Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Technical Memo
TM-91-02

Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation

Knut Hinkelmann

January 1991

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 und Saarbrücken is a non-profit organization which was founded in 1988 by the shareholder companies ADV/Orga, AEG, IBM, Insiders, Fraunhofer Gesellschaft, GMD, Krupp-Atlas, Mannesmann-Kienzle, Nixdorf, Philips 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
- 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

Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation

Knut Hinkelmann

DFKI-TM-91-02

Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation

Knut Hinkelmann
DFKI GmbH
Postfach 2080
6750 Kaiserslautern, F.R.Germany
hinkelma@dfki.uni-kl.de

Abstract

A compilative approach for forward reasoning of horn rules in Prolog is presented. Pure horn rules - given as Prolog clauses - are to be used for forward and backward reasoning. These rules are translated into Prolog clauses, denoting one forward reasoning step. Forward chaining is triggered by an initial fact, from which the consequences are derived. Premises of forward rules are verified by Prolog's backward proof procedure using the original clauses. Thus, without any changes to the Prolog interpreter integrated bidirectional reasoning of the original horn rules is possible. Breadth-first and depth-first reasoning strategies with enumeration and collection of conclusions are implemented. In order to translate forward clauses into WAM operations several improvements are introduced. To avoid inefficient changes of program code derived facts are recorded in a special storage area called retain stack. Subsumption of a new conclusion by previously derived facts is tested by a built-in procedure. As a reasonable application of this kind of forward reasoning its use is demonstrated for integrity constraint checking.

1 Introduction

Reasoning in rule-based systems can be done using two principal directions. While forward inference begins with the facts in the knowledge base reasoning bottom-up to derive new facts, backward inference applies the rules in a top-down fashion.

Conventional forward reasoning production systems [For81] have separate memories for facts (working memory) and rules (production memory). The working memory is assumed to contain all true assertions as ground facts. The rules in the production memory are only used to *modify* the working memory. Consequently the conditions of the rules in the production memory need only be tested against the facts.

Interpreting horn clauses of logic programs in the natural forward implication direction leads to the view of a logic program as a declarative rule system. The conclusion is a fact which is true if the premises are satisfied. Obviously, in the conclusion arbitrary evaluable expressions are prohibited. A characteristics of logic programs is the common representation of facts *and* rules in a single knowledge base. An uncontrolled application of forward reasoning computing all consequences of a knowledge base — if possible at all in finite time — would make all knowledge explicit as facts, leaving back the rules as redundant knowledge. This would be contrary to the philosophy of logic systems. That is why in the approach presented here

forward resoning is used only to compute the implications of a single knowledge item. By account of rule-implicit knowledge backward reasoning is still necessary to determine whether the remaining premises hold, after a rule has been triggered by a fact unifying with one of the premises.

Prolog, the most prominent logic programming language, has received much attention in the AI community. In contrast to production systems the reasoning direction of Prolog is backward (goal-directed). Because of the existence of rather efficient implementations it seems appropriate to use it as the starting point for a logic-based integration of forward and backward reasoning. Several attempts have been made to integrate forward chaining into Prolog. Common to most of these approaches is that they use disjoint sets of rules for both reasoning directions [Mor81], [CDE87], [FFM89]. On the other hand the naive or semi-naive bottom-up evaluation of horn clauses is used in deductive databases [BR86], [BR88]. There is no integration with backward reasoning but they are used to support goal-directed bottom-up reasoning with magic-sets [BMSU86] or Alexander method [RLK86]. Kowalski describes bidirectional reasoning over one rule set in pure horn logic [Kow79]. Bidirectional reasoning can be achieved by

- explicit choice of the reasoning direction using call primitives (e. g. KEE [Int86])
- dynamic choice depending on cost estimates [TG87]
- predefined interfaces, e. g. verifying conditions in forward rules by backward reasoning, or triggering forward rules by successful backward proofs.

We will introduce a compilative approach to perform, besides the usual backward chaining, forward reasoning over Prolog clauses. Forward reasoning has to be explicitly activated while premises in forward chaining are verified by Prolog's backward proof procedure.

While a (meta) rule interpreter would read and execute a rule at execution time, rule translation generates code more adequate for execution. Two knowledge compilation techniques for doing forward reasoning will be presented: *horizontal compilation* and *vertical compilation.* The attributes 'horizontal' and 'vertical' refer to the abstraction levels of source and target language. A compilation will be called 'horizontal', if source and target language are merely at the same level. Vertical compilation is more close to conventional compilation of higher level programming languages to machine code.

Our approach consists of two steps (Fig. 1). First, we will show a 'horizontal' transformation of a Prolog clause C into a set of clauses $\{C_1, \ldots, C_n\}$ (n $\leq$ number of premises of C), which, if executed by Prolog, are equivalent to the forward execution of the original clause C. This transformation is called 'horizontal', because the target language is the same as the source language: horn clauses. Based on these experiences of doing forward reasoning in a backward reasoning system an improved 'vertical' compilation into a forward chaining Warren Abstract Machine (WAM [War83]) code will be presented.

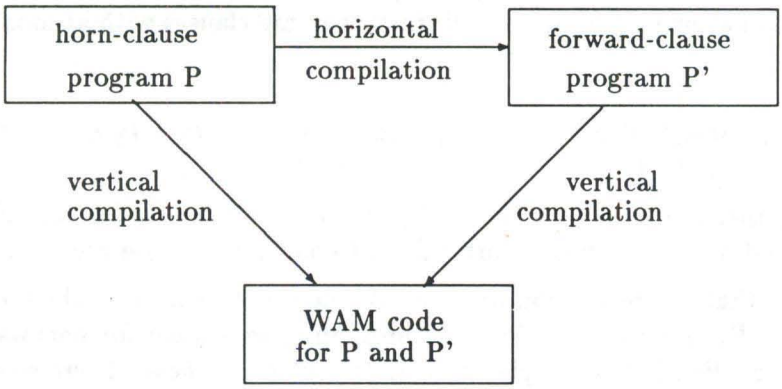


Figure 1: Backward and Forward Compilation of Horn Clauses

2 Rule Characterization

2.1 Logical Implications

A rule in a conventional rule-based system has the following structure:

$$P_1 \wedge P_2 \wedge \ldots \wedge P_m \rightarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

The preconditions $P_1, \ldots, P_m$ on the left-hand side of the rule must be satisfied for the rule to fire. Firing a rule means execution of the actions $A_1, \ldots, A_n$ of the right-hand side.

To get a reasoning system based on predicate calculus we restrict $P_1, \ldots, P_m$ and $A_1, \ldots, A_n$ to logical propositions and interpret "$\rightarrow$" as logical implication. Then $A_1\sigma, \ldots, A_n\sigma$ are logical consequences of the knowledge base, if $P_1\sigma, \ldots, P_m\sigma$ are satisfied ($A_i\sigma, P_j\sigma$ are instances of A_i, P_j respectively). In the rest of the article, the term *deduction rules* refers to these implications. $P_1, \ldots, P_m$ are premises or antecedents and $A_1, \ldots, A_n$ are conclusions of the rule.

2.2 Forward and Backward Reasoning over the Same Rule Set

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 system integrating forward and backward reasoning this means, in the ideal case, that the application direction of a rule need not be visible to the programmer. As a consequence both reasoning directions should be applied to the same rule set. For the sake of knowledge base consistency this seems desirable, too. Assuming two different rule bases in which semantically equivalent rules occur, inconsistencies may arise when only one rule set is updated. Thus our first step is to commit ourselves to a rule syntax suited equally well for each reasoning method.

2.3 Horn Rules

As a common rule structure for combined forward and backward reasoning we decided for horn rules. Logic formulas can be transformed to clauses — universally quantified disjunctions of

positive and negative atoms called literals. Horn clauses are clauses with at most one positive literal:

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

In the rest of the paper the term horn rule will be used synonymously for horn clauses if they are to be interpreted as implications. Horn rules without any premise are facts.

It should be noted, that the restriction to horn rules does not depend on the forward reasoning characteristics. Rather the use of a logic programming system for backward reasoning demands the choice. But it will be shown that the expressiveness of our rule language is increased – compared to that of production systems – by the use of logic variables.

Conventional production rule systems allow a conjunction of conclusions and disjunctions of premises. The conclusions $A_1, \ldots, A_n$ are pairwise independent, because new variables in the conclusion part are prohibited and all variables must be bound during the matching process of the preconditions. A transformation of this kind of rules to horn rules is trivial and could be performed by a precompiler. Rules with conjunctive conclusions

$$P_1 \wedge P_2 \wedge \ldots \wedge P_m \rightarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

are transformed into

$$\begin{aligned} P_1 \wedge P_2 \wedge \ldots \wedge P_m &\rightarrow A_1 \\ P_1 \wedge P_2 \wedge \ldots \wedge P_m &\rightarrow A_2 \\ &\ldots \\ P_1 \wedge P_2 \wedge \ldots \wedge P_m &\rightarrow A_n \end{aligned}$$

These horn rules could be executed without any loss of efficiency taking into account structure sharing between equal premises in different rules. But implementation methods like TREAT [Mir87] or Rete [For82] algorithm are not appropriate in a logic programming framework. They match premises only against ground facts in the working memory, while in our approach premises have to be verified by backward reasoning, which is not possible by doing propagation like Rete algorithms.

If a rule's premise part contains disjunctions it is first transformed into disjunctive normal form

$$D_1 \vee D_2 \vee \ldots \vee D_p \rightarrow A$$

with $D_i, i = 1, \ldots, p$, being conjunctions of literals $P_{i1}, \ldots, P_{iq}$. This kind of rule is equivalent to a sequence of horn rules

$$\begin{aligned} D_1 &\rightarrow A \\ D_2 &\rightarrow A \\ &\ldots \\ D_p &\rightarrow A \end{aligned}$$

Compared to production systems, logic programs have the advantage of compactly representing factual knowledge with variables. Conventional production rules do not allow variables

to occur only in the action part, because derived facts have to be ground. Also, because of shared variables between more than one action undesired dependencies would be established (cf. [FFM89]). However, this is considered harmless for horn rules, because each rule has just one conclusion. After rule firing, variables in the conclusion need not be instantiated, leading to nonground facts in the next chaining step. Therefore unification (instead of pattern matching applied in conventional production systems) for premise satisfaction is important in a logic programming framework.

2.4 Prolog

Since Prolog rules are executed in backward direction it suggests itself to start with Prolog as the basis for an integrated forward and backward reasoning rule system. But some Prolog-specific extra-logic features, which depend on program execution and implementation, are *prohibited* in rules which are to be applied in both directions:

- control of program execution: cut, fail
- side effects: input/output, retraction and assertion of clauses
- meta predicates: clause, functor, arg, ...

The remaining logic part of Prolog itself is used as backward reasoning subcomponent. Thus our goal to integrate forward and backward reasoning over the same rule set is reduced to the task of doing *forward reasoning of logic programs.*

3 Forward Reasoning Characteristics

Prolog's backward reasoning proof procedure starts with a goal $? - q(\ldots)$. It looks for a rule which has a *conclusion* unifiable with this goal. If a rule $q(\ldots) : -p_1(\ldots), \ldots, p_m(\ldots).$ is found, backward reasoning is applied recursively to the premises $p_1(\ldots), \ldots, p_m(\ldots)$ of the rule. These premises are now considered as goals. Forward reasoning, on the other hand, is initiated by a fact $p_j(\ldots)$ which is unifiable (or matching) with one of a rule's *premises.* If the remaining premises of the rule are also satisfied (either by facts or by a backward proof), the conclusion $q(\ldots)$ of this rule is derived. We call the fact $p_j(\ldots)$ the trigger of the forward rule application.

A forward reasoning system repeatedly executes a match-select-act-cycle. In the match phase the antecedents of rules are tested for satisfaction. All rule instantiations with satisfied antecedents form the conflict set. In the select phase one rule instance is chosen from the conflict set and executed in the act phase. In a logic programming framework control should correspond to conventional logic programs. In our approach rules and predicates are executed sequentially in a left-to-right manner. The conflict set is not built explicitly. As soon as an applicable rule is found during the match phase it will be applied. Various reasoning strategies (depth-first vs. breadth-first) and answer presentations (all at once vs. enumerating one by one) can be easily realized.

Knowledge in logic programs is represented implicitly by facts and rules rather than by an explicit enumeration of all true facts. Forward reasoning of horn rules has to take consideration of this pecularity. In particular, forward chaining starts with an initial fact $p(x_1, \ldots, x_n)$. Only propositions derived from this fact are computed by the following procedure:

1. Set the actual fact F to $p(x_1, \ldots, x_n)$.

2. Find the next potentially applicable rule: Rules are processed sequentially. A rule $C : -P_1, \ldots, P_m.$ is triggered, if any $P_i, 1 \leq i \leq m$, is unifiable with the actual fact F with substitution σ.

   If no rule is applicable, go to 5.

3. Test the rule's conditions: The conjunction of the remaining premises $P_1, \ldots, P_{i-1}$, $P_{i+1}, \ldots, P_m$ is verified by conventional backward reasoning.

   If it is not satisfiable, go to 2.

4. Apply the rule: If it is satisfiable with substitution $\tau \geq \sigma$, record the instantiated conclusion $C\tau$ as a derived fact.

5. Select an actual fact F for further reasoning. At least two reasoning strategies are possible:

   breadth first: The actual fact F is kept until there is no further rule for it. Then F is set to the oldest not already expanded fact.

   depth first: F is set to the most recently derived fact $C\tau$ for which there are any rules to be applied.

   Stop if there are no (more) facts with applicable rules. Else proceed with 2.

6. Display the recorded facts as the consequences of the initial fact $p(x_1, \ldots, x_n)$.

The procedure terminates, if the initial fact has finitely many consequences in the given knowledge base.

The integrated breadth-first forward and backward reasoning will be exemplified with a little program about geometry and manufacturing in the domain of mechanical engineering (appendix A). Given a fact `cylinder(a2,4,2)` the `rspear`-rule is triggered for forward reasoning: `cylinder(a2,4,2)` is unifiable with the first premise `cylinder(Cyl,Length1,Radius)`. The remaining premises can be verified by conventional backward reasoning, as the reader may check. The conclusion `rspear(c(a2,a1),5,2)` is recorded. It is called a *reached node*. Every reached node is also an *open node* until forward chaining proceeds with it. Since no further rule is applicable for our initial fact `cylinder(a2,4,2)` forward reasoning continues with the open node `rspear(c(a2,a1),5,2)` inferring `rot_part(c(a2,a1))` and `material(c(a2,a1),metal)`, from which `manufactured(c(a2,a1),lathe_tooling)` is derived.

4 Horizontal Compilation to Prolog Clauses

The horizontal compilation presented in this section takes a set of horn rules $\mathcal{P} = \{C_1, \ldots, C_n\}$ and produces a set of Prolog clauses $\mathcal{P}' = \{C'_1, \ldots, C'_m\}$, which are the corresponding rules of $\mathcal{P}$ for forward reasoning (see below). Executing $\mathcal{P} \cup \mathcal{P}'$ by Prolog's backward reasoning behaves like forward reasoning of $\mathcal{P}$ following the strategy described above (chapter 3). To perform this kind of forward reasoning, a fact $p(x_1, \ldots, x_n)$ has to be compared with every

premise $p_i(\ldots), 1 \leq i \leq m$, of a rule. That is, a rule $q(\ldots) :- p_1(\ldots), \ldots, p_m(\ldots).$ is translated into a *sequence* of forward rules following this pattern:

$$
\begin{array}{lcl}
forward(p_1(\ldots), q(\ldots)) & :- & p_2(\ldots), \ldots, p_m(\ldots), retain(q(\ldots)). \\
forward(p_2(\ldots), q(\ldots)) & :- & p_1(\ldots), p_3(\ldots), \ldots, p_m(\ldots), retain(q(\ldots)). \\
 & \ldots & \\
forward(p_m(\ldots), q(\ldots)) & :- & p_1(\ldots), \ldots, p_{m-1}(\ldots), retain(q(\ldots)).
\end{array}
$$

The transformation process allows forward reasoning in Prolog without any changes to the backward reasoning interpreter. Applying a forward clause corresponds to a one step forward execution of the original horn rule, triggerd by $p_i(\ldots)$. A goal $? - forward(p_i(\ldots), q(\ldots))$ succeeds, if $q(\ldots)$ is a one-step derivation of $p_i(\ldots)$. The rules have the following intended semantics:

> "If the actual fact is unifiable with $p_i(\ldots)$ with most general unifier σ, then prove the remaining premises $p_1(\ldots)\sigma, \ldots, p_{i-1}(\ldots)\sigma, p_{i+1}(\ldots)\sigma, \ldots, p_m(\ldots)\sigma$. If they are satisfied giving substitution $\tau \geq \sigma$, *retain* the conclusion $q(\ldots)\tau$ for further reasoning."

In [YT86] a translation of production rules into Prolog is presented, too. In contrast to our approach, for every rule exactly one Prolog clause is generated. The first premise appears as part of the conclusion, because it is emphasized as a trigger for rule application. Thus, a kind of goal-directed forward reasoning is used to derive the outstanding facts. Goal-directed reasoning, however, could be performed by Prolog itself using the original clauses.

4.1 Rule Translation for Forward Reasoning

The order of rules is significant in logic programming á la Prolog and thus are also significant in the transformed programm. The transformation process preserves this rule sequence: If rule R_1 is before R_2 in the original program its translation R_1' is also before R_2'. Facts of the original knowledge base are ignored during translation. The simplest rules to be translated are those with exactly one premise. The translation process results here in a single `forward`-rule. Thus,

```
cylinder(Name,Length,Radius) :- truncone(Name,Length,Radius,Radius).
```

is translated into the clause

```
forward(truncone(Name,Length,Radius,Radius), cylinder(Name,Length,Radius)) :-
        retain(cylinder(Name,Length,Radius)).
```

Then a goal `?- forward(truncone(a2,3,2,2),X)` would derive the instantiated conclusion `X = cylinder(a2,3,2)`, without invocation of backward verification.

The first argument term of `forward/2` serves as a trigger if it is unifiable with the actual fact. If a rule has more than one non-primitive premise, several forward clauses are generated, one for each premise that is not a negated premise or a Prolog built-in. Therefore the translation of the rule

```
rspear(c(Cyl,Cone),Length,Radius) :-  cylinder(Cyl,Length1,Radius),
                                      rcone(Cone,Length2,Radius),
                                      Length is Length1 + Length2,
                                      connected(Cyl,Cone).
```

has three instead of four clauses, since the third premise `Length is Length1 + Length2` will not be unifiable with an assertion:

```
forward(cylinder(Cyl,Length1,Radius),rspear(c(Cyl,Cone),Length,Radius)) :-
        rcone(Cone,Length2,Radius),
        Length is Length1 + Length2,
        connected(Cyl,Cone),
        retain(rspear(c(Cyl,Cone),Length,Radius)).
forward(rcone(Cone,Length2,Radius),rspear(c(Cyl,Cone),Length,Radius)) :-
        cylinder(Cyl,Length1,Radius),
        Length is Length1 + Length2,
        connected(Cyl,Cone),
        retain(rspear(c(Cyl,Cone),Length,Radius)).
forward(connected(Cyl,Cone),rspear(c(Cyl,Cone),Length,Radius)) :-
        cylinder(Cyl,Length1,Radius),
        rcone(Cone,Length2,Radius),
        Length is Length1 + Length2,
        retain(rspear(c(Cyl,Cone),Length,Radius)).
```

The translation of the entire sample knowledge base of appendix A is presented in appendix B. It consists of one large procedure for `forward/2`. Clauses generated from each source rule are grouped together.

4.2 Retaining of Conclusions

A forward reasoning system applies rules with satisfied premises by recording its conclusion. These derived facts may again trigger other forward rules. The forward reasoning process can be visualized as a deduction tree, with the initial fact as its root and derived facts as nodes. The sons of a node N are the instantiated conclusions of applied rules triggered by N. Therefore, all derived facts are also called *reached nodes*. The order in which the deduction tree is built up depends on the search strategy. Presently, breadth-first and depth-first strategies are realized (see below).

Reasoning depth first the most recently derived fact is selected for forward chaining corresponding directly to Prolog's execution strategy. Proceeding breadth-first, however, the managing overhead increases. For every reached node it has to be recorded whether it is already expanded, i. e. whether it has been used to trigger a forward rule. The definition of the predicate `retain/1` (Fig. 2) reflects this overhead.

A *new* derivation `Conclusion` is asserted twice: in the form `reached(Conclusion)` and as `open_node(Conclusion)`. The predicate `reached/1` indicates that its argument is a fact derived by forward reasoning. The status of a new fact as *reached* remains until the end of the forward chaining process. The information about open nodes is necessary for breadth-first strategy. As soon as `Conclusion` has been selected for forward chaining, the clause `open_node(Conclusion)` will be retracted (c.f. section 4.3).

To avoid redundancies, the conclusion of an applied rule is asserted only if it has not already been derived: The premise `not_reached(Conclusion)` succeeds, if the derived fact

```
retain(Conclusion) :- not_reached(Conclusion),
                      assertz(reached(Conclusion)),
                      assertz(open_node(Conclusion)).
```

Figure 2: Retaining of Conclusions

`Conclusion` is *not subsumed* by any formerly reached node. Since in a logic programming framework facts and conclusion may contain unbound variables a simple unification with previously reached nodes does not suffice, in particular if the new conclusion is more general.

Consider a new conclusion `manufactured(X,lathe_tooling)` and a formerly reached node `manufactured(a1,lathe_tooling)`. Simple unification with previous nodes would regard `manufactured(X,lathe_tooling)` as already reached although it is more general. A correct solution must therefore test for subsumption with matching. A possible realization is given in appendix C. First the recently derived fact is made ground using *new* terms, and then it is unified with the previously reached nodes. If it is unifiable with any reached node it is rejected. [1]

4.3 Depth-first and Breadth-first Strategies

Appendix E shows the definitions of depth-first and breadth-first strategies of forward reasoning. They derive consequences for all instantiations of the initial fact. For both strategies the selection of applicable rules depends on the sequence of rules in the program.

4.3.1 Enumerating Derivations Depth First

Depth-first search (appendix E) always proceeds with the most recently derived fact. If no further step is possible, another path is tried by backtracking. Depth-first forward chaining is activated by a call of the top-level predicate `df_enum/2`. The procedure `df_one/2` enumerates the consequences for its first argument, `Fact`, one at a time. The one-step consequence of `Fact` is bound to `Conclusion`. The next reasoning step is activated by backtracking. A query to the knowledge base of appendix A – together with its horizontally compiled version of appendix B – using depth-first strategy could look like:

```
?- df_enum(truncone(a2,4,2,2),Result).

Result = cylinder(a2,4,2);
Result = rspear(c(a2,a1),5,2);
Result = rot_part(c(a2,a1));
Result = manufactured(c(a2,a1),lathe_tooling);
Result = material(c(a2,a1),metal);
Result = rot_part(a2);
Result = manufactured(a2,lathe_tooling);
no
```

[1] In our current implementation an already reached node `manufactured(a1,lathe_tooling)`, which is more special than a new node, will not be retracted.

4.3.2 Enumerating Derivations Breadth First

Appendix E also presents two versions for breadth-first forward reasoning, enumerating the conclusions one by one (`bf_enum/2`) and presenting them all at once (`bf_all/2`). Both reasoning strategies access *open nodes*, retained in `forward/2`. Open nodes are all those leaves of the deduction tree that are not yet selected for breadth-first rule firing.

In an initialization step the temporary store is cleared. The first clause of `bf_enum/2` computes the facts derivable from the initial fact in one step. The second clause computes derivations of already derived open nodes.

The procedure for `forward_one/1` stops, if a new node is reached. If an open node is found, `forward/2` is called. In `forward_one/1` the actual node is bound to the result variable `Inference`. Via backtracking further solutions are derived by `forward/2`. If no further inferences can be drawn, the next open node is tried. An enumerating breadth-first query could proceed like:

```
?- bf_enum(truncone(a2,4,2,2),Result).

Result = cylinder(a2,4,2);
Result = rot_part(a2);
Result = rspear(c(a2,a1),5,2);
Result = manufactured(a2,lathe_tooling);
Result = rot_part(c(a2,a1));
Result = material(c(a2,a1),metal);
Result = manufactured(c(a2,a1),lathe_tooling);
no
```

4.3.3 Presenting Derivations all at once

A goal `?- bf_all(Fact,Inferences)` succeeds, if `Inferences` is instantiated to a list with all facts derivable from `Fact`. The procedures `bf_all/2` and `forward_all/1` are very similar to `bf_enum/2` and `forward_one/1`, respectively. Backtracking, however, is initiated by the clause itself using `fail` and not by the calling predicate or the user. If no further forward reasoning step is possible, the derived facts are collected in a list, which is unified with the result variable `Inferences`:

```
?- bf_all(truncone(a2,4,2,2),Result).

Result = [cylinder(a2,4,2), rot_part(a2), rspear(c(a2,a1),5,2),
          manufactured(a2,lathe_tooling), rot_part(c(a2,a1)),
          material(c(a2,a1),metal), manufactured(c(a2,a1),lathe_tooling)];
no
```

4.4 Rule combinations

For reasons of efficiency the compiler could combine rules with equivalent premises into one forward clause as exemplified with the two rules

```
p1(a,b) :- q(a),r(b).
p2(b) :- q(a),r(b).
```

Both rules have equivalent premises but different conclusion. They are transformed to two forward clauses:

```
forward(q(a),[p1(a,b),p2(b)]) :- r(b), retain(p1(a,b)), retain(p2(b)).
forward(r(b),[p1(a,b),p2(b)]) :- q(a), retain(p1(a,b)), retain(p2(b)).
```

The conclusions of the rules are collected in a list and the call to `retain` has to be duplicated for the various conclusions. For breadth-first strategy with derivation of all consequences no changes to the implementation are necessary. For enumeration of consequences, however, the reasoning strategy has to be adapted, because of the list of conclusions. This makes clear how forward reasoning could also treat more general rules with a conjunction of conclusions.

5 Vertical Compilation into WAM Code

The Warren Abstract Machine [War83] is an often implemented architecture for backward reasoning of horn clauses. After horizontal transformation of a horn clause program P into a forward clause Prolog program P' (cf. chapter 4) the clauses of P and P' are compiled vertically into WAM code (see fig. 1). But since the WAM was developed especially for backward reasoning, several improvements for forward rules are possible. They extend the WAM by a special stack area for derived facts, called *retain stack*, and a one-way unification for subsumption tests. In the following subsections names of operations, stacks, and registers are taken from [GLLO85]. The tags REF(erence), STR(ucture), and LIS(t) are borrowed from [AK90].

5.1 The Retain Stack

Derived facts in horizontally compiled forward rules are retained by space consuming double assertion with predicates `reached/1` and `open_node/1` (see 4.2). Such assertions are rather inefficient because program code itself is altered dynamically. But the information whether a node is reached or open is necessary. It is held more compactly at machine level in a special data area which will be called the *retain stack* `RETAIN` (see fig. 3). The stack is organized as a list: The first REF cell points to the current entry and the following LIS cell points to the beginning of the next item. Every entry on the stack is an internal representation of a proposition derived by forward rule application. It consists of variable, constant, list and structure cells distinguished by tags. An example is given in fig. 3.

The pointer `RTOP` indicates the top of the retain stack. All entries of the retain stack are reached nodes. For the breadth-first strategy of forward reasoning the expansion of open nodes is performed in the order in which they are generated. This order is identical to the order of the nodes on the retain stack. Open nodes are accessed by the register `ON`. Every node with an address higher than `ON` is an open node. Whenever the node at address `ON` is expanded, `ON` is increased.

5.2 Compiling Retain

The clause for the predicate `retain/1` (fig. 2) in a forward clause is compiled into a sequence of WAM operations pushing its argument – the derived fact – onto the retain stack.

```
retain/1:  not_r_subsumed X1        % Test for subsumption
           push_fact_retain X1      % Copying the fact to RETAIN
```

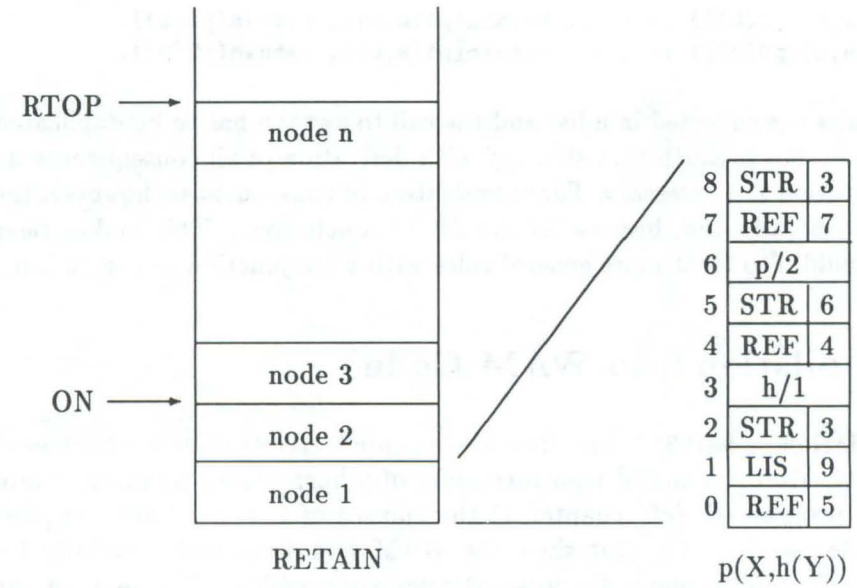


Figure 3: Retain Stack

To accept the new fact it must be secured that it is not subsumed by any structure already existing on the stack. A new operation `not_r_subsumed Xi` is introduced performing this test. The new fact referenced by `Xi` is matched against *every* entry on the retain stack. It calls the function `subsumes(x,y)` to test subsumption. The functions `unify(x,y)` and `subsumes(x,y)` differ only in two cases: If x and y are unbound REF cells then a new constant c_i is created and y is bound to c_i. If y is an unbound REF cell and x is a non-REF cell, the test fails, because a REF cell is not subsumed by a value. The rest of the procedure remains unchanged. Backtracking occurs, if subsumption of the derived fact with any previously derived fact succeeds (see appendix D).

If subsumption fails no backtracking occurs and the new fact is pushed onto the retain stack by the operation `push_fact_retain Xi`. The values on the retain stack are "more persistent" than values on the global stack or the local stack. While values on the local and global stack may be destroyed by backtracking derived facts must survive for the whole forward inference chain. Because of this no reference from the retain stack to any other memory cell is permitted. This is why a derived fact is *copied*. Before pushing variables are dereferenced. If the dereferenced value is not an unbound variable cell it is *copied* onto the retain stack and dereferencing is performed recursively for every subvariable in the functor structure. Otherwise, for an unbound variable, a new REF cell is pushed onto the retain stack referring to itself. Finally, `RTOP` is increased, completing the retain operation.

5.3 Compiled Strategies

The clauses representing different reasoning strategies (appendix E) refer to structures residing on the retain stack. So their compiled version needs some modifications compared to a straightforward compilation. These modifications are rather obvious, but since the retain stack is an extension to the conventional WAM, novel operations are introduced.

- Performing forward chaining initialization resets the pointers `ON` and `RTOP` to the bottom address of the retain stack: `fc_initialize`
- Accessing an open node is realized by getting the structure at stack position `ON`. A call to `open_node Xi` meets the requirements setting `Xi` to the open node.
- Retracting an `open_node` fact is equivalent to increasing `ON` to point to the successive stack content: `next_open_node`
- Breadth-first reasoning stops, if there exists no further open node. This is equivalent to the state when `ON = RTOP`
- Collecting all derived facts in the second clause for `forward_all` is very simple. Since the retain stack is organized as a list just load the bottom address of the stack into register `Xi`. Calling `collect_all` on source level corresponds to the execution of the new WAM instruction `fc_collect Xi`.

The depth-first and breadth-first strategy clauses are invariant, independent from the knowledge base to be compiled. Thus after compilation of the original source program their code will just be added to the forward program.

6 Application: Integrity Constraints

These kind of forward rules could be used to detect whether knowledge base updates would lead to inconsistencies. Consider a logic program with integrity constraints denoting negative or disjunctive knowledge. These integrity constraints are represented as clauses with the atom `inconsistent` as conclusion (negation as inconsistency [GM86]). Thus, the following rule demands that two connected parts must have the same radius at their contact point.

```
inconsistent :-  connected(P1,P2),
                 truncone(P1,_,_,R1),
                 truncone(P2,_,R2,_),
                 R1 =\= R2.
```

A real knowledge base will have many of these integrity constraints. Now assume, that the facts `truncone(a3,5,2,2)` and `truncone(a4,4,3,0)` are deducible from the knowledge base and we want to connect these truncated cones. Using Prolog's proof strategy one has to assert the fact `connected(a3,a4)` and then to ask the query `?- inconsistent`. This procedure would invoke all integrity constraints even if they are independent from the new fact. Instead, it would be more efficient to invoke only those rules, that are directly influenced by this new assertion. The following goal succeeds, if `connected(a3,a4)` is inconsistent with the integrity constraints

```
?- bf_enum(connected(a3,a4),R), R == inconsistent.
```

7 Conclusions and Future Work

An approach for combined forward and backward reasoning of horn rules has been presented. The whole system is embedded in a logic programming environment. The same horn rule set

is used for both reasoning directions. In a first step horn rules are horizontally transformed into clauses corresponding to one step of forward reasoning. Conclusions of applied rules are not asserted directly into the knowledge base. Instead, they are recorded as arguments of special predicates. The intended use is to generate hypotheses following from a given fact and perform tests by the backward reasoning logic program. Breadth-first and depth-first strategies have been presented. The ideas have been implemented for the horn clause subpart of RELFUN [Bol90], a relational/functional language with valued clauses.

For vertical compilation a special retain stack to record derived facts extends the WAM. This stack saves memory space and makes access more efficient compared to a direct compilation of the code, obtained by horizontal transformation. A source of inefficiency is the sequential subsumption test for reached clauses. It could be improved by a kind of hashing taking the functor as a key. The subsumption test of a new fact with previously derived ones can also be made more efficient by variations of the WAM's unification operations. As a further improvement [Olt91] presents a tree-like organization of these operations. Further deliberations concern the permanent assertion of derived facts. After forward chaining has terminated the user can be given the opportunity to select them for assertion if she or he considers them relevant for further processing.

In the current status forward rules are represented with a single predicate `forward`. Access to applicable rules is just supported by indexing on the first argument's functor. Introducing a special code area for forward clauses, those clauses with the same predicate for the trigger premise could be grouped together. Then, unification of the actual fact with the rule's trigger could be supported by WAM operations. The special code area takes precautions to distinguish the original backward rules and facts from the compiled forward clauses.

The plain control strategy is induced by the SLD-resolution procedure of logic programming. Forward rules are selected for execution in a strictly sequential manner. But implementation methods like TREAT [Mir87] or Rete [For82] algorithm are not appropriate since premises are proved by backward reasoning in our approach. Besides breadth-first and depth-first strategies, more sophisticated control strategies are conceivable, e. g. best-first, requiring some kind of estimation. Especially in larger applications, where rules reflect an expert's heuristics, more flexible control strategies are desirable. [PH88] promotes control of rule firing at instance level taking into account variable instantiations. Their strategies for rule selection are decribed by the programmer using a kind of meta rules.

References

[AK90] Hassan Ait-Kaci. The WAM: A (Real) Tutorial. Report 5, Digital, Paris Research Laboratory, January 1990.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 1–15. ACM, 1986.

[Bol90] Harold Boley. A relational/functional Language and its Compilation into the WAM. SEKI Report SR-90-05, Universität Kaiserslautern, 1990.

[BR86] 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.

[BR88] Francois Bancilhon and Raghu Ramakrishnan. Performance evaluation of data intensive logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 441–517. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1988.

[CDE87] D. Chan, P. Dufresne, and R. Enders. Report on phocus. Technical Report TR-LP-21-02, ECRC, Arabellastraße 17 D8 München 81, April 1987.

[FFM89] Tim Finin, Rich Fritzson, and Dave Matuszek. Adding Forward Chaining and Truth Maintenance to Prolog. In *Artificial Intelligence Applications Conference*, pages 123–130, Miami, March 1989. IEEE.

[For81] Charles L. Forgy. *OPS5 User's Manual.* Carnegie-Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania 15213, 1981.

[For82] Charles L. Forgy. Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[GLLO85] John Gabriel, Tim Lindholm, E. L. Lusk, and R.A. Overbeek. A Tutorial on the Warren Abstract Machine for Computational Logic. Report ANL-84-84, Argonne National Laboratory, Argonne, Illinois 60439, June 1985.

[GM86] D.M. Gabbay and Sergot M.J. Negation as Inconsistency I. *Journal of Logic Programming*, 1(1), 1986.

[Int86] IntelliCorp. *KEE Software Development System User's Manual; KEE Version 3.0.* Intellicorp Corp., Mountain View, CA, 1986.

[Kow79] Robert Kowalski. *Logic for Problem Solving.* Artificial Intelligence Series. Elsevier North Holland, 1979.

[Mir87] Daniel P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. In *Proc. of AAAI-87*, pages 42–47, Philadelphia, PA, 1987.

[Mor81] Paul Morris. A Forward Chaining Problem Solver. *Logic Programming Newsletter*, 2:6–7, Autumn 1981.

[Olt91] Thomas Oltzen. Term subsumption in the wam. Projektarbeit (in German), 1991.

[PH88] Charles J. Petrie and Michael N. Huhns. Controlling Forward Rule Instances. MCC Technical Report ACA-AI-012-88, Microelectronics and Computer Technology Corporation, 3500 West Balcones Center Drive, Austin, TX, January 1988.

[RLK86] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The alexander method - a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, pages 273–285, 1986.

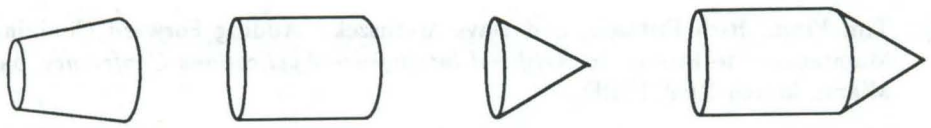
[TG87] Richard Treitel and Michael R. Genesereth. Choosing Directions for Rules. *Journal of Automated Reasoning*, 3:395–431, 1987.

[War83] David. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

[YT86] Akira Yamamoto and Hozumi Tanaka. Translating Production Rules into a Forward Reasoning Prolog Program. *New Generation Computing*, 4:97–105, 1986.

A Geometrical and Manufacturing Knowledge

An example knowledge base with geometrical and manufacturing knowledge.




```
% Rules:

cylinder(Name,Length,Radius) :-          % A cylinder is a truncated
    truncone(Name,Length,Radius,Radius).  % cone with left radius
                                          % equal to right radius

lcone(Name,Length,Radius) :-             % A left cone is a truncated
    truncone(Name,Length,0,Radius).       % cone with left radius 0

rcone(Name,Length,Radius) :-             % a right cone is a truncated
    truncone(Name,Length,Radius,0).       % cone with right radius 0

rspear(c(Cyl,Cone),Length,Radius) :-     % A right spear is composed of
    cylinder(Cyl,Length1,Radius),         % a cylinder connected to
    rcone(Cone,Length2,Radius),           % a cone with tip on the right.
    Length is Length1 + Length2,
    connected(Cyl,Cone).

rot_part(X) :- truncone(X,_,_,_).        % truncated cones and right
rot_part(X) :- rspear(X,_,_).            % spears are rotation
                                          % symmetric parts

manufactured(Part,lathe_tooling) :-      % Metallic rotation symmetric
    rot_part(Part),                       % parts are manufactured by
    material(Part, metal).                % lathe_tooling.

material(c(Cyl,Cone),Mat) :-             % The material of a right spear
    rspear(c(Cyl,Cone),_,_),              % is the same as that of each
    material(Cyl,Mat),                    % component.
    material(Cone,Mat).


% Facts:

truncone(a1,1,2,0).
connected(a2,a1).
material(a1,metal).
material(a2,metal).
```

B Horizontally Compiled Rules

Horizontal compilation of the geometrical and manufacturing knowledge base (appendix A):

```
  forward(truncone(Name,Length,Radius,Radius),cylinder(Name,Length,Radius)) :-
       retain(cylinder(Name,Length,Radius)).

forward(truncone(Name,Length,0,Radius),lcone(Name,Length,Radius)) :-
       retain(lcone(Name,Length,Radius)).

forward(truncone(Name,Length,Radius,0),rcone(Name,Length,Radius)) :-
       retain(rcone(Name,Length,Radius)).

forward(cylinder(Cyl,Length1,Radius),rspear(c(Cyl,Cone),Length,Radius)) :-
       rcone(Cone,Length2,Radius),
       Length is Length1 + Length2,
       connected(Cyl,Cone),
       retain(rspear(c(Cyl,Cone),Length,Radius)).
forward(rcone(Cone,Length2,Radius),rspear(c(Cyl,Cone),Length,Radius)) :-
       cylinder(Cyl,Length1,Radius),
       Length is Length1 + Length2,
       connected(Cyl,Cone),
       retain(rspear(c(Cyl,Cone),Length,Radius)).
forward(connected(Cyl,Cone),rspear(c(Cyl,Cone),Length,Radius)) :-
       cylinder(Cyl,Length1,Radius),
       rcone(Cone,Length2,Radius),
       Length is Length1 + Length2,
       retain(rspear(c(Cyl,Cone),Length,Radius)).

forward(truncone(X,_,_,_),rot_part(X)) :- retain(rot_part(X)).

forward(rspear(X,_,_),rot_part(X)) :- retain(rot_part(X)).

forward(rot_part(Part),manufactured(Part,lathe_tooling)) :-
       material(Part,metal),
       retain(manufactured(Part,lathe_tooling)).
forward(material(Part,metal),manufactured(Part,lathe_tooling)) :-
       rot_part(Part),
       retain(manufactured(Part,lathe_tooling))

forward(rspear(c(Cyl,Cone),_,_),material(c(Cyl,Cone),Mat))  :-
       material(Cyl,Mat),
       material(Cone,Mat),
       retain().
forward(material(Cyl,Mat),material(c(Cyl,Cone),Mat))  :-
       rspear(c(Cyl,Cone),_,_),
       material(Cone,Mat),
       retain(material(c(Cyl,Cone),Mat)).
forward(material(Cone,Mat),material(c(Cyl,Cone),Mat)) :-
       rspear(c(Cyl,Cone),_,_),
       material(Cyl,Mat),
       retain(material(c(Cyl,Cone),Mat)).
```

C Subsumption on Source Level

A derived fact is retained, if it has not been reached by a former inference step:

```
retain(Conclusion) :- not_reached(Conclusion),
                      asserta(reached(Conclusion)),
                      assertz(open_node(Conclusion)).
```

A call to `?- not_reached(Conclusion)` succeeds, if `Conclusion` is not subsumed by any previously reached fact. First `Conclusion` is instantiated with *new* terms, then a proof must fail:

```
not_reached(Conclusion) :-
    instantiate(Conclusion,ConcInst,0,_),!,
    not(reached(ConcInst)).
```

A goal `?- instantiate(X,Xinst,C1,C2)` succeeds, if the instantiation of `X` with *new* terms equals `Xinst`. `C1` and `C2` are counters identifying the terms. The terms look like `const(C1)`. The functor `const/1` must be used nowhere else in the program:

```
instantiate(X,const(C1),C,C1) :-          % variable instantiation:
    var(X),                               % unbound variables are
    !,                                    % are instantiated to
    C1 is C+1.                            % const(C1)

instantiate([],[],C,C) :- !.

instantiate([H|T],[Hinst|Tinst],C,C2) :-  % instantiating head
    !,                                    % and tail of lists
    instantiate(H,Hinst,C,C1),
    instantiate(T,Tinst,C1,C2).

instantiate(X,X,C,C) :-                   % constants are
    atomic(X),                            % instantiations
    !.                                    % of themselves

instantiate(X,Xinst,C,C1) :-              % instantiating
    X =.. [Pred|Args],                    % arguments of terms
    instantiate(Args,Argsinst,C,C1),
    Xinst =..[Pred|Argsinst].
```

D Subsumption on WAM Level

A call to **retain/1** in a forward clause is compiled into a sequence of WAM operations pushing its argument – the derived fact – onto the retain stack.

```
retain/1:  not_r_subsumed X1        % Test for subsumption
           push_fact_retain X1      % Copying the fact to RETAIN
```

The WAM operation **not_r_subsumed Xi** tests, whether the value referenced by **Xi** is not subsumed by any structure on the retain stack:

```
not_r_subsumed =
    success <- false; RS <- 0;  % Initialization: Start address of RETAIN
    while (RS < RTOP) and not success
       subsumes(Xi,RS);
       <tag,value> <- RETAIN[RS + 1];   % second cell contains address of
       RS <- value;                     % successor structure: <LIS,addr>
    endwhile;
    if success
       then begin                              % Backtracking if
            P <- STACK[B + STACK[B] + 4];      % subsumption succeeds
            end
       else begin                              % Accept: subsumption failed
            P <- P + instruction_size(P);      % next instruction
            end
```

Procedure **subsumes(a1,a2)** sets variable **success** to **true**, if the structure at **a2** is subsumed by or equal to the structure at **a1**:

```
procedure subsumes (a1, a2: address);
   push(a1,PDL); push(a2,PDL);
   success <- true;
   while (not(empty(PDL)) and success) do
      begin
      d2 <- deref(pop(PDL)); d1 <- deref(pop(PDL));
      if d1 <> d2
         then begin
              <t1, v1> <- STORE[d1]; <t2, v2> <- STORE[d2];
              if (t1 = REF)
                 then begin
                      STORE[d2] <- new_constant; bind(d1,d2)
                      end
                 else if (t2 = REF)
                         then success <- false
                         else begin
                              f1/n1 <- STORE[v1]; f2/n2 <- STORE[v2];
                              if ((f1 = f2) and (n1 = n2))
                                 then for i <- 1 to n1 do
                                      begin
                                      push(v1 + i, PDL); push(v2 + i, PDL)
                                      end
                                 else success <- false
                              end
              end
      end
end subsumes;
```

E Depth-first and Breadth-first Forward Reasoning

```
% Depth-first enumeration:
          df_enum(Fact,Inference) :- fc_initialize,
                                     call(Fact),
                                     df_one(Fact,Inference).

          df_one(Fact,Inference) :- forward(Fact,Conclusion),
                                    df_one_more(Conclusion,Inference).

          df_one_more(Conclusion,Conclusion).
          df_one_more(Conclusion,Next) :- df_one(Conclusion,Next).

% Breadth-first enumeration:
          bf_enum(Fact,Inference) :- fc_initialize,
                                     call(Fact),
                                     forward(Fact,Inference).
          bf_enum(Fact,Inference) :- forward_one(Inference).

          forward_one(Inference)  :- open_node(Fact),
                                     retract(open_node(Fact)),
                                     forward(Fact,Inference).

% Breadth-first: all at once
          bf_all(Fact,Inferences) :- fc_initialize,
                                     call(Fact),
                                     forward(Fact,_),
                                     fail.
          bf_all(Fact,Inferences) :- forward_all(Inferences).

          forward_all(Inferences) :- open_node(Fact),
                                     retract(open_node(Fact)),
                                     forward(Fact,_),
                                     fail.
          forward_all(Inferences) :- collect_facts(Inferences).


% Auxiliary clauses:
          collect_facts([First|Rest]) :- reached(First),!,
                                         retract(reached(First)),
                                         collect_facts(Rest).
          collect_facts([]).

          fc_initialize       :- abolish(open_node,1),
                                 abolish(reached,1).
```

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

RR-90-13
Franz Baader: Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles
40 pages

RR-90-14
Franz Schmalhofer, Otto Kühn, Gabriele Schmidt: Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories
20 pages

RR-90-15
Harald Trost: The Application of Two-level Morphology to Non-concatenative German Morphology
13 pages

RR-90-16
Franz Baader, Werner Nutt: Adding Homomorphisms to Commutative/Monoidal Theories, or: How Algebra Can Help in Equational Unification
25 pages

RR-91-01
Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, and Gert Smolka : On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations
20 pages

RR-91-02
Francesco Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, Werner Nutt: The Complexity of Existential Quantification in Concept Languages
22 pages

RR-91-03
B.Hollunder, Franz Baader: Qualifying Number Restrictions in Concept Languages
20 pages

RR-91-05
Wolfgang Wahlster, Elisabeth André, Winfried Graf, Thomas Rist: Designing Illustrated Texts: How Language Production is Influenced by Graphics Generation.
17 pages

RR-91-06
Elisabeth André, Thomas Rist: Synthesizing Illustrated Documents
A Plan-Based Approach
11 pages

RR-91-07
Günter Neumann, Wolfgang Finkler: A Head-Driven Approach to Incremental and Parallel Generation of Syntactic Structures
13 pages

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 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

TM-90-03
Franz Baader, Bernhard Hollunder: KRIS: Knowledge Representation and Inference System -System Description-
15 pages

TM-90-04
Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, Hans-Jürgen Profitlich: Terminological Knowledge Representation: A Proposal for a Terminological Logic
7 pages

TM-91-01
Jana Köhler
Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

TM-91-02
Knut Hinkelmann
Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 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

D-90-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Formalismus zur Repräsentation von Geo-metrie- und Technologieinformationen als Teil eines Wissensbasierten Produktmodells
66 Seiten

D-90-06
Andreas Becker: The Window Tool Kit
66 Seiten

Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation

Knut Hinkelmann

TM-91-02
Technical Memo