*, 16(10):63–73, October 1983.

[Falter, 1992] Christian Falter. Compilation von Vorwärtsregeln in einer hybriden Expertensystem-Shell. Diplomarbeit, Universität Kaiserslautern, FB Informatik, Postfach 3049, D-6750 Kaiserslautern, 1992. In German.

[Forgy, 1981] Charles L. Forgy. *OPS5 User's Manual*. Carnegie-Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania 15213, 1981.

[Hanschke and Hinkelmann, 1992] Philipp Hanschke and Knut Hinkelmann. Combining Terminological and Rule-based Reasoning for Abstraction Processes. In *GWAI-92*. Springer-Verlag, April 1992.

[Hinkelmann, 1991a] Knut Hinkelmann. Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation. DFKI Technical Memo TM-91-02, DFKI GmbH, January 1991.

[Hinkelmann, 1991b] Knut Hinkelmann. Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter. Technical Report Technical Memo TM-91-13, DFKI GmbH, October 1991.

[Hinkelmann, 1992] Knut Hinkelmann. Forward Logic Evaluation: Compiling a Partially Evaluated Meta-interpreter into the WAM. In *Proceedings German Workshop on Artificial Intelligence, GWAI-92*. Springer-Verlag, September 1992.

[Hintze, 1992] Helge Hintze. Kriterien für die effiziente Interpretation deklarativer Regelsysteme. Diplomarbeit, Universität Kaiserslautern, FB Informatik, Postfach 3049, D-6750 Kaiserslautern, 1992. In German.

[Labisch, 1991] Thomas Labisch. Implementierung einer semi-naiven Strategie für bottom-up Evaluierung von RELFUN Hornklauseln. Projektarbeit, Universität Kaiserslautern, FB Informatik, Juni 1991.

[Naqvi and Tsur, 1989] Shamim Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, Rockville, Maryland USA, 1989.

[Oltzen, 1992] Thomas Oltzen. Term Subsumtion in der WAM. Projektarbeit, 1992. In German.

[Sintek and Stein, 1992] Michael Sintek and Werner Stein. A Generalized Intelligent Indexing Method. Workshop "Sprachen für KI-Anwendungen, Konzepte - Methoden - Implementierungen" in Bad Honnef, May 1992.

[Ullman, 1989] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, Rockville, Maryland USA, 1989.

[Warren, 1983] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

[Yamamoto and Tanaka, 1986] Akira Yamamoto and Hozumi Tanaka. Translating Production Rules into a Forward Reasoning Prolog Program. *New Generation Computing*, 4:97–105, 1986.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
D-6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all publisched papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-91-28
Rolf Backofen, Harald Trost, Hans Uszkoreit:
Linking Typed Feature Formalisms and Terminological Knowledge Representation Languages in Natural Language Front-Ends
11 pages

RR-91-29
Hans Uszkoreit: Strategies for Adding Control Information to Declarative Grammars
17 pages

RR-91-30
Dan Flickinger, John Nerbonne:
Inheritance and Complementation: A Case Study of *Easy* Adjectives and Related Nouns
39 pages

RR-91-31
H.-U. Krieger, J. Nerbonne:
Feature-Based Inheritance Networks for Computational Lexicons
11 pages

RR-91-32
Rolf Backofen, Lutz Euler, Günther Görz:
Towards the Integration of Functions, Relations and Types in an AI Programming Language
14 pages

RR-91-33
Franz Baader, Klaus Schulz:
Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures
33 pages

RR-91-34
Bernhard Nebel, Christer Bäckström:
On the Computational Complexity of Temporal Projection and some related Problems
35 pages

RR-91-35
Winfried Graf, Wolfgang Maaß: Constraint-basierte Verarbeitung graphischen Wissens
14 Seiten

RR-92-01
Werner Nutt: Unification in Monoidal Theories is Solving Linear Equations over Semirings
57 pages

RR-92-02
Andreas Dengel, Rainer Bleisinger, Rainer Hoch, Frank Hönes, Frank Fein, Michael Malburg:
Π_{ODA}: The Paper Interface to ODA
53 pages

RR-92-03
Harold Boley:
Extended Logic-plus-Functional Programming
28 pages

RR-92-04
John Nerbonne: Feature-Based Lexicons: An Example and a Comparison to DATR
15 pages

RR-92-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner, Michael Schulte, Rainer Stark:
Feature based Integration of CAD and CAPP
19 pages

RR-92-06
Achim Schupetea: Main Topics of DAI: A Review
38 pages

RR-92-07
Michael Beetz:
Decision-theoretic Transformational Planning
22 pages

DFKI Documents

D-92-02
Wolfgang Maaß: Constraint-basierte Plazierung in multimodalen Dokumenten am Beispiel des Layout-Managers in WIP
111 Seiten

D-92-03
Wolfgan Maaß, Thomas Schiffmann, Dudung Soetopo, Winfried Graf: LAYLAB: Ein System zur automatischen Plazierung von Text-Bild-Kombinationen in multimodalen Dokumenten
41 Seiten

D-92-04
Judith Klein, Ludwig Dickmann: DiTo-Datenbank - Datendokumentation zu Verbrektion und Koordination
55 Seiten

D-92-06
Hans Werner Höper: Systematik zur Beschreibung von Werkstücken in der Terminologie der Featuresprache
392 Seiten

D-92-07
Susanne Biundo, Franz Schmalhofer (Eds.): Proceedings of the DFKI Workshop on Planning
65 pages

D-92-08
Jochen Heinsohn, Bernhard Hollunder (Eds.): DFKI Workshop on Taxonomic Reasoning Proceedings
56 pages

D-92-09
Gernod P. Laufkötter: Implementierungsmöglichkeiten der integrativen Wissensakquisitionsmethode des ARC-TEC-Projektes
86 Seiten

D-92-10
Jakob Mauss: Ein heuristisch gesteuerter Chart-Parser für attributierte Graph-Grammatiken
87 Seiten

D-92-11
Kerstin Becker: Möglichkeiten der Wissensmodellierung für technische Diagnose-Expertensysteme
92 Seiten

D-92-12
Otto Kühn, Franz Schmalhofer, Gabriele Schmidt: Integrated Knowledge Acquisition for Lathe Production Planning: a Picture Gallery (Integrierte Wissensakquisition zur Fertigungsplanung für Drehteile: eine Bildergalerie)
27 pages

D-92-13
Holger Peine: An Investigation of the Applicability of Terminological Reasoning to Application-Independent Software-Analysis
55 pages

D-92-14
Johannes Schwagereit: Integration von Graph-Grammatiken und Taxonomien zur Repräsentation von Features in CIM
98 Seiten

D-92-15
DFKI Wissenschaftlich-Technischer Jahresbericht 1991
130 Seiten

D-92-16
Judith Engelkamp (Hrsg.): Verzeichnis von Softwarekomponenten für natürlichsprachliche Systeme
189 Seiten

D-92-17
Elisabeth André, Robin Cohen, Winfried Graf, Bob Kass, Cécile Paris, Wolfgang Wahlster (Eds.): UM92: Third International Workshop on User Modeling, Proceedings
254 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-92-18
Klaus Becker: Verfahren der automatisierten Diagnose technischer Systeme
109 Seiten

D-92-19
Stefan Dittrich, Rainer Hoch: Automatische, Deskriptor-basierte Unterstützung der Dokumentanalyse zur Fokussierung und Klassifizierung von Geschäftsbriefen
107 Seiten

D-92-21
Anne Schauder: Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

D-92-25
Martin Buchheit: Klassische Kommunikations- und Koordinationsmodelle
31 Seiten

D-92-26
Enno Tolzmann:
Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX
28 Seiten

D-92-27
Martin Harm, Knut Hinkelmann, Thomas Labisch: Integrating Top-down and Bottom-up Reasoning in COLAB
40 pages

Integrating Bottom-up and Top-down Reasoning in CoLAB

Martin Harm, Knut Hinkelmann, Thomas Labisch

D-92-27
Document