



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document

D-92-22

**Indexing Principles for
Relational Languages Applied to
PROLOG Code Generation**

Werner Stein

February 1993

**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, SEMA Group, 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
- Computer Linguistics
- Programming Systems
- Deduction and Multiagent Systems
- Document Analysis and Office Automation.

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

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

Friedrich J. Wendl
Director

Indexing Principles for Relational Languages Applied to PROLOG Code Generation

Werner Stein

DFKI-D-92-22

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 1993

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 applicable 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.

Indexing Principles of Relational Languages Applied to PROLOG Code Generation

Werner Stein
Universität Kaiserslautern
W-6750 Kaiserslautern, F.R. Germany

February 24, 1993

Abstract

In this paper we propose an extensible, flexible, multi-argument indexing technique for relational languages. We present a compiler producing indexing header code for a PROLOG emulator based on the Warren Abstract Machine. We will show that our technique combines positive aspects of relational database methods and other existing WAM-based indexing schemes. All the indexing concepts introduced are implemented in LISP for the relational-functional programming language RELFUN.



Acknowledgements

to Harold Boley, Micheal Sintek and Hans Günther Hein for their helpfull discussions and for reading this paper before publishing. Thanks to Hans Günther Hein and Thomas Krause for their first introductions into RELFUN and WAM techniques. I would like to express my gratitude to Harold Boley, Michael Sintek and all others working in the ARC-TEC-project for supporting me in making this paper a success. I also thank my wife Ester for taking my mind off when I got struck and for giving me new energy when I was down.

The ideas described in this paper were first presented at the Workshop “Sprachen für KI-Anwendungen, Konzepte – Methoden – Implementierungen” 1992 in Bad Honnef. Parts of the paper are puhlised in [24].

This paper is part of a collaborative work together with Michael Sintek. The other part is published in [23].

Contents

1 Overview	8
I Introduction	9
2 Indexing: What is it, Where Does it Come From?	9
2.1 Index Functions	9
2.2 DB-Indexing	10
2.3 Indexing in PROLOG	10
2.4 Index Algorithms	11
3 What is Indexing Good For?	12
II Theory	13
4 Exponentially Large Index Trees	14
5 NP-Complete Index Problem	16
III Basic State of the Art	19
6 Looking at Other Approaches	20
6.1 Hardware Oriented Approaches	20
6.1.1 m-in-n-Coding	20
6.2 Software Oriented Approaches	20
6.2.1 General WAM-indexing	20
6.2.2 Complete Indexing	22
6.3 Index Assistant Functions	25
6.3.1 Shallow Backtracking	25
6.3.2 Quadratic Indexing	25
7 Developmental Environment	25
7.1 Global RELFUN Project Structure	26
IV Implementation	27

8	A Partitioned Implementation	27
8.1	First Part	29
8.2	Second Part	29
8.2.1	The Classified Clauses (indexing part)	29
8.2.2	The (ν -)WAM	31
9	Separate Compilation: Indexing-Code, Clause-Code	32
10	Idea	33
10.1	Index Trees	33
10.1.1	General Informed Index Trees	33
10.1.2	Header Informed Index Trees	34
10.2	Horizontal Compilation Scheme	35
10.3	Example	37
10.4	Standard Indexing	38
10.5	Improved Indexing I (not only first argument)	43
10.6	Improved Indexing II (not only one argument)	47
11	RFM Indexing	50
11.1	The Way of Compiling Index Code	50
11.1.1	Creating the Index Tree	51
11.1.2	Flattening the Index Tree	51
11.1.3	Cutting the Index Tree	51
11.1.4	Expanding the Index Tree	53
12	Sample Session	53
12.1	Interface	59
13	Comparisons	60
14	Extensions	60
14.1	Improved Indexing III (not only main structure)	62
15	Benchmark Results	62
V	Appendix	65
A	User Commands	65

B Program	65
B.1 MODULE: IDX.LSP	65
B.2 MODULE: IIF.LSP	66
B.3 MODULE: LINEAR.LSP	66
B.3.1 Algorithms	67
B.4 MODULE: ICG.LSP	70
C Benchmark Sources	71
C.1 nrev Benchmark	71
C.2 dnf Benchmark	71
C.3 NET Benchmark	72
D Extended Abstract	76

List of Figures

Figure 3: uninformed index tree	13
Figure 5: exponentially growing index tree	15
Figure 7: RELFUN's clauses	19
Figure 12: merge-complete index tree	24
Figure 13: global RELFUN structure	26
Figure 14: RELFUN structure with indexing	27
Figure 16: separate compilation	32
Figure 17: graphical representation & corresponding instructions	34
Figure 19: sample h-i-index tree: no indexing	38
Figure 20: sample h-i-index tree: first argument indexing, partitions	39
Figure 21: block-variable-size = 0	40
Figure 22: sample h-i-index tree: first argument indexing, no partitions	42
Figure 23: block-variable-size = max	42
Figure 24: sample h-i-index tree: second argument indexing	44
Figure 25: sample h-i-index tree: one of two arguments indexed	46
Figure 26: sample h-i-index tree: two arguments indexed	48
Figure 27: sample h-i-index tree: fully indexed	49
Figure 28: tree-sharing	51
Figure 29: cut h-i-index tree without tree-sharing	52
Figure 30: cut h-i-index tree with tree-sharing	52
Figure 31: nested index tree	56
Figure 32: flattened index tree	56
Figure 33: cut index tree	57
Figure 34: extended index tree	57
Figure 35: extension: assert	60
Figure 36: retract clause number 4	61
Figure 38: flattening algorithm	67
Figure 39: cutting algorithm	68
Figure 40: extending algorithm	68

List of Tables

Table: 1: Main loop of PROLOG	9
Table: 2: Main loop of a simple DB-language	10
Definition 1: uninformed index tree	13
Table: 4: Procedure with an exponentially large index tree	14
Definition 2: complete-position-set-problem	17
Table: 6: NP-complete index problem	17
Table: 8: general three level indexing scheme	21
Table: 9: WAM indexing instruction set	21
Table: 10: three level complete indexing scheme	23
Table: 11: complete indexing instruction set	23
Table: 15: WAM registers	31
Definition 3: g-i-index tree	33
Definition 4: h-i-index tree	34
Table: 18: constraints	35
Definition 5: flow-path through an h-i-index tree	35
Definition 6: valid h-i-index tree	35
Definition 7: depth of a flow-path	36
Definition 8: depth of an h-i-index tree	36
Definition 9: breadth of a flow-path	36
Definition 10: breadth of an h-i-index tree	36
Definition 11: chw	36
Definition 12: cow	37
Definition 13: block-variable-size	37
Table: 37: run-time results	63

1 Overview



In the last few years PROLOG has changed its appearance from an experimental to a more and more realistic language. This is due to many people's thinking about good compiling techniques and useful extensions. Perhaps the single most important factor of efficiency of large PROLOG programs is indexing, an optimization which can always be applied (independent of other compiler optimizations). The original WAM (defined by D.H.D.Warren [29]) only provides a first argument indexing scheme. We will show that a simple extension of the WAM control instruction set can speed up execution efficiency.



In the first part, we will introduce the idea of indexing and where it comes from. The difference between relational database indexing methods and those for PROLOG-like languages will be discussed. As a result we will show how indexing changes the run-time and the memory-management behavior of a PROLOG emulator.



The second part begins with a short introduction to the theoretical area of indexing. The two main theoretical problems concerning indexing will be revealed. We will show that indexing is a non-trivial problem, which should be intelligently solved by heuristics.



In the third part, we present several possible indexing methods, different implementations, their advantages and disadvantages. We also enumerate the environment of this work, namely the characteristics of the implementation of the language RELFUN [7, 17, 18, 19], which provides the basis of our LISP realization.



An abstract graphical representation scheme for indexing algorithms (called index trees [23]) is introduced to compare several indexing methods and their results. In the fourth part we describe the ideas behind the RELFUN indexing method. Based on index trees produced from the RELFUN code (described in [23]), we show how to generate indexing WAM code.



Fast but not least, we demonstrate how our method is implemented and how it is used. We show a few results and compare it with other existing indexing methods such as complete indexing.

We assume the reader is familiar with PROLOG and its most well-known implementation environment, the Warren Abstract Machine, called WAM¹.

¹If this is not the case we refer the reader to the following (incomplete) list of publications: [22],[1],[11],[29],..

Part I

Introduction



ince the development of PROLOG the language has become more and more wide-spread. Kowalsky's [16] equation:

$$ALGORITHM = LOGIC + CONTROL$$

constituted a revolution in the understanding of programming languages. One philosophy of logic programming languages is to hide control inside a general inference engine. The declarative semantics of these languages allows the programmer to write down *what* shall be done and leave it to the system *how* to do it. This allows problem formulation which is both elegant and natural. But logic programming does not only have strong points: in the early days of PROLOG a lack of efficient control strategies resulted in inefficient problem-solving behavior. So PROLOG was more of a logic programming 'toy tool' than an implementation environment for serious applications. But researchers like D.H.D. Warren[29, 30], Hickey and Mudambi[14], P.van Roy[27], R.A.O'Keefe[21] A. Taylor[26, 25], ... have paved the path to PROLOG compilers now approaching the speed of C.

They use global optimization methods and native-code compilation to obtain these results. Perhaps the single most important factor of efficiency of large PROLOG programs is indexing, an optimization which can always be applied (independent of other compiler optimizations). The indexing issue is at least as old as PROLOG, because it is (like unification and backtrack control) a basic mechanism of PROLOG-like knowledge bases. However, there is not as much research in this area as in the other areas. In this paper we want to explore new techniques, bringing together DB-technology[14] and results from other WAM-based indexing schemes.

2 Indexing: What is it, Where Does it Come From?

2.1 Index Functions

A view popular among users is that PROLOG (actually, DATALOG²) is an intelligent relational database system.

This is suggested by comparing the main loop of PROLOG with the main loop of data-base languages.

In both cases, we need a function finding entries in a data base (or program) which are possible candidates to satisfy a query G . Such a function is called an *index function*. Since data-base techniques are much older (and therefore more elaborated), PROLOG could try to profit from them.

²Subset of PROLOG without compound terms (lists and structures)

```

Goal  $G$ 

1. find next clause  $H : -G_1, G_2, \dots, G_n$  with:  $H$  unifying  $G$  (or
    $\gamma(H) = \gamma(G)$  i.e.  $\gamma = mgu(H, G)$  exists).

2. try to solve  $\gamma(G_1), \dots, \gamma(G_n)$ 
   else goto 1

```

Figure 1: Main loop of PROLOG

```

Goal  $G$ 

1. find set of entries  $\{E_1, E_2, \dots, E_n\}$ 
   with:  $E_i$  matched by  $G$ .

```

Figure 2: Main loop of a simple DB-language

2.2 DB-Indexing

Most DB systems provide a set of indexing functions (based on B^* , hashing, ...) to the user. The DB administrator chooses exactly one indexing method for a specific problem. Lots of parameters (key-argument, type, ...) influence the indexing method. Each indexing function returns the set of matching answers.

Logical formulas over many key-arguments are reduced to set operators (join, diff, merge,...) with respect to the indexing scheme on single key-argument places.

Obviously, DB-indexing methods are very domain-specific and we shall see that in contrast to PROLOG, DB-indexing need not cope with problems like side-effects, recursion, the order of answers, non-DATALOG facts and non-ground facts where recursion and non-DATALOG leads to infinite answer sets. Moreover, PROLOG indexing has to be automatic or at least be applicable by an average user.

So, when transferring DB-technology to PROLOG, we are forced to look for new, specially adapted indexing schemes.

2.3 Indexing in PROLOG

A main feature of PROLOG is its nondeterministic behavior: a definition may be expressed so that there are alternative evaluation possibilities reached by backtracking.

The order of clauses and even duplicates are characteristic for the procedural semantics of programs. So PROLOG indexing functions do not return sets but sequences³ $\langle c_{i_1}, \dots, c_{i_n} \rangle$ of clauses for possible alternative answers. Consider a sequence S_G of clauses c_i . We say S_G is *correct* wrt G if all clauses which PROLOG would try successfully or with any side-effect constitute a subsequence of S_G . We also say a clause c is *indispensable* if c is in all *correct* S_G 's. Moreover we can say that S'_G is *better* than S_G if both are *correct* and $S'_G \subset S_G$.

³the order is given by the sequence of clauses in the program code; we will transfer set operators such as " \setminus " and " \subset " to sequences in the obvious manner

In other words: if part 1 of the PROLOG main loop would try, step-by-step, all clauses in a *correct* S_G , it would give all correct answers. Now, we can sometimes find out in advance that an alternative clause $c_i \in S_G$ will not succeed and have no side-effect. Then, we can hold that $S_G^+ = S_G \setminus \langle c_i \rangle$ is *better* for G than is S_G . If we can *control* the search for *indispensable* clauses so that fruitless alternatives are never tried, we will achieve a more efficient evaluation.

The reward of cutting down a priori the sequence of alternatives S_G for a goal G seems to be even more promising. If we look at the and/or-tree of a PROLOG program then S_G is the set of all the or-branches for node G . Reducing S_G is always a reduction of the search space. Many conditions can be imposed a priori on S_G .

In any case, a necessary (but fruitless since not restricting anything) condition for S_G is:

$$S_G \subseteq \langle c | c \in G \text{ is a clause of the program, in the original order}^4 \rangle$$

The most restrictive condition (but also fruitless, since this is the goal of the whole PROLOG unification process) for S_G is:

$$S_G = \langle c | c \text{ is an indispensable clause for } G \rangle$$

Such a condition could be found automatically only in DATALOG-like programs without recursion.

The simplest non-trivial condition takes the relation name into account:

$$S_G = \langle c | c \text{ is a clause from the procedure of the relation called in } G \rangle$$

The task is to come up with more restrictive conditions and methods to constrain, perhaps step by step, the sequence S_G without spending too much effort in finding these restrictions. On the other hand, the conditions must be as restrictive as possible, preventing too much unnecessary clauses which would result in backtracking. It is well-known that backtracking is a time-consuming and memory-expensive job (see also section 8.2.2).

2.4 Index Algorithms

We defined indexing, coming from DB techniques, as functions returning a sequence of potentially matching clauses. In contrast to DB techniques, in PROLOG the global flow of the program leads to correct answer substitutions, so index functions are not only called when calling a goal but also during the unification process in the body of a clause. Each branching (deterministic or non-deterministic) could be seen as performed by an *index algorithm*.

Index algorithms give a more general view for indexing as index functions do.

In the rest of the paper we prefer index functions. As we will see in a later section, they provide for a separate compilation of index code and clause code, as we need it in our implementation (see section 9).

3 What is Indexing Good For?

PROLOG provides a depth-first tree-search method to the user, so he must live with this tree-search algorithm. The PROLOG system must organize this search as efficiently as possible. Thus, we need an adequate method (like indexing) to reduce the branching attempts. But what is indexing good for? What is the real advantage if we cut down a priori the branching attempts?

On the one hand, indexing can reduce the run time of the program. If the retrieval algorithm deciding whether a clause is in S_G is fast enough then time is saved by the system not unifying alternative clauses and pruning the PROLOG and/or-tree at the right time. The earlier the sequence of alternatives can be reduced, the more efficient the system gets.

On the other hand, nondeterministic execution automatically consumes lots of storage. There must be storage space for collecting backtrack information in any PROLOG realization model. Cutting down the branching attempts a priori relieves the system from saving this information for many non-unifying clauses, since they are never tried. But if a nondeterministic execution leads to side-effects or correct answer substitutions then backtrack information must be stored, and we can only try to minimize the storage used by it. We call these optimization functions *index assistant functions* (see section 6.3). Unfortunately, indexing does not only have strong points. We also have to see the negative aspects:

1. indexing information is normally scattered all over the program. Searching and qualifying it is not a trivial task and compilation time can increase in an unbounded fashion.
2. indexing code increases program size. In addition to the code of the program, there is a need for storage of the indexing code. It can happen that the amount of indexing code is exponentially larger than the bare WAM code.

You can already see that there will be no *one best way* for indexing, because of the negative and positive points. But a balanced, user-dependent or procedure-dependent compilation can lead to a fair compromise.

This only gives a qualitative view of the “optimal” behavior of indexing. In a later part (see part IV) we discuss a more quantitative view for a special PROLOG implementation based on the WAM and on our indexing method.

Part II

Theory



We have seen that indexing changes the run-time and the memory management behavior of a PROLOG implementation. We now want to reveal the theoretical aspects of index trees and discuss the worst case performance of an “index tree without choice-point creation”. Index trees are abstract representations of index algorithms.

In this section we will speak of *uninformed index trees*.

Definition 1: uninformed index tree

An uninformed index tree is a tree whose nodes are labeled with a sequence of clause numbers:

1. the root node is labeled with $(1, \dots, n)$
2. for each inner node α with sub-nodes β_i the following holds:

$$\alpha = \bigcup_i \beta_i$$

$$\rightarrow \forall_i \beta_i \subseteq \alpha$$

Each node is labeled with the sequence of clauses S_G which must still be tried at this point. The edges represent unknown condition⁵. If one condition is satisfied we can reduce the set S_G to the node linked with this edge. The following is an example of an uninformed index tree:

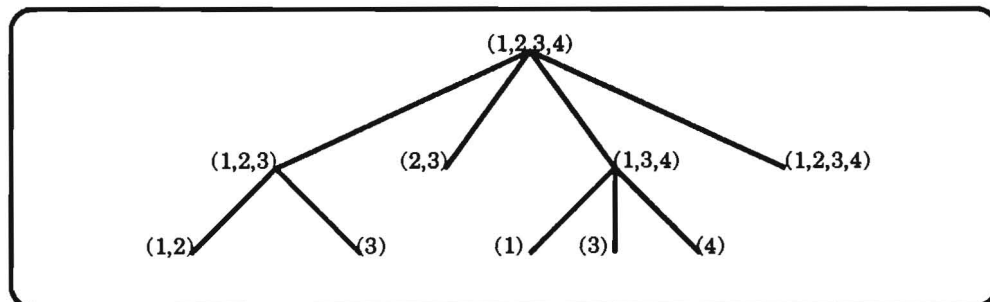


Figure 3: uninformed index tree

In this example, the root node (= no indexing is yet done) consists of four different constraints. If one of them is satisfied, we follow the corresponding edge, knowing that we only have to try the reduced set of clauses to get all possible alternatives.

⁵that is why the trees are called uninformed index trees

First, we will show that index trees can result in exponentially large code length. This is due to a worst case intermixed presentation of constants and variables in the argument positions in the head of the clauses. In this case, a set of n clauses can be partitioned by two constraints into a set of $(n - 1)$ clauses and another one with $(n - 2)$ clauses, which are used recursively to construct the child subtrees until the leaves of the index tree correspond to single clauses.

Secondly, for any reasonable definition of optimality, the problem of finding an *optimal* index tree is NP-complete [11]. This observation can be made if the index scheme provides indexing of inner structures. In this case the problem of finding a minimal subset of argument positions such that two rules do not unify in all positions of this set, can be reduced to the NP-complete set-covering problem [4].

Solutions for these problems are approximated in two different parts of our indexing method (see section 10), but this will be explained later.

4 Exponentially Large Index Trees

Consider a procedure p with n rules and $n(n - 1)/2$ parameters. We want to show that the number of nodes in the corresponding index tree can have a complexity of $O(2^n)$.

The clauses are numbered from 1 to n . Since p has n clauses, the number of pairs of disjoint clauses (i, j) with $i < j$ is

$$\sum_{x=1}^{n-1} x = n(n - 1)/2$$

Since p also has $n(n - 1)/2$ parameters, we can select a unique argument position r_{ij} for each pair of clauses (i, j) .

Assume clause i to be a fact whose k^{th} parameter a_{ik} is

1. an *anonymous* variable (denoted by “_”), if there is no j such that $r_{ij} = k$ or $r_{ji} = k$.
2. the *constant* i , if such a j exists.

It is important that for each pair of clauses (i, j) constructed in the above way, the heads will unify in each argument except for r_{ij} .

For $n = 4$ and $\langle r_{ij} \rangle = \begin{pmatrix} * & * & * & * \\ 1 & * & * & * \\ 2 & 3 & * & * \\ 4 & 5 & 6 & * \end{pmatrix}$ we will give an example:

- | | |
|----------------------------|---------------------------------|
| 1. $p(1, 1, -, 1, -, -)$. | Since $r_{1,3} = 2 \Rightarrow$ |
| 2. $p(2, -, 2, -, 2, -)$. | the second argument of |
| 3. $p(-, 3, 3, -, -, 3)$. | clause 1 is 1 and |
| 4. $p(-, -, -, 4, 4, 4)$. | the second argument of |
| | clause 3 is 3 |
| | ... |

Figure 4: Procedure with an exponentially large index tree

We now assume we already have an index tree for a definition p with n rules. The number of nodes of this index tree is s_n .

We extend the definition of p by another clause (w.l.g. we add clause 0) and by the missing $(n+1)n/2 - n(n-1)/2 = n$ argument places, filling them in the described manner. Observe that this can be done incrementally. The new index tree has the following form:

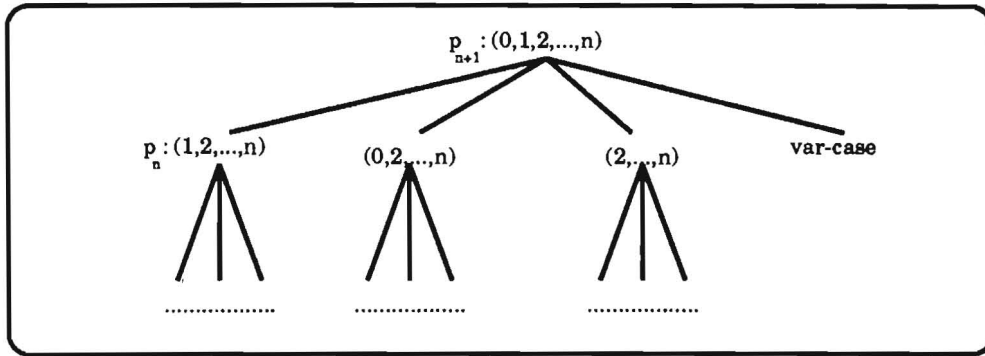


Figure 5: exponentially growing index tree

1. the root node $(0, 1, 2, \dots, n)$

→ 1 additional node

2. we must distinguish between the cases that either the input parameter for the observed argument position is instantiated or not:

- (a) if it is instantiated:

Without loss of generality, we can distinguish the first two clauses ($r_{0,1} = 1$).

Then the definition for P_{n+1} looks like this:

$P_{n+1}(0, \dots)$

$P_{n+1}(1, \dots)$

$P_{n+1}(-, \dots)$

$P_{n+1}(-, \dots)$

⋮

- i. if we the first argument is 0 the subtree has the root-node $(0, 2, \dots, n)$

→ s_{n-1} additional nodes

- ii. if we the first argument is 1 the subtree has the root-node $(1, 2, \dots, n)$
 $\rightarrow s_{n-1}$ additional nodes
- iii. else (if we the first argument is neither 0 nor 1) the subtree has the root-node $(2, \dots, n)$
 $\rightarrow s_{n-2}$ additional nodes
- (b) if it is not instantiated (we can unify all clauses):⁶
 We have a subtree with at least one node (if we stop indexing the rest)
 $\rightarrow \geq 1$ additional node

Thus, the number of nodes of the extended index tree is:

$$\Rightarrow s_{n+1} \geq \begin{cases} 2s_{n-1} + s_{n-2} + 1 & : x > 2 \\ 4 & : x = 2 \\ 1 & : x = 1 \end{cases}$$

The solution of this recurrence equation reveals the complexity of $O(2^n)$. Thus, we have an example for an exponential growth of an index tree. Since each edge in this tree represents a distinction of a set of clauses, the edge has to be compiled into at least one indexing instruction. Thus, the produced indexing code is likewise exponential with respect to the number of rules.

This result is not as discouraging as it seems since most applications do not have the rate $1 : O(n)$ between the number of rules and the number of parameters⁷. But it is discouraging enough, since even a linear growth of the code caused by indexing is not desirable with a large factor.

Note that a compiler producing an index tree in the described manner produces exponentially large index code and a compiler with no indexing only will produce linearly growing index code (one choice-point constructor for each clause). But in the second case more memory is used in run-time when the choice-points are created. Then a strongly recursive definition of a procedure can quickly exhaust the whole memory. Another point is that choice-point instructions are a waste of time, whereas the constraints are mostly implemented on a low level and therefore permit time saving.

Incidentally, indexing methods using information from mode analysis or other global information gathering systems (or from the user himself) can find a good ratio between the usage of choice-point constructions and indexing instructions. So we can conclude again: there is not *one single way*.

5 NP-Complete Index Problem

We have seen that index code can grow exponentially with respect to the number of clauses. But how can we even find a good set of constraints to reduce the set of alternative possibilities. We will show that this problem is NP-complete if we provide looking to inner structures to discriminate the clauses.

⁶in a later section this case is called the var-case \rightarrow no reduction of the set of clauses is possible with respect to this argument position

⁷Think of a procedure with 6 rules which would have to have at least 15 parameters

Consider a procedure P with n arguments. We say a set S (a subset of these n argument positions) is a *complete position set* if no two rules unify all positions in S .

The NP-hard *complete-position-set problem* is the following:

Definition 2: complete-position-set-problem

Find the smallest integer n such that there is for a head deterministic^a procedure p of size^b N a complete position set of size n .

^aA procedure p is called *head deterministic* if at most one rule of the definition of p is good for any goal with only instantiated arguments

^bA measure could be the number of characters in its ASCII representation

Such a minimal complete position set could be used to build an index tree with a minimal use of choice-point constructors.

By reducing the complete position set problem to the well known set covering problem[4], we show that the first one is at least as hard as the second one:

$$\begin{aligned} \text{Let } C_k &= \{(i, j) \mid \text{rules } i \text{ and } j \text{ differ in the } k^{\text{th}} \text{ position}\}, \\ C_0 &= \{(i, j) \mid i \text{ and } j \text{ are rules}\}, \\ S &\subseteq \{i \mid i \text{ is the number of a rule}\} \end{aligned}$$

$\Rightarrow S$ is a complete position set $\Leftrightarrow C_0 = \bigcup_{k \in S} C_k$, since two rules must differ at least in one argument position.

To show that the complete position set problem is exactly NP-complete we reverse the above reduction.

$$\begin{aligned} \text{Let } R &= \{C_1, \dots, C_m\}, \\ C_i &\in C_0, \\ |C_0| &= n(n-1)/2 \end{aligned}$$

Then, we have to construct a procedure P such that

$$C_k = \{(i, j) \mid \text{rule } i \text{ and } j \text{ differ in the } k^{\text{th}} \text{ position of the definition of } p\}$$

This is done in the following way:

The parameters of the rules are lists of size $n(n-1)/2$, constructed similar to the arguments of the example in section 4, except that if (i, j) is not an element of C_k , then all r_{ij} are set to anonymous variables. The following example helps to understand this construction:

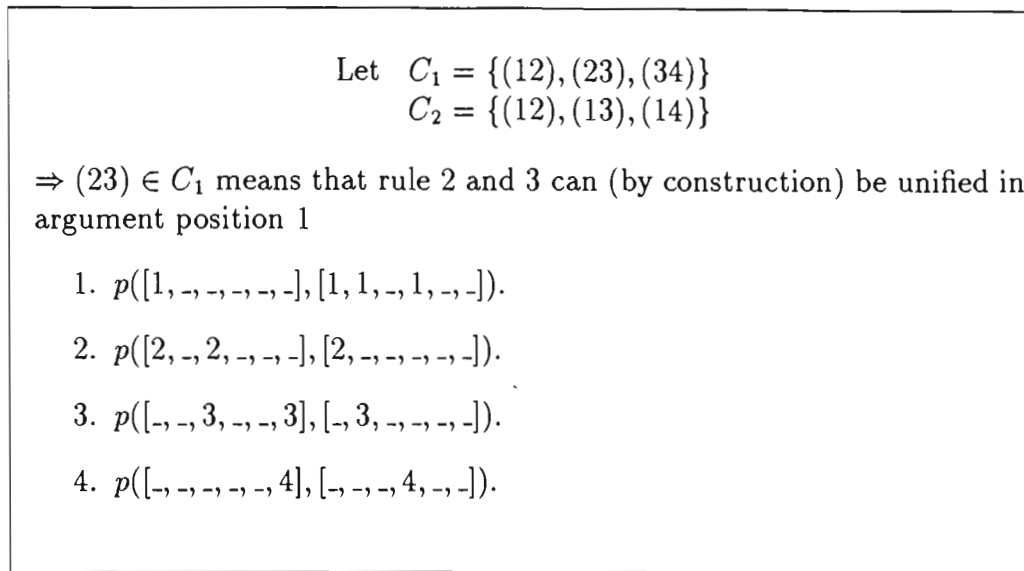


Figure 6: NP-complete index problem

If we define a procedure P in the above way, C_k holds exactly the definition.

In our indexing scheme we must cope with both problems: the exponentially growing behaviour and the NP-complete index problem.

Since we do not want to generate exponentially growing index trees we “intelligently” limit the size of an index tree depending on the kind of the program. To make this limit as flexible as possible we define several measures which constrain the size of the index tree (see section 10). To get around the NP-complete problem we use “intelligent” heuristics to approximate the solution. These heuristics are based on domain-specific knowledge about the program given by the user.

Part III

Basic State of the Art



fter this theoretical approach, we now turn to more practical things. From the beginning we always spoke about an implementation of indexing methods for PROLOG. In fact, the implementation is done for RELFUN [5, 6, 7, 17, 18, 19]. But we now want to show that we need not to distinguish between these languages if we introduce indexing methods. The RELFUN programming language is introduced as an attempt to integrate the capabilities of the relational and functional styles. We distinguish between hornish and footed clauses.

A hornish clause is a normal PROLOG Horn clause, except that its premises may contain nested function calls. Footed clauses differ syntactically from hornish ones by having an “&” in front of the last premise. The value of this last premise is the return value of a footed clause.

To show that hornish clauses correspond to a subset of footed clauses you only have to consider RELFUN’s transformation algorithm *footen*, mapping hornish clauses (in particular, PROLOG’s Horn clauses) to footed clauses:

$$\begin{aligned} \text{footen: } \quad \textit{hornish} &\rightarrow \quad \textit{footed} \\ h : -g_1, g_2, \dots, g_n &\rightarrow h : -g_1, g_2, \dots, g_n \& \textit{true} \end{aligned}$$

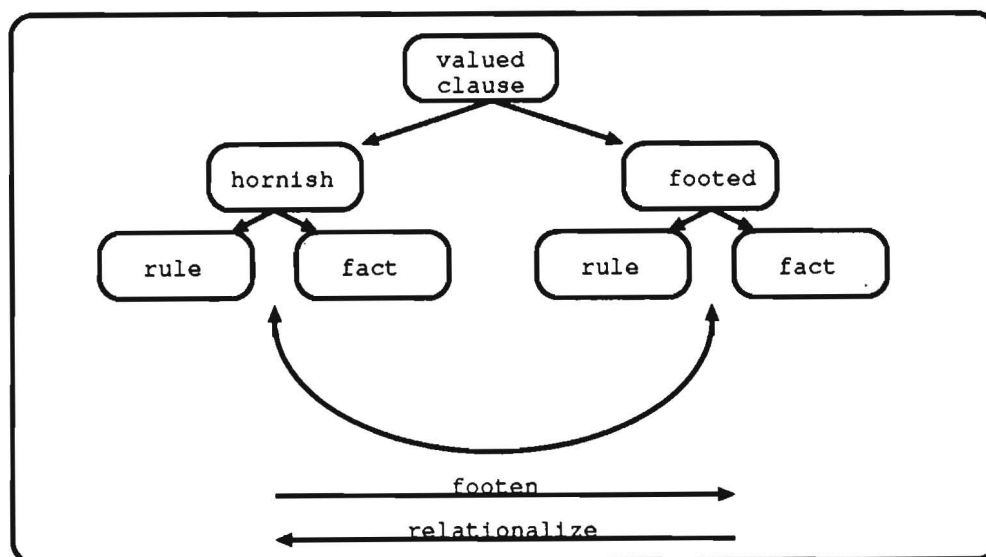


Figure 7: RELFUN’s clauses

RELFUN’s inverse transformation algorithm is called *relationalize*, which flattens nested calls and introduces an extra argument taking the return value⁸. The latter transformation shows that it suffices to consider the PROLOG subset of RELFUN when discussing indexing schemes.

⁸For other RELFUN features (higher order,...) you can find similar horizontal transformations in [5, 6, 23]

6 Looking at Other Approaches

In this subsection we provide an overview of different indexing schemes. They can be distinguished into *hardware oriented* and *software oriented* approaches.

The hardware oriented approaches are based on DB-techniques. A hash-function returns, for a given query, a sequence of clauses as potential matches. This is done separately from the program, so rules (maybe a very large number of clauses) can be stored separately (e.g. externally).

Most software oriented indexing schemes have a mixed storage of index and clause code, so the whole program must be loaded at run time.

6.1 Hardware Oriented Approaches

Several indexing methods are based on bit-matrix representation of clauses in a procedure. They are *field encoding*, *superimposed coding with embedded position and variables*, and *superimposed coding with external variables* [14].

All those are based on the principle of *n-in-m-coding*.

6.1.1 m-in-n-Coding

In this method the value of an attribute is compressed into a binary word of *width n* with a fix number of *m* bits set to 1. This number is called the *weight*. The problem is how to represent variables so that they can match with anything. In [COLOMB] the three enumerated possibilities to do this are proposed.

The main advantage of this method is that you can construct hardware that handles up to 8.000 clauses and more in the presented manner. Together with the linear searching hash-function you reach a very high efficiency. Another key property is that m-in-n-coding results in highly compressed code, so that large clause-code can be separately stored (externally) from the small index code and only single rules are loaded.

6.2 Software Oriented Approaches

In contrast to the hardware oriented approaches, the software oriented approaches do not use hash-function returning *a set* of potential matching clauses, but the program flow leads to all those clauses. That is why the index-code and the clause code are scattered over the program code.

6.2.1 General WAM-indexing

The WAM provides the user with indexing techniques that can only discriminate the first argument[29, 31], thinking that PROLOG programmer have a natural tendency to write code in data structured-directed manner.

Hassan Ait-Kaci in [1] introduced a so called *three-level-indexing scheme* using all the indexing instructions from the WAM.

First a definition of a procedure is partitioned into subsequences. Those clauses who have a variable at the first argument position are the search bottleneck and separate the subsequences from each other. The subsequences are linked with a try-chain.

The subsequences were indexed in a *three-level-indexing* manner of the form:

first level	:	discrimination on type (constant, structure, list, empty-list and variables)
second level	:	discrimination on value (only for constants and structures)
third level	:	enumeration of clauses

Figure 8: general three level indexing scheme

The WAM indexing instruction-set is:

index level	instruction	arguments
first	switch_on_term	labels to the next level index instructions for constant, structure, list, emptylist and variables (possibly more types)
second	switch_on_constant switch_on_structure	number of constants (structures) and a hashtable with a label for each constant (functor)
third	try retry trust (and/or try-me-else retry-me- else trust-me)	—

Figure 9: WAM indexing instruction set

The first and second level indexing instructions are deterministic choices. The instructions of the third level are also called choice-point constructors because of handling the backtrack mechanism in the WAM. Second level list indexing is really third level indexing on list structures, the second level being skipped by special handling of lists in the WAM.

As an example you can see the general WAM indexing code for the following program:

```

p/1:  try t1
      retry 2
      trust t2
t1:   switch-on-type const,fail,fail,fail,var1
const: switch-on-const (1,1),(2,2),fail
p(1).
p(2).
p(X).
p(s(1)):-.....⇒ struc: switch-on-struc (s/1,4),(r/1,5),fail
p(r(2)):-..... var2:  try 4
p([]):-.....    retry 5
p([X|Y]):-..... retry 6
           trust 7
1:        code-for-clause-1
2:        code-for-clause-2
.
.

```

One of the main ideas in this index scheme is to separate the index code from the rest. Therefore it can only take the head of a clause into account. Other techniques not only indexing the head but also during the unification process in the code of the body of the clause. An other source of optimization for WAM based indexing techniques is the extension of the WAM by new types and branch-instructions.

We want to describe the most famous indexing scheme which takes these two ideas into account. Other attempts are more or less comparable with it.

6.2.2 Complete Indexing

In [14] Timothy Hickey and Shyam Mudambi present several indexing techniques based on the WAM. The first one (complete indexing) uses global information (like modes) to perform indexing.

First of all the program is transformed, creating new special code for each mode that might occur for a procedure call.

As an example we look at the following program:

1. `top :- p([1,2,3,4],X), write(X).`
2. `p([],0).`
3. `p([X|Y],N) :- p(Y,M), N is M+1.`

`p` is only called with a constant argument in the first position and a variable in the second. The new code for the procedure `p` is specialized for this mode. It is represented in the procedure `p_cd`⁹. If we assume that in the program `p` is also called with other modes, the

⁹c stands for constant and d for don't know

compiler will produce other specialized procedures for these modes. The PROLOG clauses for these specialized procedures will not differ from the original ones, but the produced WAM code takes the mode information into account. The transformed code is:

1. `top :- p_cd([1,2,3,4],X), write_c(X).`
2. `p_cd([],0).`
3. `p_cd([X|Y],N) :- p_cd(Y,M), N is M+1.`

Then the clauses are transformed into a normal form:

1. `p_c...cd...d(T1,...,Tn,Z1,...,Zm):-`
2. `P1,...,Pr`
3. `Z1 = S1,...,Zm = Sm,B1,...,Bs.`

Where:

$T_i \equiv$ arguments with mode *constant*

$S_i \equiv$ argument with mode *dont know*

$Z_i \equiv$ new Variable not yet occurring in the clause

$P_i \equiv$ { Primitives: goals without side effects and whose parameters are known to be ground after head-unification

$B_i \equiv$ { either a non-primitive goal or causing side effect or with unbound arguments after head-unification.

The generated indexing code is in some sense also a three level indexing of the following form, corresponding to the normal-form:

first level	:	indexing head-code
second level	:	indexing primitive-code
third level	:	indexing body-code

Figure 10: three level complete indexing scheme

The first one is a sequentially indexing on the first n c-mode arguments. This is done by unifying the known structure of these arguments and indexing inner different possibilities with a new index-instruction called *g_switch reg table*. This new instruction assumes that the argument register *reg* contains a ground term, and switches to the appropriate location after a hash-table look up in *table*.

The indexing primitive-code contains a set of new branch-instructions implemented in the WAM (e.g *if_gt if_eq if_le*), so control jumps to a given label.

The indexing bodies are compiled with the standard WAM techniques.

index level	instruction	arguments
first	g_switch	2: argument-number and list of tuples (atom link)
second	if_gt if_eq if_ls atomic functor	2- 3: test-arguments(1-2) and true-link
third	see WAM instruction set	—

Figure 11: complete indexing instruction set

Example:

1. `merge_ccd(L, [], L).`
2. `merge_ccd([], [B|Bs], [B|Bs]).`
3. `merge_ccd([A|As], [B|Bs], [A|Cs]) :- A <= B,`
`merge_ccd(As, [B|Bs], Cs).`
4. `merge_ccd([A|As], [B|Bs], [B|Cs]) :- A >= B,`
`merge_ccd([A|As], Bs, Cs).`

Normal-form:

1. `merge_ccd(L, [], X1) :- L=X1.`
2. `merge_ccd([], [B|Bs], X1) :- X1=[B|Bs].`
3. `merge_ccd([A|As], [B|Bs], X1) :- A <= B, X1=[A|Cs],`
`merge_ccd(As, [B|Bs], Cs).`
4. `merge_ccd([A|As], [B|Bs], X1) :- A >= B, X1=[B|Cs],`
`merge_ccd([A|As], Bs, Cs).`

Index tree:

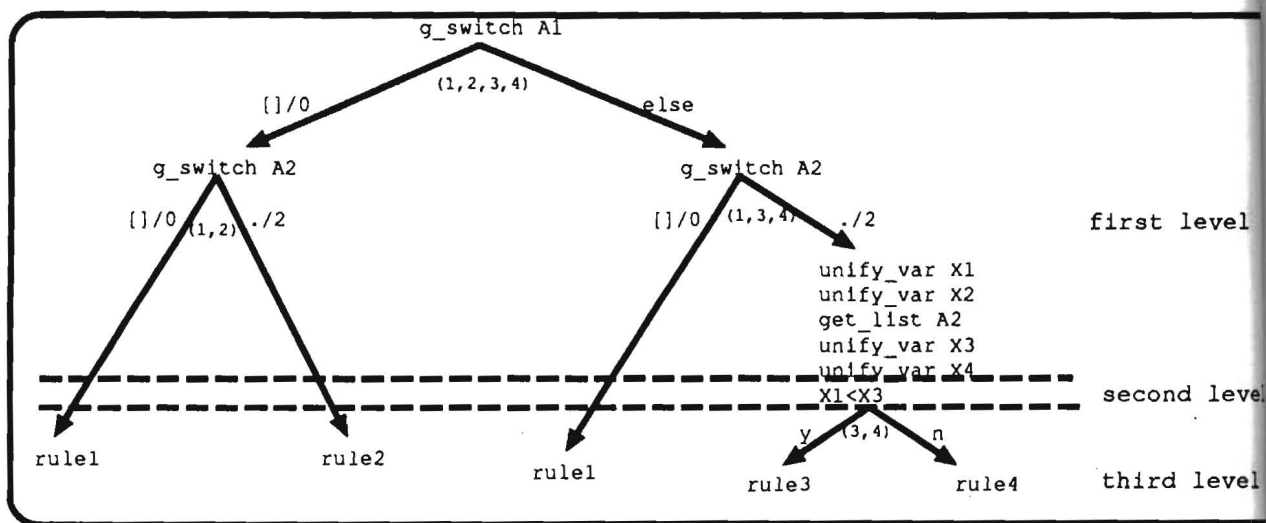


Figure 12: merge-complete index tree

In this index tree we have added the constraints to the edges, so we have now no longer an uninformed index tree but an (informed) index tree. Furthermore, we have added instructions to the nodes which must be executed if we reach the corresponding node.

6.3 Index Assistant Functions

Indexing can also be performed by some functions not changing the program flow but optimizing the time and memory consumption of the index algorithm. We want to separate these algorithms from the *pure* indexing scheme and call them *index assistant functions*.

6.3.1 Shallow Backtracking

This approach is adapt to the complete indexing algorithm, only performing the backtracking method of primitive deterministic¹⁰ procedures. The idea behind this method is the following:

While unification of the head index code and the primitive index code takes place, only a link to the next alternative clause is needed as backtrack-information because no heap variables will be bound, nor will any nonprimitive goal in the body be called, and no side effect will occur. On the other hand, after successful unification of the head and the primitives no backtracking in this procedure is possible because the only possible matching clause is selected.

This reduces the code space requirements at run-time, but good global analyzing methods are needed to detect primitive deterministic procedures.

6.3.2 Quadratic Indexing

An other approach performing primitive deterministic procedures is the quadratic indexing scheme. A tree-sharing method reduces the nodes in an index tree to have a size at most $O(n^2)$. The index tree is transformed into a directed acyclic graph (DAG).

7 Developmental Environment

We have seen that there are several methods to perform and implement indexing. In this project we tried to bring together most positive aspects of the above approaches. But doing this we also had to respect the global structure of our already existing developmental environment.

Our work is embedded in the ARC-TEC and RELFUN/RFM projects. RELFUN is a PROLOG-like language with functional extensions implemented in COMMON LISP.

At the beginning of our work, the state of the RELFUN project was the following:

¹⁰primitive deterministic is a extended definition of head deterministic which looks not only to the head of all clauses but also to the primitive index-instructions

7.1 Global RELFUN Project Structure

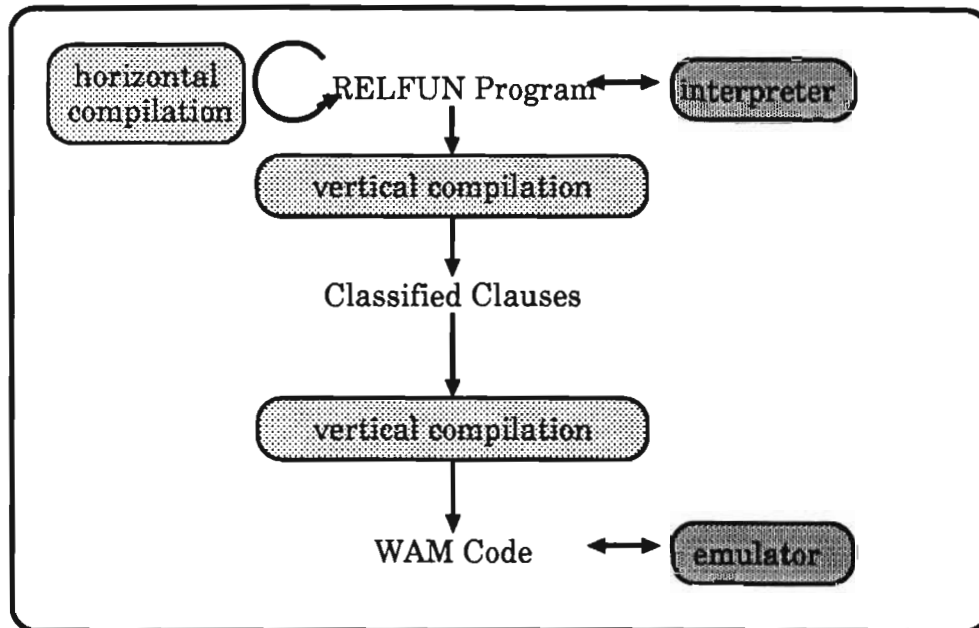


Figure 13: global RELFUN structure

The compilation task is divided into several horizontal¹¹ and vertical¹² compilation steps. The reason for this is that we prefer to do most of the compilation work at source level (rather than at code level) in order to be independent from a special low-level language or machine structure as much as possible.

Another detail in our compiler is a special language between the RELFUN language and the low-level WAM-code. This language, called “classified clauses”, was developed by Harold Boley and Thomas Krause [17, 18, 19] and is based on a tagged PROLOG-in-LISP syntax, extended with global and local information.

The right place to collect all indexing information which is necessary for our indexing scheme is surely this intermediate language. So one modification had to take place in the first vertical compilation step between the RELFUN program and the classified clauses.

Another modification had to generate the indexing WAM code and thus had to take place in the second vertical compilation step between the classified clauses and the WAM code.

Finally, the emulator had to be changed a little bit to allow new (better) indexing methods. Our emulator is based on the ν -WAM ([22]), a LISP implementation of the WAM ([29]), good for rapid prototyping and experimental extensions. It was changed for handling RELFUN’s functional extensions by Hans-Günther Hein (see [12]).

¹¹source to source

¹²source to code

Part IV

Implementation

8 A Partitioned Implementation



he previous section has shown that in the compilation environment of RELFUN, it is the best way to divide the implementation of an indexing method into at least two parts. In figure 14 the cut line between the two working areas for the implementation parts is given.

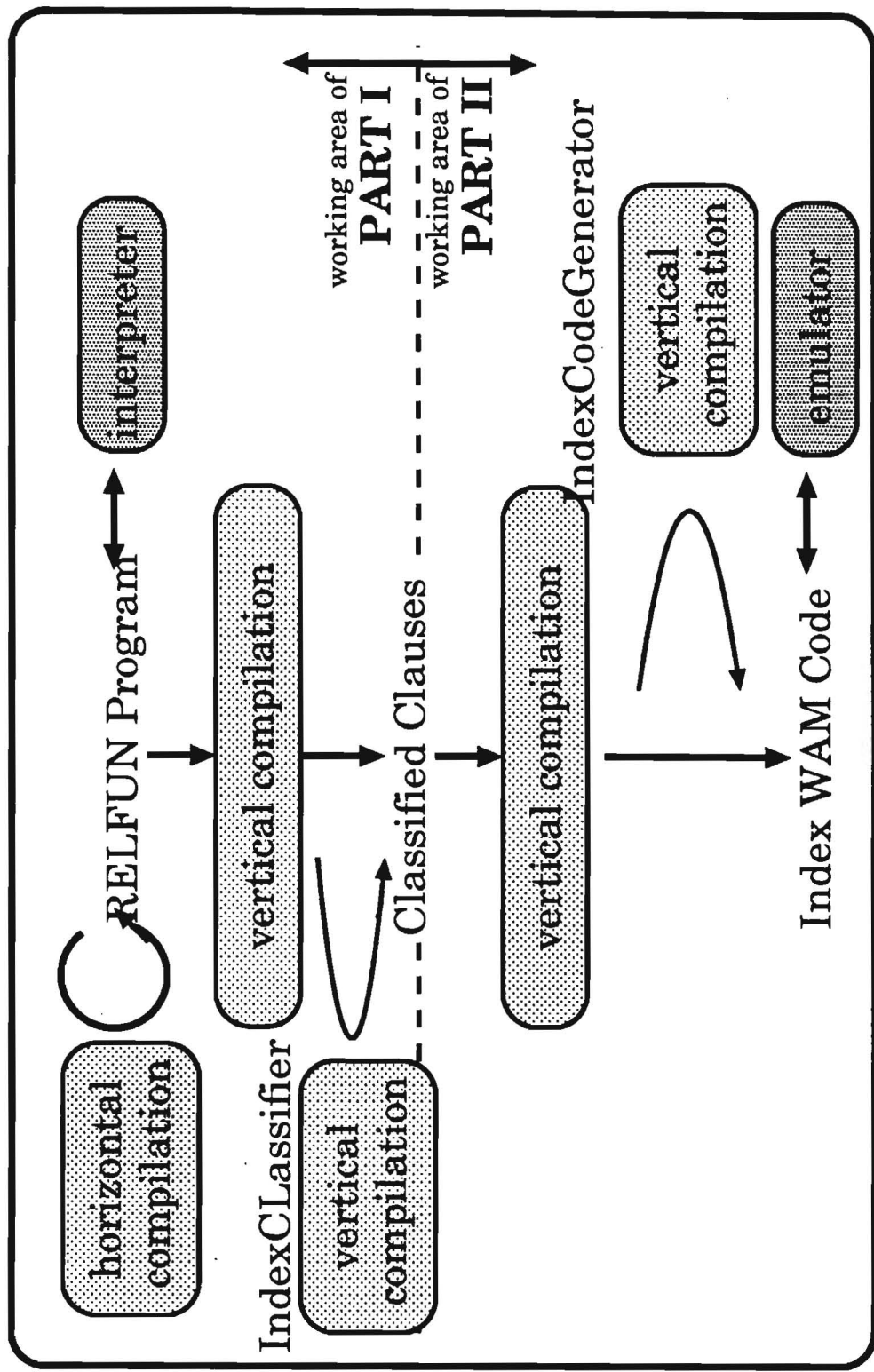


Figure 14: RELFUN structure with indexing

8.1 First Part

The first implementation area is placed on a high level (between RELFUN and the classified clauses).

In [23] Michael Sintek explains his ideas on how to extend the classified clauses with indexing information. In his paper he also proposes the idea of transforming higher-order predicates (resp. functions) on this high level, with respect to indexing handling.

Since for the rest of this paper only indexing is looked at, we can concentrate the introduction of the classified clauses only to the indexing part¹³.

In this paper we also do not want to explain how we get the indexing information from the RELFUN program. Since the general indexing problem is NP-complete, we use specialized heuristics approximating the solution. If the reader is interested in this aspect, we refer him to the paper of Michael Sintek [23].

8.2 Second Part

The second part of the implementation is working below the level of the classified clauses. Its main task is the generation of indexing WAM-code from the indexing information of the classified clauses. Since the standard WAM only permits us to index the first argument, another modification extends the emulator with a special index-register. We already said that one of our philosophies is to be independent from a special low-level language, thus these last modifications must be as small as possible¹⁴. In spite of this fact we developed a general indexing method, able to handle even special features such as higher-order predicates and domain specific compilation.

Before the introduction of our ideas and implementations, we first want to refresh (resp. introduce) the two languages involved in this vertical compilation step: the classified clauses and the WAM instructions.

8.2.1 The Classified Clauses (indexing part)

As a result of our approaches in implementing new indexing techniques in the RELFUN compiler we had to extend the classified clauses by new index information.

- EBNF for classified clauses - indexing part

```
<indexing> ::= (indexing [ <iblock> ] )
```

```
<iblock> ::= <pblock> | <sblock>
```

```
<pblock> ::= (pblock <rblock> { <sblock> | <iblock> }+ )
```

¹³for more detail see [17] and [7]

¹⁴only one new register and one new instruction is added to the ν -WAM

```

<rblock> ::= (rblock <clauses> { arg-col }+ )
<clauses> ::= (clauses { <clause-number> }+ )
<arg-col> ::= (arg <arg-number> { <base-type> }+ )
<base-type> ::= <const> | <struct> | <var>
<const> ::= (const <symbol>)
<struct> ::= (struct <symbol> <arity>)
<var> ::= (var <symbol>)
<iblock> ::= (iblock <clauses> { arg-col }+ )
<sblock> ::= (sblock <rblock> <seqind> [ <pblock> ] )
<seqind> ::= (seqind { <seqind-arg> }+ )
<seqind-arg> ::= (arg <arg-number>
                  (info <inhomogeneity>)
                  <constants>
                  <structures>
                  <lists>
                  <empty-lists>
                  [ <others> ])
<constants> ::= (const { <element> }* )
<structures> ::= (struct { <element> }* )
<element> ::= ( <element-name> <clauses> [ <iblock> ] )
<element-name> ::= <symbol> | ( <symbol> <arity> )
<lists> ::= (list <clauses> [ <iblock> ] )
<empty-lists> ::= (nil <clauses> [ <iblock> ] )
<others> ::= (other <clauses> [ <iblock> ] )
<clause-number> ::= 1|2|3|4|5|6|7...

```

- Explanations:

- iblock = indexed block

- pblock = partitioned block
- sblock = standard index block
- lblock = block consisting of only one clause
- rblock = raw block containing the initial data
- seqind = sequential indexing
- arg-col = argument column
- others = (possibly indexed) clauses for elements not occurring in any hash table

Since we have not yet presented our index method it is not possible at this point to understand the full meaning of the index part. But the reader already familiar with the WAM index scheme can immediately recognize some well known features (e.g. an “sblock” is more or less the standard WAM “switch on type” instruction).

8.2.2 The (ν -)WAM

The WAM is an instruction set and storage model for the efficient execution of PROLOG, developed by D.H.D. Warren[29]. A short description of the WAM storage model will be given here, rather than a precise definition of the instruction semantics [1]. The ν -WAM [22] is a LISP implementation of the WAM, useful for rapid prototyping and experimental extensions. We use a version of the ν -WAM by Hans Günther Hein [12] (called the RFM WAM) that can handle value returning for RELFUN’s footed clauses. However, since indexing is not influenced by these extensions, we can restrict the following treatment to the original WAM.

The WAM storage model consists of the following primary areas:

1. the local stack, contains environment and choice-point frames
2. the heap, stores data structures created by unification
3. the trail, holds bound variables to be unbound during backtracking
4. the code array, stores the WAM code

Various state registers to manage the storage areas and a set of argument registers for passing parameters and calculating temporary results make the storage model complete.

The WAM registers are the following:

Register	Contents
<i>E</i>	Current environment
<i>P</i>	Current instruction pointer
<i>B</i>	Current choice-point
<i>CP</i>	Continuation instruction pointer
<i>H</i>	Heap pointer
<i>S</i>	Heap structure pointer
<i>HB</i>	Heap backtrack pointer
<i>TR</i>	Trail pointer
<i>X_n</i>	Argument registers

Figure 15: WAM registers

Now let's have a look to the storage areas:

The *heap* is primarily for storing compound data structures. In the *trail* addresses of bindings are stored, for unbinding upon backtracking. The most interesting area for indexing is the *stack*. It holds two types of variable-length frames: *environments* and *choice-points*.

The environments hold both local and bookkeeping information. A choice point holds arguments passed to a nondeterministic procedure and backtrack information (*E, B, H, CP, P, TR*). Both frame-types were linked by different chains.

Remember the main-loop of PROLOG:

For a given goal, we select (by indexing) a sequence of clauses to try. If they are nondeterministic, i.e. if we cannot narrow down the field of possible matching clauses to one, a choice-point is created and loaded with the register and local information.

Now we can imagine the gain that indexing can give us:

Each trying of a clause costs the creation of at least one choice-point with at least 6 pointers and each returning from a failing call must remove these. In addition to this, a call normally implies many unifications and compounded data structures (and may be other calls) until the failing point is reached. Indexing prevents the machine from such fruitless work.

The ν -WAM instruction set only provides general WAM-indexing (shown in section 2.5.2)

9 Separate Compilation: Indexing-Code, Clause-Code

Our goal was to enhance RELFUN, based on the just finished compiler, rather than writing a new compiler. This led us to the idea to separate the compilation of the indexing code from the compilation of the clause code. With this technique we are also able to recompile and store separately the two code types¹⁵. Additionally we can now switch on and off this indexing method just by skipping the function for generating the indexing code. This could be necessary if we want to compare the behavior of the different codes (e.g. for debugging and tuning). The emulator has been changed to run both code types.

¹⁵large clause code could be stored externally so that only a small indexing code would be left in main memory

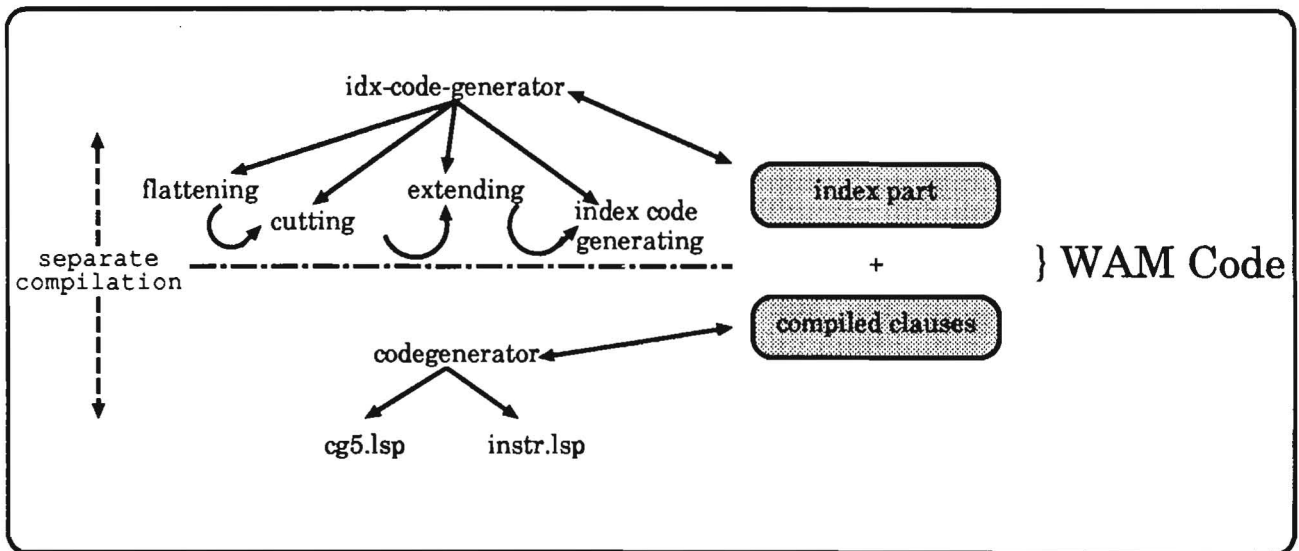


Figure 16: separate compilation

10 Idea

Our first task was the introduction of a graphical representation for general indexing methods. This allows us to discuss the advantages and disadvantages of different methods without implementing them. An index tree is an abstract representation for a special index algorithm. It describes the procedural semantics of such an algorithm.

10.1 Index Trees

10.1.1 General Informed Index Trees

Definition 3: g-i-index tree

A general-informed-index tree (g-i-index tree) is a tree with labeled nodes:

1. try-nodes (circles)
2. constraint-nodes (boxes)
3. clause nodes and fail-nodes

A clause node either contains a sequence of machine instructions or a label to a sequence of machine instructions. All outgoing edges of a constraint node are labeled with constraints.

In the previous section we described why we prefer separate compilation of index code and clause code. Therefore we can specialize the definition of g-i-index trees to *header informed index trees* (or *h-i-index trees*) only describing index functions (see section 2.3).

This definition guaranties the possibility of a separate compilation and storage model for index and clause code.

10.1.2 Header Informed Index Trees

Definition 4: h-i-index tree

An h-i-index tree is a g-i-index tree with: clause node only contains the number of the corresponding clause; no inner node is a clause node and all leaves are clause nodes; each constraint node must have a special node with a so-called "var" edge, which is satisfied in all cases not satisfied before^a.

^aother restrictions could be: each input must be satisfied by at least one edge or each constraint node must have a special so-called "else" edge satisfying all inputs, which are not satisfied by another edge

Based on the WAM, we only have the following nodes and edges in our h-i-index tree:

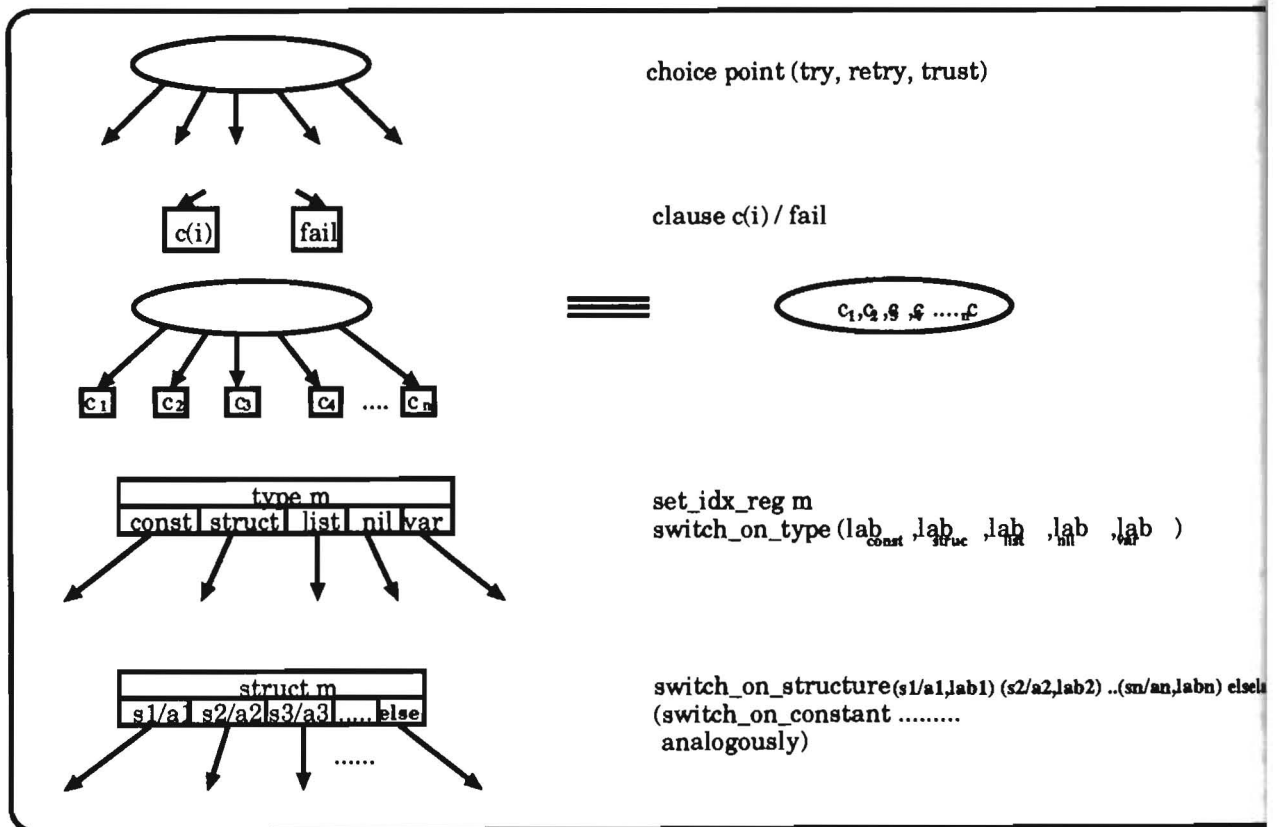


Figure 17: graphical representation & corresponding instructions

10.2 Horizontal Compilation Scheme

Another representation scheme for index functions uses a PROLOG-like notation with auxiliary predicates and extralogicals like `cut (!)`, `bound`, `constantp`, ... The advantage of this notation is that “intelligent” compilers can compile this code into a normal WAM (using only first argument indexing) and thus having nearly the same results as with an extended WAM (using multi-argument indexing) and a multi-argument indexing method.

Figure 17 contrasts the WAM instructions with the corresponding h-i-index tree node types with the edge types and the PROLOG-like expression for the semantics of the constraints.

nodes	edges	constraints
type-nodes (type m)		
	constant edge	(bound arg_m) (constantp arg_m)
	structure edge	(bound arg_m) (structurep arg_m)
	list edge	(bound arg_m) (listp arg_m)
	nil edge	(bound arg_m) (nilp arg_m)
	variable edge	(unbound arg_m)
structure-nodes (struc m)		
	func/arity-edge	(eq (functor arg_m) func) (eq (arity arg_m) arity)
constant-nodes (const m)		
	const-edge	(eq arg_m const)

Figure 18: constraints

Definition 5: flow-path through an h-i-index tree

A flow-path through an h-i-index tree is built as follows:
Begin with the root node. If it is a

1. try-node: go through each subtree sequentially (normally from left to right)
2. constraint-node: go through the subtree linked by the first satisfied edge
3. clause-node: this is always a leaf (the labeled clause is called)

Definition 6: valid h-i-index tree

An h-i-index tree is called valid if it corresponds to a definition of a predicate (this means if a PROLOG machine whose index function follows the flow-path through the h-i-index tree is sound and complete with respect to PROLOG semantics).

This definition allows us to index using a valid h-i-index tree without any loss of the PROLOG semantics, but tells us nothing about the efficiency of the h-i-index tree.

Definition 7: depth of a flow-path

The depth of a flow-path through an h-i-index tree is the number of occurrences of try-nodes not following a “var” edge.

Since it makes no sense to link two try nodes (they can always be merged together), the depth of a flow-path corresponds to the number of arguments which constrain the set of alternative evaluation possibilities reached by backtracking.

Definition 8: depth of an h-i-index tree

The depth of an h-i-index tree is the maximal depth of a flow-path through the h-i-index tree.

Definition 9: breadth of a flow-path

The breadth of a flow-path through the h-i-index tree is the number of constrain nodes following only the “va.” edges.

In contrast to the depth of a flow-path, the breadth corresponds to the number of arguments which are tried to index for until the first succeeds.

Since in normal WAM implementations the instructions for the two constraint nodes (*switch-on-constant* and *switch-on-structure*) have no *var* edge¹⁶, we had to extend the definition of these instructions.

Definition 10: breadth of an h-i-index tree

The breadth of an h-i-index tree is the maximal breadth of a flow-path through the h-i-index tree.

We now still need definitions which give us a quantitative measure for the cost of h-i-index trees. The first definition (*chw*) gives us a measure for the costs of building choice-points at run-time if we use a special h-i-index tree¹⁷:

¹⁶they fail if the constant (or structure) is not found in the hash-table

¹⁷This definition also holds if we want to measure the costs of building choice-points at run-time following one special flow-path in the h-i-index tree. In this case, “max” must be substituted by “first satisfied constraint”

Definition 11: chw

The choice-weight (chw) of an h-i-index tree is defined as follows:

$$\begin{aligned} chw([c_i]) &= 0 \\ chw(\text{try-circle}_{t_1, \dots, t_n}) &= chw(t_1) + \dots + chw(t_n) + n \\ chw(\text{switch-box}_{t_1, \dots, t_n}) &= \max(chw(t_i)) \end{aligned}$$

Another definition is needed to measure the memory costs for an h-i-index tree:

Definition 12: cow

The code-weight (cow) of an h-i-index tree is defined as follows:

$$\begin{aligned} cow([c_i]) &= 0 \\ cow(\text{try-circle}_{t_1, \dots, t_n}) &= cow(t_1) + \dots + cow(t_n) + n \\ cow(\text{switch-box}_{t_1, \dots, t_n}) &= cow(t_1) + \dots + cow(t_n) + 1 \end{aligned}$$

10.3 Example

Throughout the rest of this paper, we will consider the following simple 6-fact procedure (the line numbers are only for use in the index graph):

```
1: f(1,30).
2: f(2,10).
3: f(1,20).
%-----
4: f(X,50).
%-----
5: f(4,70).
6: f(1,80).
```

We think it is simple enough to permit an overview of the code; at the same time it is hard enough to show all indexing features and the main ideas. In PROLOG, variables always need special handling, so in indexing too. Therefore we divide the program into partitions, separated by those clauses with variables in one fixed argument column. We will later see that this “partition-rule” can be weakened, allowing only a maximal number of variables in a partition block. This led us to a new definition:

Definition 13: block-variable-size

The block-variable-size of a procedure is the maximal number of variables allowed in a constant block.

Up to now the RELFUN compiler transformed the above program without producing special indexing code, only trying sequentially all clauses with a try-chain. It should be mentioned that if all arguments in a goal are unbound then there is no better way than doing this¹⁸. The index graph looks like this:

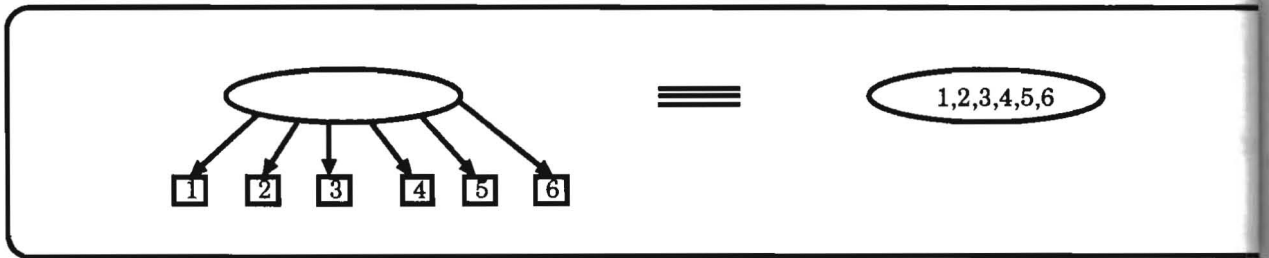


Figure 19: sample h-i-index tree: no indexing

The WAM indexing code is the following:

```
try 1
retry 2
retry 3
retry 4
retry 5
trust 6
```

From now on we always want to show first a “horizontally compiled” PROLOG-like presentation¹⁹ for an indexing method (see section 10.2), then show the h-i-index tree (see section 10.1) and finally the WAM index code. We hope the reader will be able to understand an h-i-index tree without further explanation, to transform it into WAM code and even to see that this method is easy to extend by other features.

10.4 Standard Indexing

To illustrate the graphical representation, we start this subsection with standard WAM indexing, introduced in section 6.2.1.

We just have said that the handling of variables in the area of indexing is not unique. In a first trial we separate the partitions by a try-chain. Thus each partition is either a single clause with a variable at the indexing position, or a set of clauses with only constants at the indexing position²⁰. In this case the block-variable-size is set to zero. Another possibility is to allow variables in a constant partition. We will see that in this case we must push down the clauses with variables in the indexing argument position in each leaf of the original h-i-index tree of the constant-partition.

But now we want to have a look at the advantages and disadvantages of the first possibility. One can easily verify the PROLOG-like presentation (the original source is shown above):

¹⁸indexing will have no effect

¹⁹a general representation scheme for index functions only applicable for cut-less programs

²⁰so second-level-indexing does not need a *var* link

```

1: f(X,Y):- f01(X,Y).           % first partition: clauses 1..3
2: f(X,50).                   % second partition: clause 4
3: f(X,Y):- f02(X,Y).         % third partition: clauses 5, 6

4: f01(X,Y):- bound(X),!,f1(X,Y). % indexing possible
5: f01(1,30).                 % no indexing
6: f01(2,10).
7: f01(1,20).

9: f02(X,Y):- bound(X),!,f2(X,Y). % indexing possible
10: f02(4,70).                % no indexing
11: f02(1,80).

12: f1(X,Y):- constantp(X),!,f3(X,Y).% constant constraint
                                     % all other fail

13: f2(X,Y):- constantp(X),!,f4(X,Y).

14: f3(1,Y):-!,f5(Y).
15: f3(2,10):-!.

16: f4(1,80):-!.
17: f4(4,70):-!.

18: f5(30).
19: f5(20).

```

The predicate

- *f* branches into the three partitions.
- *f01* is first-level-indexing including the var-case for the first partition.
- *f02* is first-level-indexing including the var-case for the second partition.
- *f1* second-level-indexing for the first partition (only constants are possible).
- *f2* second-level-indexing for the second partition (only constants are possible).
- *f5* third-level-indexing for the first partition in case of constant 1 at the indexing argument-position.

Now, let us have a look at the graphical representation of the index function; we will easily find a corresponding node for each predicate.

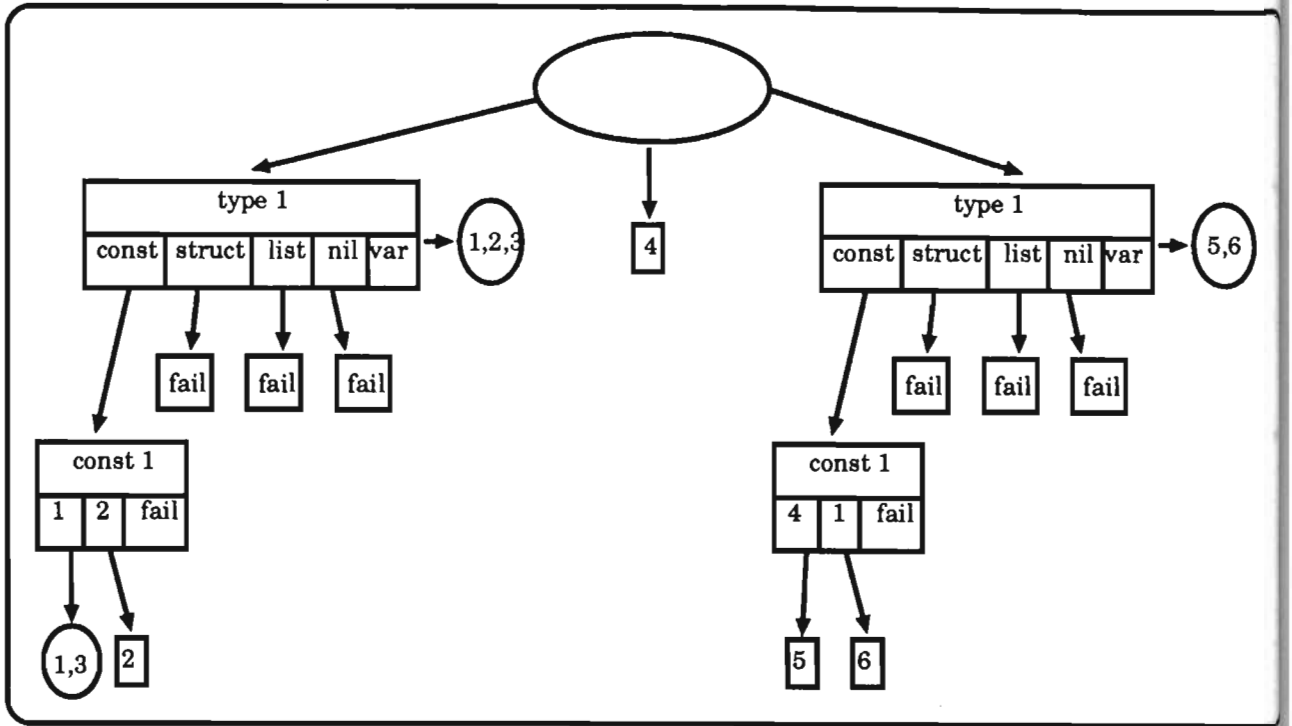


Figure 20: sample h-i-index tree: first argument indexing, partitions

We now want to discuss the h-i-index tree from a more quantitative point of view. The two functions *chw* and *cow* allow such a quantitative statement:

$$chw(idx) = 8$$

$$cow(idx) = 14$$

You certainly cannot conclude something for the whole class of h-i-index trees from this single h-i-index tree. But what happens if the data base grows? Since we allow no variables in a constant partition²¹, the worst case we can assume is to divide a large constant partition inserting a new clause with a variable at the index position. If we do so the subtree for the constant partition is split in the middle and the new clause is inserted. We get two new try-subtrees: one switching to the variable partition and a second switching to one part of the split constant partition.

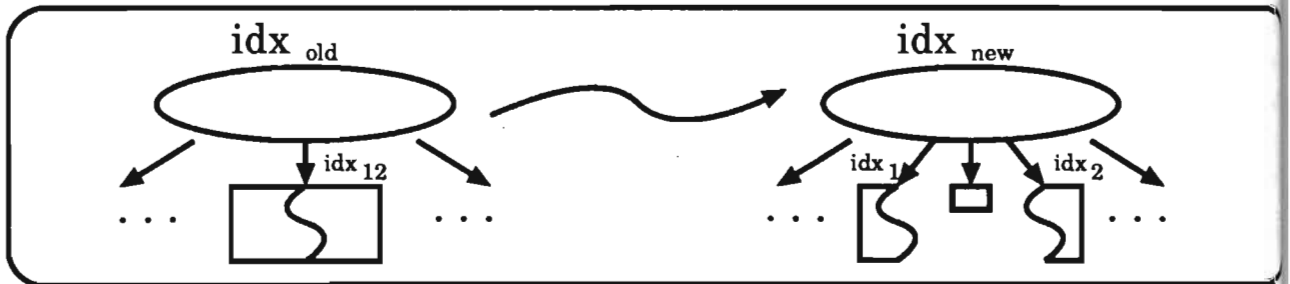


Figure 21: block-variable-size = 0

²¹the block-variable-size is set to zero

It is easy to see that chw of a type subtree is the number of choice-point constructors in the variable case²². If we split a constant partition in such a way that each new partition contains at least two clauses than we can verify that

$$chw(idx_1) + chw(idx_2) = chw(idx_{12})$$

So only the two choice-point constructors in the main try-chain increment the value of chw by 2.

$$\rightarrow chw(idx_{new}) \leq 2 + chw(idx_{old})$$

For cow we can state that the two new choice-point constructors again will cost code but in addition the constraint boxes in the new try subtree will too. For the same reasons as above the number of choice-point constructors does not grow in the split type subtrees. So we get:

$$cow(idx_1) + cow(idx_2) \leq cow(idx_{12}) + \text{no_of_constraint_boxes}$$

For the new h-i-index tree:

$$\rightarrow cow(idx_{new}) \geq 2 + cow(idx_{old}) + \text{number_of_constrains_in_split_idxtree}$$

We get similar results under the assumption of inserting a new constant partition.

We can state that in the worst case

$$chw(idx_{neu}) = 2 + chw(idx_{old})$$

$$cow(idx_{neu}) = 2 + cow(idx_{old}) + \text{number_of_constrains_in_split_idxtree}$$

Another possible indexing method is to propagate the variable partition into each possible subtree. In our example this means that the variable partition is involved in the two constant partitions and we only have one mixed partition:

```

1: f(X,Y):- bound(X), !, f1(X,Y).      % only one partition
2: f(1,30).                            % no indexing possible
3: f(2,10).
4: f(1,20).
5: f(X,50).
6: f(4,70).
7: f(1,80).

8: f1(X,Y):- constantp(X), !, f2(X,Y). % constant constraint
9: f1(X,50):- !.                        % all other -> variable partition

10: f2(1,Y):- !, f3(Y).
11: f2(2,Y):- !, f4(Y).
12: f2(4,Y):- !, f5(Y).
12: f2(X,50):-!.

```

²²this is the worst case for indexing

```

13: f3(30).
14: f3(20).
15: f3(50).                % variable partition included
16: f3(80).

17: f4(10).
18: f4(50).                % variable partition included

19: f5(50).                % variable partition included
20: f5(70).

```

In the h-i-index tree we can see how the variable partition is propagated into each leaf and how the two constant partitions are merged.

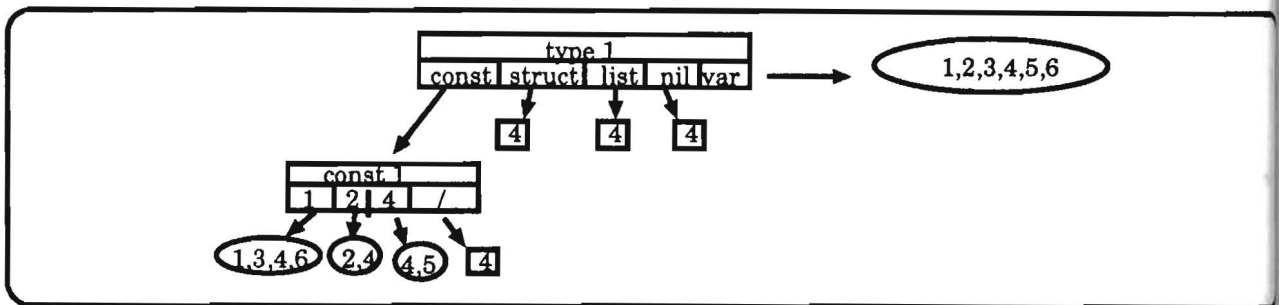


Figure 22: sample h-i-index tree: first argument indexing, no partitions

Again, we want to look at *chw* and *cow*. First we can note that:

$$chw(idx) = 6$$

and

$$cow(idx) = 16$$

But how do they vary if we add new clauses.

Since we now have only one partition²³ the try-chains are only in the leaves of the h-i-index tree and so:

$$chw(idx) = \text{number of clauses indexed by this tree}^{24}$$

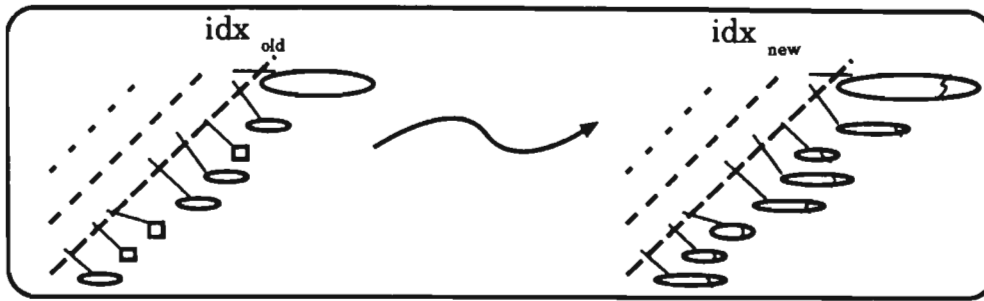
Thus

$$chw(idx_{new}) = 1 + chw(idx_{old})$$

cow does not behave as benevolent as *chw*. Assume we add a new clause with a variable in the index position. Since this clause must be propagated down to each leaf of the h-i-index tree, the code for trying this clause could occur many times in the h-i-index tree. If we have a large h-i-index tree with many different constants and structures, the h-i-index tree may have a lot of leaves and each leaf will be extended by at least one choice-point constructor.

²³the variable-block-size is infinite

²⁴this holds since in the var-case each clause must be tried

Figure 23: $\text{block-variable-size} = \text{max}$

So we get the following result:

$$\text{cow}(\text{idx}_{\text{new}}) \geq m + \text{chw}(\text{idx}_{\text{old}})$$

with

$$m = \begin{cases} \text{number of constants, struc-} \\ \text{tures, lists and nils at an ar-} \\ \text{gument position in the h-i-} \\ \text{index tree} \\ + \\ 1 \text{ (var-case ...)} \end{cases}$$

We could claim that in many cases (e.g. data bases), the choice weight in the first indexing method may grow faster than in the second one; on the other hand, the code weight behaves the other way around. One could believe that there is a trade off between chw and cow in both index methods. There exist benchmarks for both techniques where one method behaves better (concerning chw and/or cow) than the other. In our index scheme we take this fact into account by permitting the user to define whether to optimize chw or cow .

Since up to now we only switch on the first argument, the programmer is forced to choose a first argument which is good to index on. On the one hand many people think that this is no restriction²⁵, but we think there are data bases which have a natural order of arguments, which should not be changed. On the other hand it cannot be difficult in any WAM emulator implementation to change the behavior of the switch-instructions to switch on an argument register other than the first one. We will see that in the case of the ν -WAM this is really a simple thing to do.

10.5 Improved Indexing I (not only first argument)

So a first step was to allow indexing on other arguments than the first. In our example the second argument is much more discriminative than the first one. In this case the procedure is even deterministic if the mode for the second argument is constant²⁶. Our indexing method uses neither modes nor any other global information beyond the procedure level, for two reasons:

²⁵programmers should write in a data-base like manner: the first argument should be an approximation of a key-argument

²⁶note: this holds not for other modes

1. global analysers make a closed world assumption for the call-modes of all predicates. This means predicates can only be called in the way they are used in the program. Another possibility is to produce, for each mode, a specially compiled definition in addition to the normally compiled definition. However, in this case the code grows too fast.
2. non-trivial global analysers are compile-time expensive. Most of them have a complexity of $O(2^n)$ since they must find fix-points of abstractly interpreted programs.

Since we have no database-global information we can favor the *most discriminative*²⁷ argument position to index on. In our case we want to switch on the second argument.

This can be done by a simple horizontal transformation on the PROLOG-level, pushing the index-argument to the first position:

```

1: f(X,Y):- bound(Y), !, f2(Y,X).      % index second argument
2: f(1,30).                            % no indexing possible
3: f(2,10).
4: f(1,20).
5: f(X,50).
6: f(4,70).
7: f(1,80).

8: f2(Y,X):- constantp(Y), !, f3(Y,X).

9: f3(30,1):- !.                       % deterministic choice
10: f3(10,2):- !.
11: f3(20,1):- !.
12: f3(50,X):- !.
13: f3(70,4):- !.
14: f3(80,1):- !.

```

If the compilation task is done well, only one register transfer (argument-register 1 \leftrightarrow argument-register 2) and an additional execute-instruction is needed.

We only load the index register (in the standard WAM the first argument register) with another register. Instead of this, we could also define the index-register as a link to another argument-register²⁸. In this case we need no new predicate with only a new order of arguments, but a new instruction (like *set-idx-reg 2*) tells the emulator to switch for the second argument instead of the first one.

Now the index graph looks like this:

²⁷the reader may think of any reasonable definition of “most discriminative” or have a look at a paper written by Michael Sintek [23]

²⁸another advantage of doing so is that we also could link this index-register with inner structures of an argument, so we are more flexible.

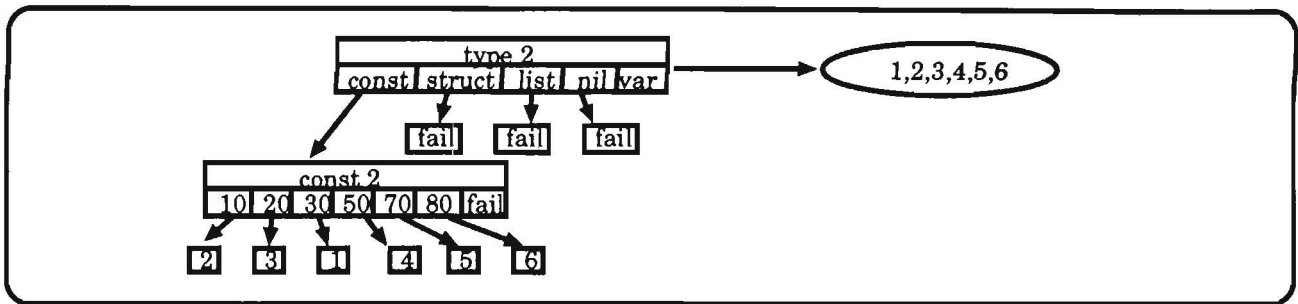


Figure 24: sample h-i-index tree: second argument indexing

In case of mode “constant” for the second argument no choice-point will be created. But we have seen that we cannot assume that any special argument is a good choice to index on²⁹, so it could happen that this index technique has no effect (if the mode for the index argument is always variable).

Once we can control the register on which we index, we can sequentially index all arguments. In our example:

- first try to index the second argument
- if this argument is instantiated then we can deterministically choose the solution
- if the second argument is a variable then (instead of trying all clauses) index on the first argument
- only if this argument is a variable too, try all clauses in a try-chain³⁰

We can simulate a sequential indexing method in the following “horizontal” way:

```

1: f(X,Y):- bound(Y), !, f2(Y,X). % index second argument
2: f(X,Y):- bound(X), !, f1(X,Y). % index first argument
3: f(1,30). % no indexing
4: f(2,10).
5: f(1,20).
6: f(X,50).
7: f(4,70).
8: f(1,80).

9: f1(X,Y):- constantp(X), !, f4(X,Y). % first argument index function

10: f4(1,Y):- !, f5(Y).
11: f4(2,Y):- !, f6(Y).
12: f4(4,Y):- !, f7(Y).

13: f5(30). % no deterministic choice
14: f5(20).

```

²⁹even if it would be in average

³⁰in this case there is no better way

```

15: f5(50).
16: f5(80).

17: f6(10).
18: f6(50).

19: f7(50).
20: f7(70).

21: f2(Y,X):- constantp(Y), !, f3(Y,X). ;second argument index function

22: f3(30,1):- !. ; deterministic choice
23: f3(10,2):- !.
24: f3(20,2):- !.
25: f3(50,X):- !.
26: f3(70,4):- !.
27: f3(80,1):- !.

```

In case of using the new WAM instruction *set_idx_reg*, the tree reduces to the following h-i-index tree (the h-i-index trees of Section 10.4 and 10.5 are just linked together):

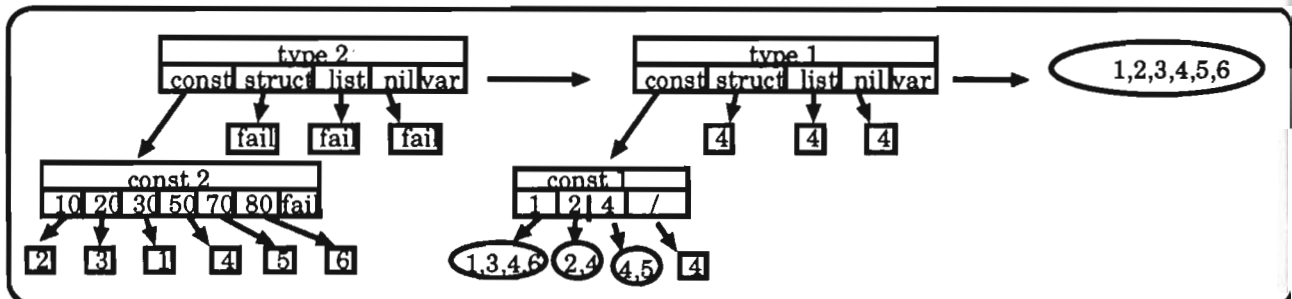


Figure 25: sample h-i-index tree: one of two arguments indexed

The multiple occurrences of a clause i only links to one compiled clause-code-version³¹. With this technique we index at most one argument: the first one which is deduced to be bound. For example, the query $f(2,X)$ is first tried to be indexed on the second (more discriminative) argument, which cannot be indexed, and then on the first one, on which we switch.

In our example this is one of the best h-i-index trees to approximate $S_{G_{opt}}$. None of our WAM-based constraints can be applied to get a better approximation (since indexing the second argument first results in a deterministic branch). But assume we know that the first argument position is instantiated in 90% of the queries and the second only in 30%. Then it would be clever to index first the first argument and then the second one. We then have the problem that the first argument is not discriminative enough to give a deterministic result. Only indexing additionally the second argument could result in a deterministic pruning of the search space.

³¹because of the separate compilation of index code and clause code, no code will be repeatedly produced

10.6 Improved Indexing II (not only one argument)

We first try to index the first argument position with a maximal partition³². If we assume to have a 1 at this position, then S_G is limited to the sequence $\langle 1, 3, 4, 6 \rangle$ so the choice is not yet deterministic. We know that in 30% of the queries the second argument is also instantiated. So why not index this smaller set of clauses for the second argument position. This method is called *multi-argument indexing*.

The following PROLOG-like representation will introduce the multi-argument indexing method.

```

1: f(X,Y):- bound(X), !, f1(X,Y).
2: f(1,30).                               % no indexing
3: f(2,10).
4: f(1,20).
5: f(X,50).
6: f(4,70).
7: f(1,80).

8: f1(X,Y):- constantp(X), !, f2(X,Y). % first-argument indexing
9: f1(X,50).

11: f2(1,Y):- !, f3(Y).
12: f2(2,Y):- !, f4(Y).
13: f2(4,Y):- !, f5(Y).
14: f2(X,50):- !.

15: f3(Y):- bound(Y), !, f6(Y). % second argument indexing
16: f3(30).
17: f3(20).
18: f3(50).
19: f3(80).

20: f4(Y):- bound(Y), !, f7(Y). % second argument indexing
21: f4(10).
22: f4(50).

23: f5(Y):- bound(Y), !, f8(Y).
24: f5(50).
25: f5(70).

26: f6(Y):- constantp(Y), !, f7(Y).

27: f7(30):- !.
28: f7(20):- !.
29: f7(50):- !.

```

³²the *block-variable-size* is maximal

30: f7(80):- !.

.

.

.

We want to explain this method by following the evaluation of the query $f(1, 50)$. First we detect that the first argument is bound, so we can start indexing on it. The cut signals that $f1$ finds all solutions if the first argument is bound. The search space is pruned down to the clauses $\langle 1, 3, 4, 6 \rangle$ (realize that the variable partition is merged in). The call for the binding of the second argument could find the solution deterministically (only one clause binds Y to 50).

Now we want to transform this PROLOG-like representation, step-by-step, into an h-i-index tree.

The first clause contains the constraint $\text{bound}(X)$. This means that we want to switch if the first argument is bound. The graph for this constraint is a type box with the argument 1. The variable edge of the type box is linked with the h-i-index tree indexing all clauses knowing that the first argument is unbound. Then we stop indexing and try sequentially all clauses. This is represented by a try circle containing the clauses 1 to 6.

The function called after we have detected that the first argument is bound, first tests the constant constraint, which is represented as a constant box linked with the constant edge of the type box. All other edges from the type box only try the variable partition (clause 4). The constant box is divided into a number of new boxes containing possible constants and a dummy box called `else`³³box. Each clause of the definition of the constant case corresponds to a part of the constant box. The `else` box takes the variable partition. Last but not least, if the choice is not yet deterministic, we call an index function indexing another argument with the reduced sequence of branching attempts. This is done the same way as above.

We only follow the case that the first argument is the constant 1. The other cases are built the same way. The difference between $f3$ and $f7$ is that in the case of definition $f3$ a unification is needed for argument binding and in the case of definition $f7$ only a matching process takes place. So in the first case the solution is not deterministic but in the second case it is (see the cut). In the h-i-index tree this is represented in the first case by a try chain (not deterministic) and in the second case by a switch-box (deterministic).

The following h-i-index tree is the described one:

³³the else box is the var case of the constraint box

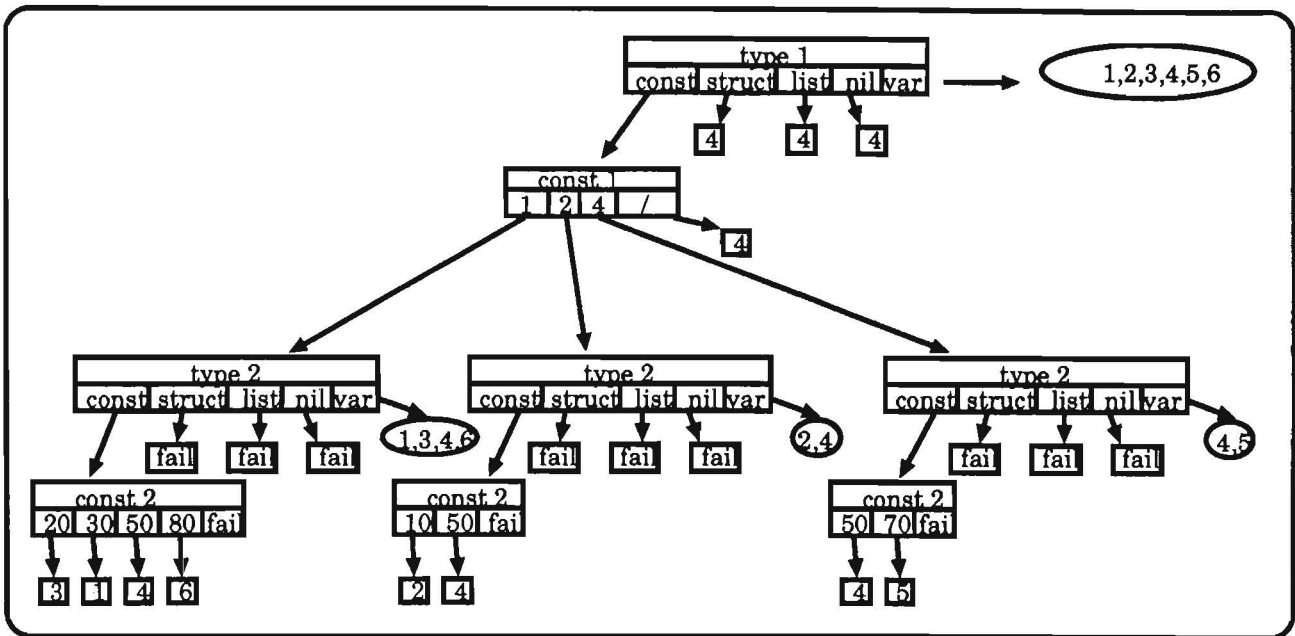


Figure 26: sample h-i-index tree: two arguments indexed

To finish the example we want to show the full h-i-index tree. It is a combination of figures 24 and 26, a sequential and multi-argument indexing method. The PROLOG-like representation now gets too large, but we think it is no longer needed for understanding.

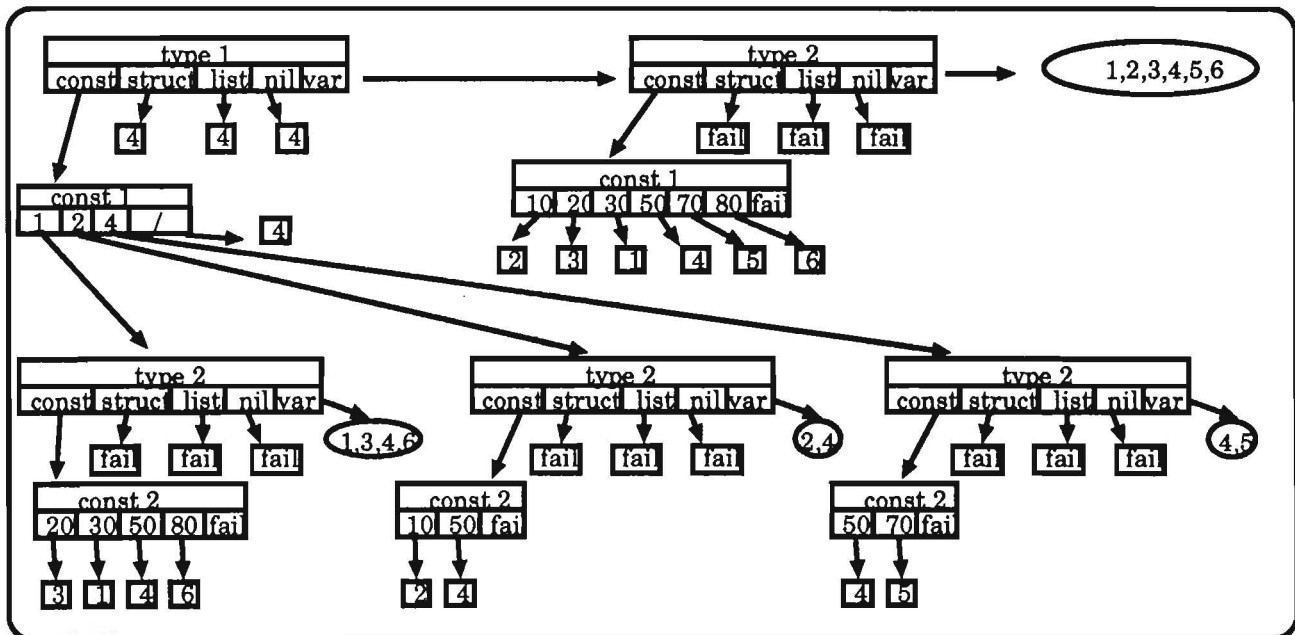


Figure 27: sample h-i-index tree: fully indexed

In section 4 we have seen that fully indexed predicates can have an exponentially large h-i-index tree. Three different methods try to limit the h-i-index tree in an “intelligent” way.

11 RFM Indexing

The most difficult method uses the *block variable size*. In our experimental applications in the ARC-TEC project we have found for each *block variable size* programs which are optimally³⁴ compiled. We have found no, trivial interdependencies between *chw* and *cow* and the *block variable size*. So we cannot give a simple remedy how to set the *block variable size* to get a more time efficient or more memory-space efficient code. This depends on many still unknown factors.

The second method to reduce the size of the h-i-index trees is the well known tree sharing method. The gain of this method depends very much on the structure of the program and the original h-i-index tree.

The last method is an “intelligent” limitation of *breadth* and *depth* of the h-i-index tree. This means neither to set *breadth* and *depth* to zero³⁵ nor to set them to infinity³⁶ but to choose a value in between in order to reduce the code size and optimize the run time efficiency as much as possible

We have found out that in real applications there are domain specific values for *breadth* and *depth* (between 2 and 4) to reach such a behavior for h-i-index trees.

The result of our approach is always to index as many arguments as possible in a heuristic deduced order and a user defined size. The user can manipulate the result with three system variables which limit the block-variable-size, the breadth and the depth of the h-i-index tree. So the user is able to control the ratio between *cow* and *chw*.

11.1 The Way of Compiling Index Code

We have just seen that the problem of compiling index code for PROLOG-like languages can be divided into two parts.

First of all we extract index information from the data base and insert it in the classified clauses. This is described in [23].

This code must be compiled into WAM code in several steps:

- create a full h-i-index tree using all index information. The result is a h-i-index tree with maximal breadth and depth.
- flatten the h-i-index tree and remove equal subtrees (tree-sharing).
- cut the h-i-index tree with respect to the user defined breadth and depth.
- expand the cut h-i-index tree with missing try-chains.
- translate the h-i-index tree into WAM code.

³⁴optimal w.r.t. time and space

³⁵then the size of the h-i-index tree is minimal but the run-time efficiency is worst case

³⁶then the index tree size is maximal but the run-time efficiency is optimal

11.1.1 Creating the Index Tree

This function only transforms the still PROLOG-like code (the classified clauses) into more WAM-like code. The transformation is very simple and can be described as follows³⁷:

- A `pblock` is transformed into a try-trust-list
- A `iblock` is transformed into its corresponding clause-label
- An `sblock` is transformed into an h-i-index tree with the information of the argument for which we index, the constant and structure subtrees and the list, nil and var subtrees. List, nil and var subtrees are normal h-i-index trees, whereas constant and structure subtrees are linked with h-i-index trees, labeled with the constant (or structure) on which we index.
- An `rblock` is not indexed at all, so we generate a try-trust-list as for `pblocks`.

The disadvantage of this syntax is that we cannot share inner index trees.

11.1.2 Flattening the Index Tree

Therefore we flat the h-i-index tree. Each subtree is substituted by a label. The labels are chosen in such a way that the same h-i-index trees have equal labels. Additionally, the set of possible alternative evaluated clauses of an h-i-index tree (S_G) is coded in its label.

We now can simply remove multiple occurrences of the same subtrees.

11.1.3 Cutting the Index Tree

The next step to reduce the size of the h-i-index tree is done by cutting the index tree at the user defined breadth and depth. Therefore we follow each flow-path through the h-i-index tree counting breadth and depth of each reached node and copy it into a new list if it is in the defined range.

If we use this method it is possible that the resulting h-i-index tree has a breadth or depth larger than defined. This is due to the tree sharing method. We allow to share h-i-index trees on different levels. Assume we have an h-i-index tree of the following form (shadowed circles represent the same h-i-index trees):

³⁷For more details in the underlying syntax and semantics for the index part of the classified clauses see [23]

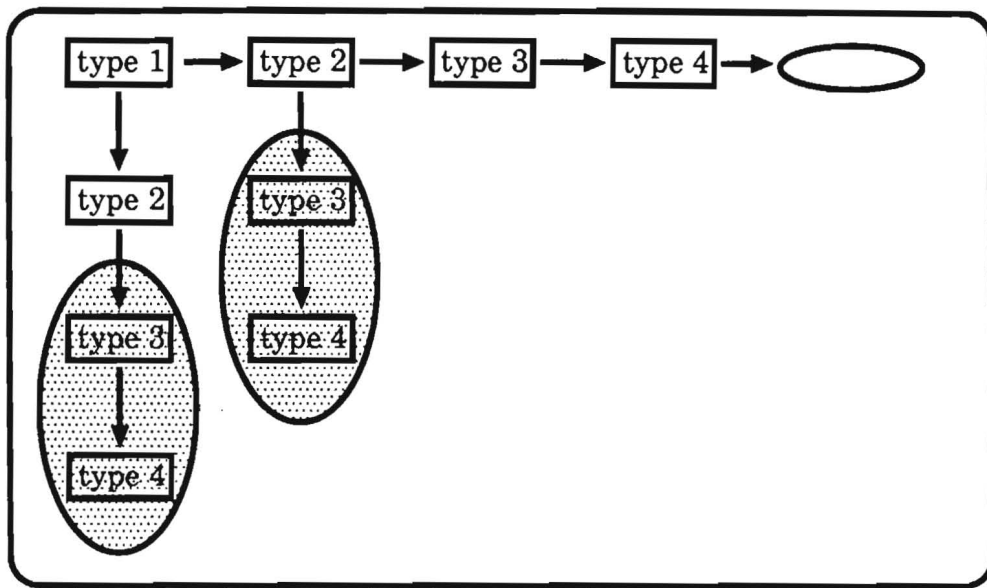


Figure 28: tree-sharing

Cutting the h-i-index tree at depth 3 without linking subtrees on different levels results in the following:

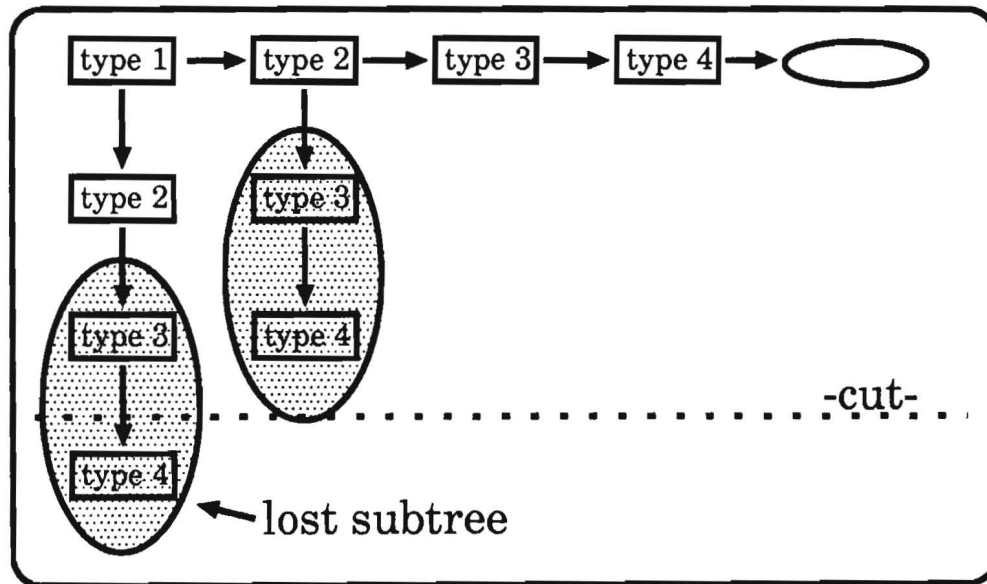


Figure 29: cut h-i-index tree without tree-sharing

Allowing links on different levels gives us:

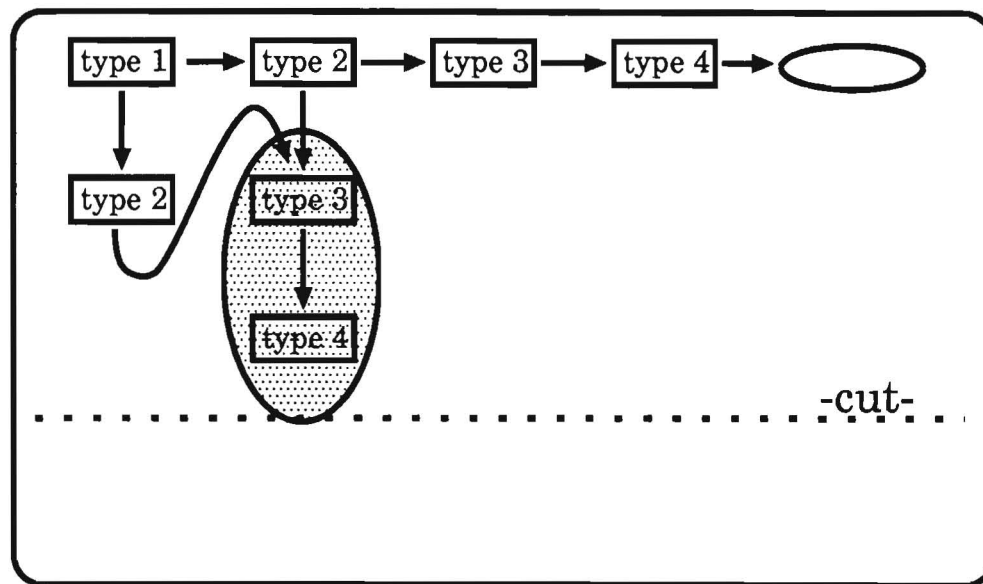


Figure 30: cut h-i-index tree with tree-sharing

In the second case we need less code but still get a more efficient program.

11.1.4 Expanding the Index Tree

Since the cut h-i-index tree is only a copy of the original tree without those branches out of range, it is possible that some subtrees are lost. Now we can use the information about the alternative clauses of the index trees coded in the labels to generate try-chains for the lost subtrees.

12 Sample Session

In order to show all index features of the compiler we now want to introduce a larger example and the solutions after each compilation step.

The example is the dnf-procedure which produces the disjunctive normal form of a logic formula with the operators 'and', 'or' and 'not'.

We begin our example with the PROLOG program of dnf and its indexing header in the classified clauses:

```
dnf(X, X) :- literal(X).
dnf(o[X, Y], o[X, Y]) :- literal(X), literal(Y).
dnf(a[X, Y], a[X, Y]) :- literal(X), literal(Y).
dnf(n[n[X]], W) :- dnf(X, W).
dnf(n[o[X, Y]], W) :- dnf(a[n[X], n[Y]], W).
dnf(n[a[X, Y]], W) :- dnf(o[n[X], n[Y]], W).
dnf(o[X, Y], W) :- dnf(X, X1), dnf(Y, Y1), norm(o[X1, Y1], W).
dnf(a[X, Y], a[a[X1, X2], Y]) :- literal(Y), dnf(X, a[X1, X2]).
dnf(a[X, Y], a[a[Y1, Y2], X]) :- literal(X), dnf(Y, a[Y1, Y2]).
```

```
dnf(a[X, Y], W) :- dnf(X, a[X1, X2]),
                   dnf(Y, a[Y1, Y2]),
                   norm(a[a[X1, X2], a[Y1, Y2]], W).
dnf(a[X, Y], W) :- dnf(X, o[X1, X2]),
                   dnf(Y, Y1),
                   dnf(o[a[X1, Y1], a[X2, Y1]], W).
dnf(a[X, Y], W) :- dnf(X, X1),
                   dnf(Y, o[Y1, Y2]),
                   dnf(o[a[X1, Y1], a[X1, Y2]], W).
```

classified clauses (only index part):

```

(proc
  dnf/2
  12
  (indexing
    (sblock
      (rblock
        (clauses 1 2 3 4 5 6 7
          8 9 10 11 12)
        (arg
          1
          (var x)
          (struct o 2)
          (struct a 2)
          (struct n 1)
          (struct n 1)
          (struct n 1)
          (struct o 2)
          (struct a 2)
          (struct a 2)
          (struct a 2)
          (struct a 2)
          (struct a 2) )
        (arg
          2
          (var x)
          (struct o 2)
          (struct a 2)
          (var w)
          (var w)
          (var w)
          (var w)
          (struct a 2)
          (struct a 2)
          (var w)
          (var w)
          (var w) ) )
        (seqind
          (arg
            1
            (info 3)
            (const)
            (struct
              ((o 2)
                (clauses 1 2 7)
                (sblock
                  (rblock (clauses 1 2 7)

```

```

          (list)
          (nil)
          (other (clauses 1 7))))))
      ((a 2)
        (clauses 1 3 8 9 10 11 12)
        (sblock
          (rblock
            (clauses 1 3 8 9 10 11 12)
            (arg
              2
              (var x)
              (struct a 2)
              (struct a 2)
              (struct a 2)
              (var w)
              (var w)
              (var w) ) )
            (seqind
              (arg
                2
                (info 1)
                (const)
                (struct ((a 2) (clauses 1 3 8 9
10 11 12)))
                (list)
                (nil)
                (other (clauses 1 10 11 12))))))
          ((n 1)
            (clauses 1 4 5 6)
            (pblock
              (rblock (clauses 1 4 5 6)

```

```

          (arg
            2
            (var x)
            (var w)
            (var w)
            (var w) ) )
          (1block (clauses 1) (arg 2 (var x)))
          (1block (clauses 4) (arg 2 (var w)))
          (1block (clauses 5) (arg 2 (var w)))
          (1block (clauses 6) (arg 2 (var w))))))
          (list)
          (nil)
          (other (clauses 1)) )
        (arg
          2
          (info 2)
          (const)
          (struct
            ((o 2) (clauses 1 2 4 5 6 7 10 11 12))
            ((a 2) (clauses 1 3 4 5 6 7 8 9 10 11 12)) )
          (list)
          (nil)
          (other (clauses 1 4 5 6 7 10 11 12)) ) ) ) )
      (fun*den
        .....
      )))
  (clauses 1 2 7)))

```

The compiler switches have the following values:

```
indexing on :min-clauses 2 :max-vars 10 :max-depth 1 :max-args 2 :debug off
```

The `min-clauses` switch is the minimal number of clauses which are tried for indexing by the compiler. The `max-vars` switch is the variable-block-size. `max-depth` and `max-args` are the compiler switches for the depth (and breadth) variables of the compiler. Last but not least the first switch (with no name) is for allowing indexing or disallowing it.

In the following we abbreviate the constraints of the type-box in the index tree: `c` is the constant constraint, `str` is the structure constraint, `l` is the list constraint, `n` is the nil constraint and the else constraint is the link on the right side of the box (without name).

The index tree corresponding to the index header of the classified clauses is of the following form:

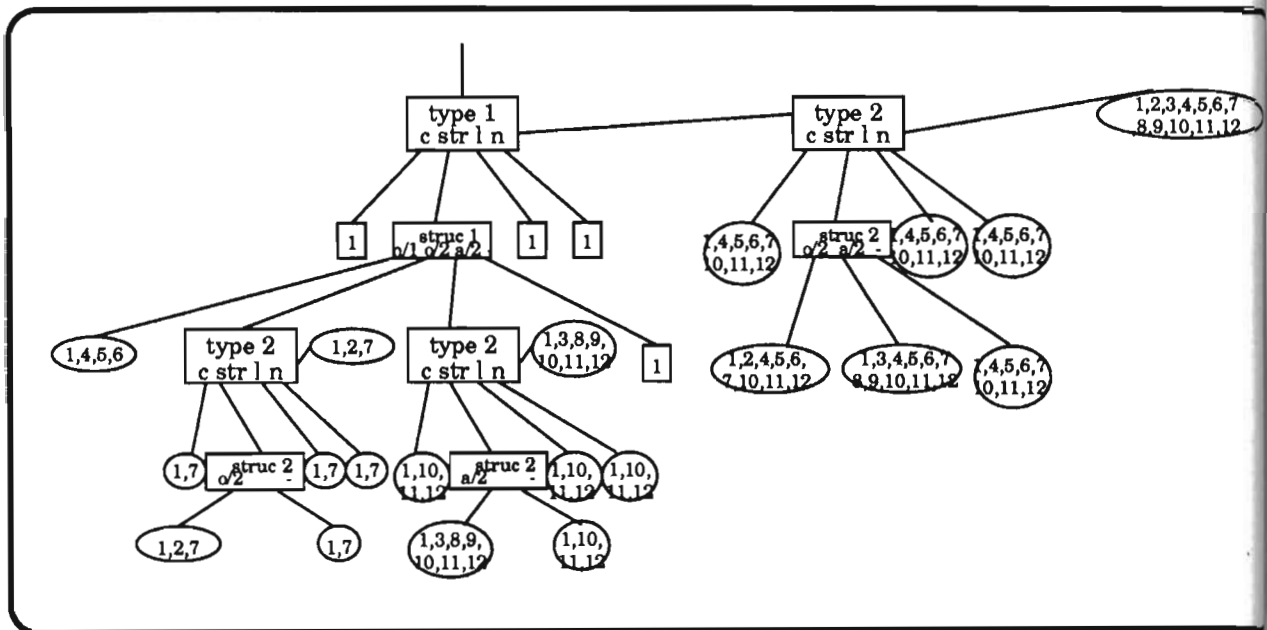


Figure 31: nested index tree

After flattening the index tree, multiple occurrences of same subtrees (like try-chain [1,2,7]) are now unique. Each subtree is a newly named index tree and links to subtrees refer to it only by name.

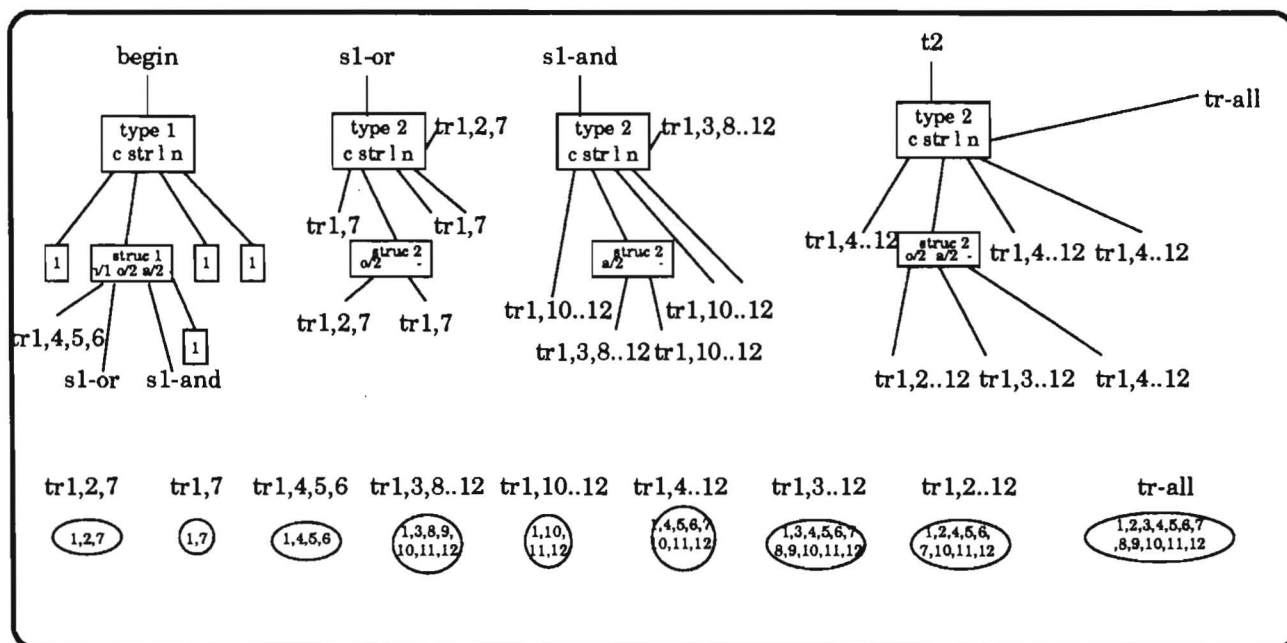


Figure 32: flattened index tree

Since the depth switch is set to 1 and the breadth switch is set to 2, the cut index tree looks like this:

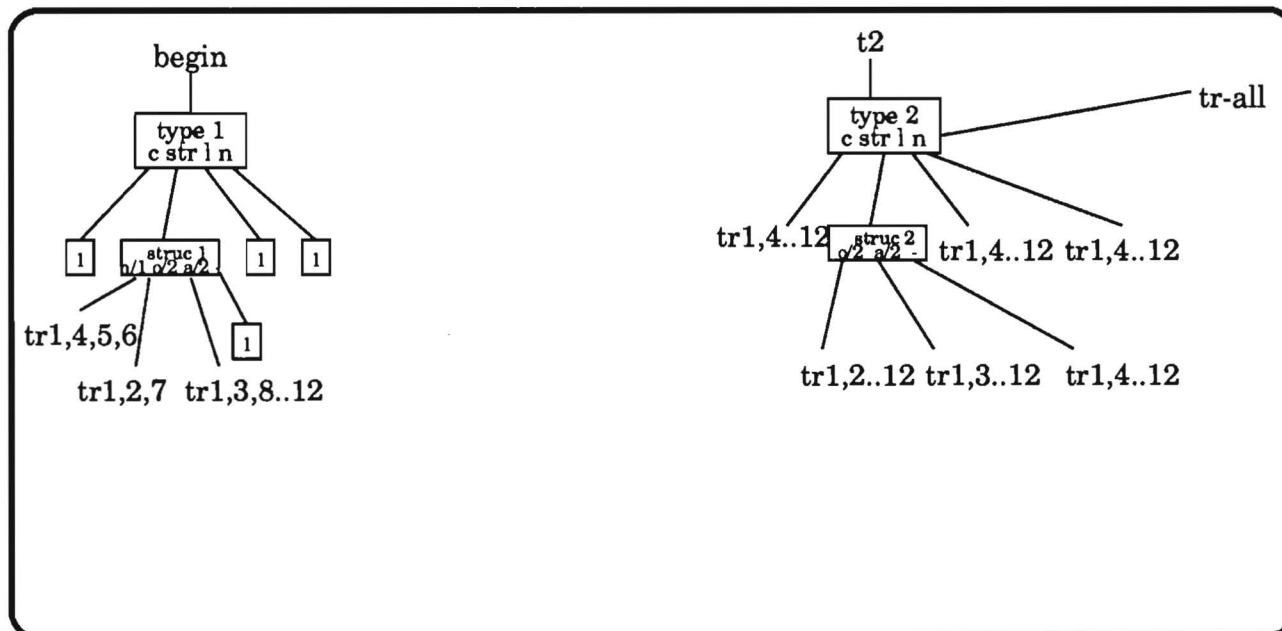


Figure 33: cut index tree

We can now see that some links (i.e. [1,2,7]) are lost and so the index tree must be extended by the corresponding try-chains:

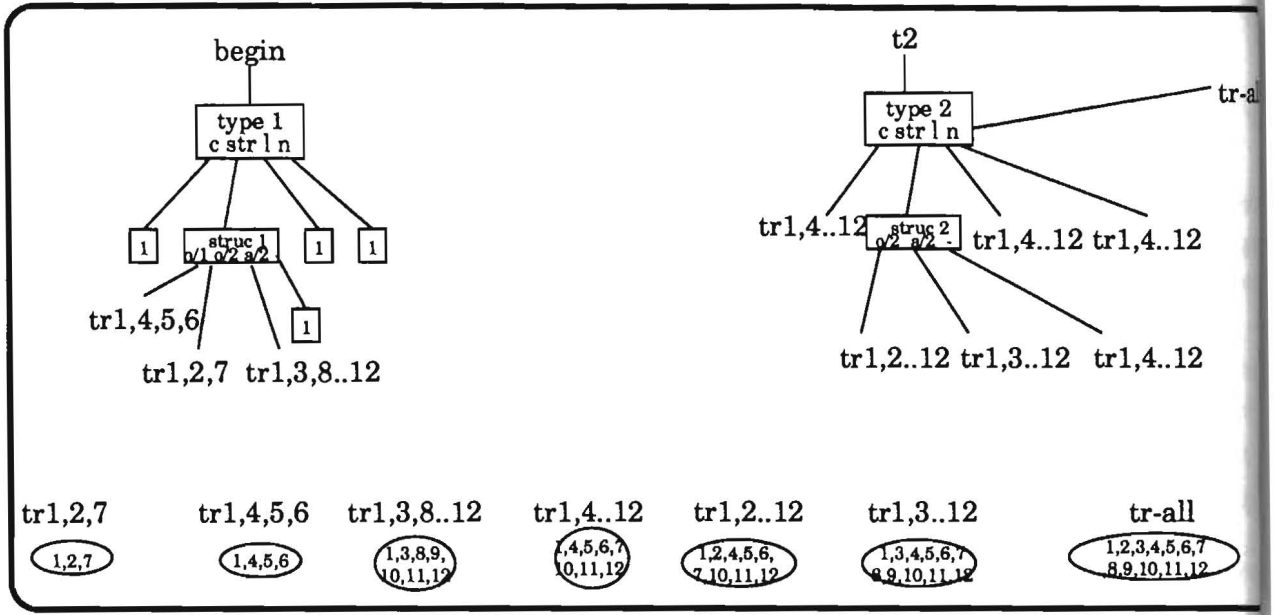


Figure 34: extended index tree

the resulting index code is:

```

((set_index_number 1)
 (switch_on_term 1 "label58" 1 1 "label50")
 "label58"
 (switch_on_structure
  3
  (((o 2) "label35") ((a 2) "label42") ((n 1) "label49"))
  1 )
 "label35"
 (set_index_number 2)
 (switch_on_term "label36" "label59" "label36" "label36" "label38")
 "label59"
 (switch_on_structure 1 (((o 2) "label38")) "label36")
 "label36"
 (try 1 2)
 (trust 7 2)
 "label38"
 (try 1 2)
 (retry 2 2)
 (trust 7 2)
 "label42"
 (set_index_number 2)
 (switch_on_term "label43" "label60" "label43" "label43" "label45")
 "label60"
 (switch_on_structure 1 (((a 2) "label45")) "label43")
 "label43"
 (try 1 2)
 (retry 10 2)
 (retry 11 2)
 (trust 12 2)
 "label45"
 (try 1 2)
 (retry 3 2)
 (retry 8 2)
 (retry 9 2)
 (retry 10 2)
 (retry 11 2)
 (trust 12 2)
 "label49"
 (try 1 2)
 (retry 4 2)
 (retry 5 2)
 (trust 6 2)
 "label50"
 (set_index_number 2)
 (switch_on_term "label51" "label61" "label51" "label51" "label57")
 "label61"
 (switch_on_structure 2 (((o 2) "label53")
                        ((a 2) "label54")) "label51")
 "label57"
 (try 1 2)
 (retry 2 2)
 (retry 3 2)
 (retry 4 2)
 (retry 5 2)
 (retry 6 2)
 (retry 7 2)
 (retry 8 2)
 (retry 9 2)
 (retry 10 2)
 (retry 11 2)
 (trust 12 2)
 "label53"
 (try 1 2)
 (retry 2 2)
 (retry 4 2)
 (retry 5 2)
 (retry 6 2)
 (retry 7 2)
 (retry 10 2)
 (retry 11 2)
 (trust 12 2)
 "label54"
 (try 1 2)
 (retry 3 2)
 (retry 4 2)
 (retry 5 2)
 (retry 6 2)
 (retry 7 2)
 (retry 8 2)
 (retry 9 2)
 (retry 10 2)
 (retry 11 2)
 (trust 12 2)
 "label51"
 (try 1 2)
 (retry 4 2)
 (retry 5 2)
 (retry 6 2)
 (retry 7 2)
 (retry 10 2)
 (retry 11 2)
 (trust 12 2))
1
.....
2
.....
WAM-code for clauses

```

12.1 Interface

We have tried to operate between the modules for the classified clauses and the WAM-compiler with an interface module: `iif.lsp`. All accessing operators for index information from the index tree and the classified clauses are handled via this module.

The label generation is done with the LISP function `gen-temp`. Since it could be necessary to expand a label to its corresponding try-chain, we always begin an index tree with its indexed clauses (the sequence S_G).

A single detail in the switch-on-type instruction has been changed to allow indexing on other arguments than the first one.

13 Comparisons

Our method is placed between the ordinary WAM indexing method and the complete indexing method, but in any case free to be extended for complete indexing.

We have tried to implement the main features of complete indexing and quadratic indexing and combine these researches with the WAM based compiler. Switches allow to relate the different strong points of several methods (code-optimization versus run-time-optimization).

In large database-like programs (like many domain specific applications) we reach the same performance as complete indexing (since they are nearly head deterministic) with not too much code-overhead in relation to no indexing.

14 Extensions

We now have to think about how to combine RELFUN-like features (like higher-order operators) and PROLOG-like features (like `assert` and `retract`) with indexing. The solution of the higher-order problem seems to be more a horizontal compilation problem rather than a problem on lower levels [5, 6].

But `assert` and `retract` is really a low-level problem. One simple solution is to allow no indexing for asserted clauses, only trying them in any case with a try-chain:

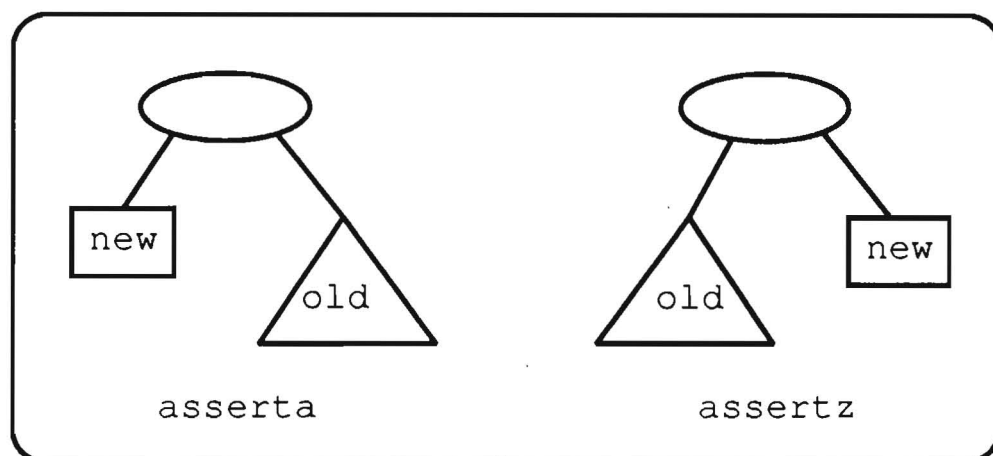


Figure 35: extension: assert

This solution also ensures “correct”³⁸ intended behavior concerning the semantic issues of the program. Predicate calls after an assertion use the new definition; all calls to a predicate

³⁸see [15]

before the assertion use the old definition. An example (using the SEPIA PROLOG system [10]) illustrates this behaviour:

```
SEPIA Version 3.0.5, Wed Jul 25 16:33 1990 Copyright ECRC GmbH
```

```
[sepia]: dynamic f/1.
```

```
yes.
```

```
[sepia]: [user].
```

```
f(1).
```

```
f(2).
```

```
f(3).
```

```
f(4) :- assert(f(5)).
```

```
user          compiled 296 bytes in 0.03 seconds
```

```
yes.
```

```
[sepia]: f(X).
```

```
X = 1      More? (;)
```

```
X = 2      More? (;)
```

```
X = 3      More? (;)
```

```
X = 4
```

```
yes.
```

```
[sepia]: f(X).
```

```
X = 1      More? (;)
```

```
X = 2      More? (;)
```

```
X = 3      More? (;)
```

```
X = 4      More? (;)
```

```
X = 5
```

```
yes.
```

We can see that the first call of `f/1` only returns the numbers form 1 to 4, even if the assertion of a new clause changes the definition of `f/1`.

A clause can be retracted by only deleting its occurrence in the index tree and recompiling the index tree (possibly a “ghost” clause survive in the program which is never tried). In the following figure we see the fully indexed index tree of the example (simple 6-fact procedure) after deleting clause number 4.

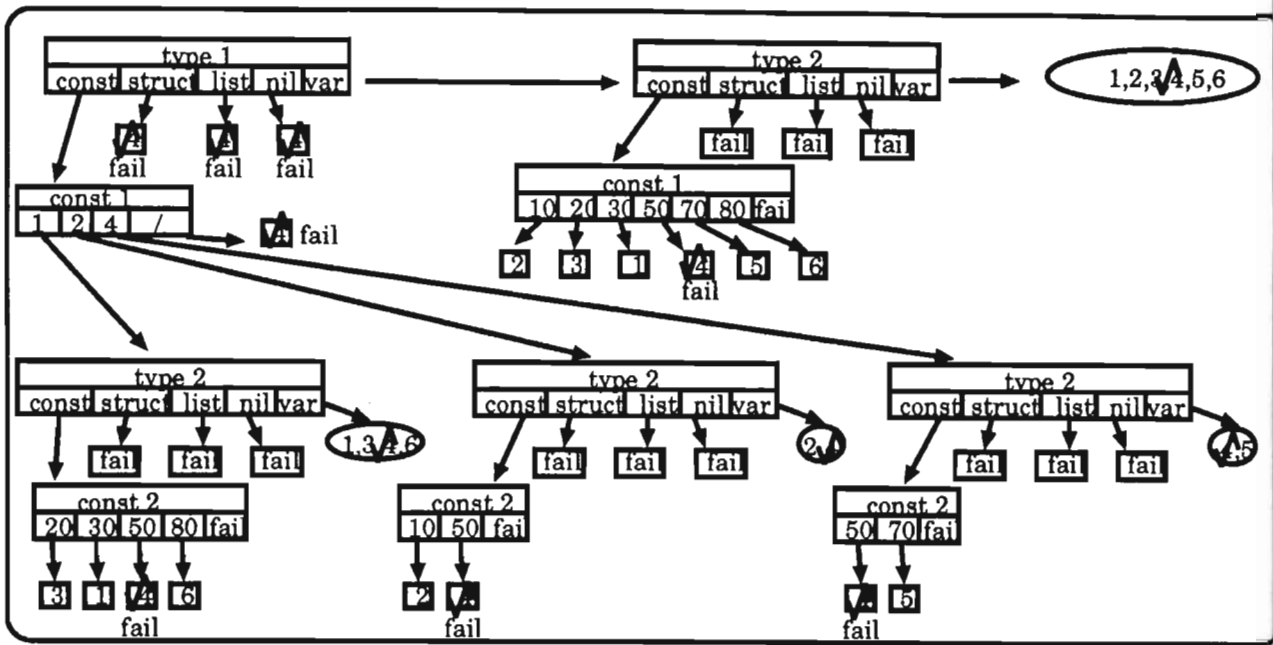


Figure 36: retract clause number 4

To ensure “correct” behavior of still called clauses we have to copy the index tree and change the copy.

Another possibility is not to change the index tree but the program code. We can substitute the complete clause code (which we want to delete) with the instruction `fail`. In this case an “empty” clause is tried in the index tree.

Then we not only have to copy the index tree but also the program code to save the old definition.

We see that the first proposal takes advantage of head-informed index trees: the possibility of separate compilation of index code and clause code.

14.1 Improved Indexing III (not only main structure)

Another problem is the extension of RFM indexing to complete indexing. This extension is not difficult since instead of linking the index register to an argument register we can also allow a link to any X-register which can be bound to any inner structure. The question is if it makes sense to produce complete indexing code, since this code is a mixture of indexing code and clause code and no local recompilation is possible (see `assert` and `retract`).

15 Benchmark Results

The next table gives an overview of three benchmarks³⁹:

The first benchmark is the well known naive reverse benchmark.

³⁹these benchmark results are not very exact, since run-time was taken by hand (our emulator has no run-time-measure-predicate). In an experimental C implementation of the WAM the Unix function `time` was used. We found out that the effect of indexing also depends on the low-level WAM implementation.

The second benchmark (dnf) is the complete program from section 12.

The third test was the NET benchmark. NET is an automatically generated tool-selection program for ARC-TEC's engineering domain. Its task is to select a cutting tool for a special workpiece for a CNC-lathe machine.

Last but not least we test the well know naive reverse benchmark.

Since the ν -WAM was conceived as a didactic prototype written in higher-level LISP, not as a PROLOG product, the absolute values are not yet competitive with well known production PROLOGs. The average speed-up gained by indexing in our database-like applications, however is a factor between 20 and 30. But even rather deterministic procedures like `append` and `reverse` produce a speed-up of at least a factor of 2.

We are currently thinking of a lower-level version of our indexing scheme which should give us competitive absolute speed.

test-name	test-environment	time
nrev : well known naive reverse benchmark 6 lines arity of procedures: 2-3		
	SUN 4 125 MB RAM no indexing	13 sec
	SUN 4 125 MB RAM indexing	7 sec
dnf : tool from Hans Günter Hein (see [13]) 105 lines arity of procedures: 2-3		
	IVORY LISP-BOARD no indexing	84 sec
	IVORY LISP-BOARD indexing	24 sec
	SUN 4 125 MB RAM no indexing	425 sec
	SUN 4 125 MB RAM indexing	120 sec
NET : 312 lines arity of procedures 2-3		
	IVORY LISP-BOARD no indexing	288 sec
	IVORY LISP-BOARD indexing	15 sec
	SUN 4 125 MB RAM no indexing	1460 sec
	SUN 4 125 MB RAM indexing	72 sec

Figure 37: run-time results

Part V

Appendix

A User Commands



Since indexing should be automatic the index-structure is hidden from the RELFUN user. The only instruction to control indexing is:

```
indexing { on | off | :min-clauses <no> | :max-vars <no> | :max-depth <no> | :max-args
<no> | :debug on | :debug off }
```

The effect of calling *indexing* without any option is displaying the current settings.

The switches have the following effects:

on (*off*) switches indexing on (off),

:min-clauses <*no*> sets the minimal number of clauses for an indexable operator definition to <*no*>,

:max-vars <*no*> sets the maximal number of variables allowed in a constant block (block-variable-size) to <*no*>,

:max-depth <*no*> sets the maximal depth of the index tree to <*no*>,

:max-args <*no*> sets the maximal number of parallelly indexable arguments (index tree breadth) to <*no*>,

:debug on (*off*): for internal use only

Mutually excluding options result in executing only the last one.

Example:

```
rfe> indexing
indexing on :min-clauses 2 :max-vars 10 :max-depth 3 :max-args 2 :debug off
```

```
rfe> indexing :min-clauses 3
indexing on :min-clauses 3 :max-vars 10 :max-depth 3 :max-args 2 :debug off
```

```
rfe> indexing :max-depth 4 :max-args 3 :max-depth 5
indexing on :min-clauses 2 :max-vars 10 :max-depth 5 :max-args 3 :debug off
```

B Program

We show the LISP function heads defined in the modules. A few algorithms are also explained (rather than giving the LISP definitions)

B.1 MODULE: IDX.LSP

```

(defvar idx.*indexing* t)
(defun idx () ..)
(defvar idx.*dbg* nil)
(defvar idx.*min-no-of-proc-clauses* 2)
(defvar idx.*max-no-of-vars* 10)
(defvar idx.*maxdepth* 3)
(defvar idx.*numberofargs* 2)
(defun idx.show-idx-constants () ..)
(defun idx.idx-1cmd (paras) ..)
(defun idx.idx-cmd (paras) ..)

```

The variables are used to set the compiler switches; they are initialized with useful values. The functions are all extensions of the RELFUN run-time-loop (for example the command to set the compiler switches: *indexing* ...). This module should include all general functions and variable declarations which are used from the RELFUN main-loop.

B.2 MODULE: IIF.LSP

```

(defun iif.number-or-nil-p (item) ..)
(defun iif.tag-of-idxtree (idxtree) ..)
(defun iif.s-label-f-idxtree (idxtree) ..)
(defun iif.s-label-f-lab+idxtree (idxtree) ..)
(defun iif.s-clauses-f-idxtree (idxtree) ..)
(defun iif.s-idxtrees-f-try-trust (try-trust) ..)
(defun iif.s-arg-f-indextree (indextree) ..)
(defun iif.s-sequindparts-f-indextree (indextree) ..)
(defun iif.if-s.o.?-sequindpart (sequindpart) ..)
(defun iif.s-s.o.?-f-sequindpart (sequindpart) ..)
(defun iif.s-idxtree-f-sequindpart (sequindpart) ..)
(defun iif.s-switchparts-f-sequindpart (sequindpart) ..)
(defun iif.s-atom-f-switchpart (switchpart) ..)
(defun iif.s-idxtree-f-switchpart (switchpart) ..)
(defun iif.s-clauses-f-clauses (clauses) ..)
(defun iif.mk-tree (class-proc) ..)
(defun iif.mapindex (blocks) ..)
(defun iif.mk-indextree (block) ..)
(defun iif.seqind-list-car-cdr (block rest clauses) ..)
(defun iif.element-from-seqind-elementlist (element) ..)
(defun iif.mk_block_from_element (element) ..)

```

This module is called *indexing interface module*. It should include all interface functions and predicates to access the indexing information from the classified clauses or the index tree.

B.3 MODULE: LINEAR.LSP

```
(defun flatten-idx (idxtree) ..)
(defun iif.s-idxtree-f-indextree (indextree) ..)
(defun linearize (lab+idxtree list-of-idxtrees) ..)
(defun lin.unique (list-o-idxtrees) ..)
(defun lin.s-label-f-idxtree (idxtree) ..)
(defun lin.cut-down-next-one (list-o-idxtrees idxtree max-args max-depth) ..)
(defun lin.find-label (label list-o-idxtrees) ..)
(defun lin.cut-down (list-o-idxtrees next-label max-args max-depth) ..)
(defun lin.mk-try-trust-label-f-label (label) ..)
(defun iif.sub-label (idxtree idxtrees list-o-all-idxtrees) ..)
(defun lin.insert-t-t (list-o-idxtrees) ..)
(defun lin.insert-try-trust (list-o-idxtrees next-label) ..)
(defun iif.mk-label-f-idxtree (idxtree) ..)
(defun lin.search-try-trust-labels (idxtree list-o-idxtrees) ..)
(defun lin.search-idxtrees-f-try-trust (idxtree list-o-idxtrees) ..)
(defun lin.s-a-label (idxtree list-o-idxtrees) ..)
(defun lin.s-a-label-if-found (idxtree list-o-idxtrees) ..)
(defun lin.search-indextree-labels (idxtree list-o-idxtrees) ..)
(defun lin.s-indextrees-f-idxtree (idxtree) ..)
(defun iif.s-typetag-f-sequind (sequind) ..)
```

This module includes the code for generating, flattening, cutting and extending an index tree from the index information of the classified clauses.

B.3.1 Algorithms

We show the algorithms for flattening, cutting and extending an index tree:

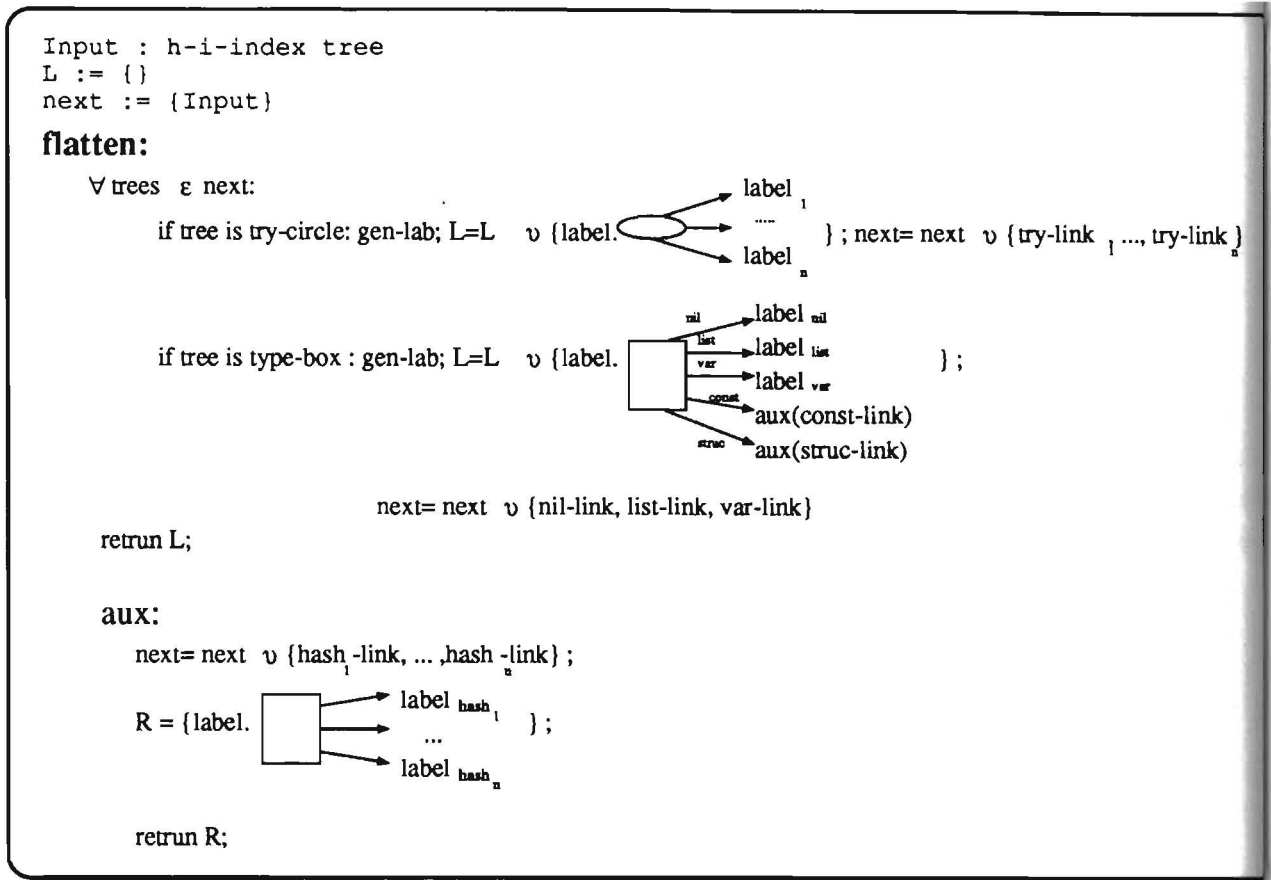


Figure 38: flattening algorithm

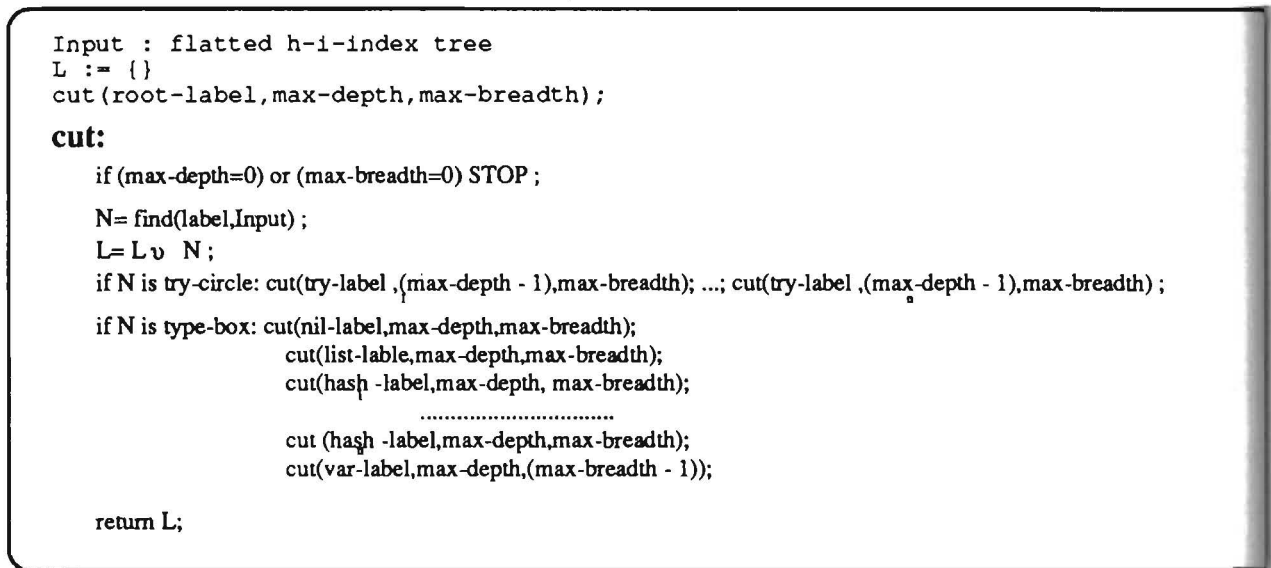


Figure 39: cutting algorithm


```

Input : flatted cutted h-i-index tree
L := {Input}
next= {root-label};

```

extend:

\forall labels \in next:

N= find(label,Input);

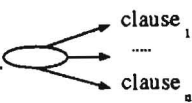
if N is "not found" : L= L \cup gen-try-trust(label);

else: L= L \cup N;

next=next \cup successors(N);

gen-try-trust:

```

return: {label.  };

```

Figure 40: extending algorithm

B.4 MODULE: ICG.LSP

```
(defun icg.mk-header (idxtree x1579) ..)
(defun icg.mk-t-r-t-list-f-idxtrees (idxtrees x1579) ..)
(defun icg.gen-t-r-t (idxtree x1579 tag) ..)
(defun icg.mk-s.o.t.-f-indextree (idxtree x1579) ..)
(defun icg.gen_switch_on_? (sequind) ..)
(defun icg.mk-symbol-label-f-idxtree (idxtree) ..)
```

The name of this module is: *indexing codegenerator*. Its task is to generate general WAM index header code from a given index tree. This code is pushed in front of the compile clause code. We assume that the label of a clause i is its number i and there are no other numeric labels in the code. So calling the first clause of a definition is just a jump to label 1.

C Benchmark Sources

We now present the benchmarks used in section 15.

C.1 nrev Benchmark

The `nrev` procedure is tested with a list of fifty elements.

```
nrev([], []).
```

```
nrev([X|Y], Z) :- nrev(Y, Z1),
  append(Z1, [X], Z).
```

```
append([], L, L).
```

```
append([X|Y], L, [X|Z]) :- append(Y, L, Z).
```

C.2 dnf Benchmark

We called this benchmark with the procedure `go4`. Only the time for finding the first solution is measured.

```
literal(z0).
literal(z1).
literal(z2).
literal(z3).
literal(z4).
literal(z5).
literal(z6).
literal(z7).
literal(z8).
literal(z9).
literal(n[X]) :- literal(X).
```

```
norm(X, X) :- literal(X).
norm(o[X, Y], o[X, Y]) :-
  literal(X),
  literal(Y).
norm(a[X, Y], a[X, Y]) :-
  literal(X),
```

```
  literal(Y).
norm(o[X, Y], o[X1, Y]) :-
  literal(Y),
  norm(X, X1).
norm(o[X, o[Y, Z]], W) :-
  norm(o[o[X, Y], Z], W).
norm(o[X, a[Y1, Y2]], o[X1, Y12]) :-
  norm(X, X1),
  norm(a[Y1, Y2], Y12).
norm(a[X, Y], a[X1, Y]) :-
  literal(Y),
  norm(X, X1).
norm(a[X, a[Y, Z]], W) :-
  norm(a[a[X, Y], Z], W).
norm(a[X, o[Y1, Y2]], a[X1, Y12]) :-
  norm(X, X1),
  norm(o[Y1, Y2], Y12).
```

```
dnf(X, X) :- literal(X).
dnf(o[X, Y], o[X, Y]) :-
  literal(X),
  literal(Y).
dnf(a[X, Y], a[X, Y]) :-
  literal(X),
  literal(Y).
dnf(n[n[X]], W) :- dnf(X, W).
dnf(n[o[X, Y]], W) :- dnf(a[n[X], n[Y]], W).
dnf(n[a[X, Y]], W) :- dnf(o[n[X], n[Y]], W).
dnf(o[X, Y], W) :- dnf(X, X1),
  dnf(Y, Y1),
  norm(o[X1, Y1], W).
dnf(a[X, Y], a[a[X1, X2], Y]) :-
  literal(Y),
  dnf(X, a[X1, X2]).
dnf(a[X, Y], a[a[Y1, Y2], X]) :-
  literal(X),
  dnf(Y, a[Y1, Y2]).
dnf(a[X, Y], W) :-
  dnf(X, a[X1, X2]),
  dnf(Y, a[Y1, Y2]),
  norm(a[a[X1, X2], a[Y1, Y2]], W).
dnf(a[X, Y], W) :-
  dnf(X, o[X1, X2]),
  dnf(Y, Y1),
  dnf(o[a[X1, Y1], a[X2, Y1]], W).
dnf(a[X, Y], W) :-
  dnf(X, X1),
  dnf(Y, o[Y1, Y2]),
  dnf(o[a[X1, Y1], a[X1, Y2]], W).
```

```

go1(X) :- dnf(a[z1,
              a[z2,
                o[z3,
                  a[z4,
                    a[z5, z6]]]]],
              X).
go2(X) :- dnf(o[o[a[z1, z2], z3],
              o[a[z4,
                a[a[z5, z6],
                  z7]],
                o[z8, z9]]],
              X).
go3(X) :- dnf(a[a[z1, a[o[z2, z3], z4]],
              a[z5, o[z6, z7]]],
              X).
go4(X) :- dnf(n[o[a[n[o[z1, z2]],
                  n[a[z3, z4]]],
                a[n[z5],
                  o[a[z6, a[z7, z8]],
                    z9]]]],
              X),
          dnf(n[o[a[n[o[z1, z2]],
                  n[a[z3, z4]]],
                a[n[z5],
                  o[a[z6, a[z7, z8]],
                    z9]]]],
              X).
isa(20, spitz).
is-leaf(20).
isa(30, spitz).
is-leaf(30).
isa(60, spitz).
is-leaf(60).
isa(80, spitz).
is-leaf(80).
isa(180, stumpf).
is-leaf(180).
isa(150, stumpf).
is-leaf(150).
isa(140, stumpf).
is-leaf(140).
isa(130, stumpf).
is-leaf(130).
isa(100, stumpf).
is-leaf(100).
isa(stumpf, winkel).
isa(spitz, winkel).
isa(rechter, winkel).
isa(rund, nicht-eckig).
is-leaf(rund).
isa(quader, viereck).
is-leaf(quader).
isa(quadrat, viereck).
is-leaf(quadrat).
isa(viereck, eckig).
isa(dreieck, eckig).
is-leaf(dreieck).
isa(rhomb, eckig).
is-leaf(rhomb).
isa(eckig, geometrie).
isa(nicht-eckig, geometrie).
isa(s1, stahl).
is-leaf(s1).
isa(s2, stahl).
is-leaf(s2).
isa(s3, stahl).
is-leaf(s3).
isa(s4, stahl).
is-leaf(s4).
isa(s5, stahl).
is-leaf(s5).
isa(s6, stahl).
is-leaf(s6).
isa(k741, k74).
is-leaf(k741).
isa(k742, k74).
is-leaf(k742).
isa(90, rechter).
is-leaf(90).
isa(0, spitz).
is-leaf(0).
isa(10, spitz).
is-leaf(10).

```

C.3 NET Benchmark

The run-time for finding the first solution of the predicate call `tool-selection(X,Y)`. is given in the benchmark results.

```

isa(k743, k74).
is-leaf(k743).
isa(k71, k7).
is-leaf(k71).
isa(k72, k7).
is-leaf(k72).
isa(k73, k7).
is-leaf(k73).
isa(k74, k7).
isa(k75, k7).
is-leaf(k75).
isa(k76, k7).
is-leaf(k76).
isa(k77, k7).
is-leaf(k77).
isa(k78, k7).
is-leaf(k78).
isa(k79, k7).
is-leaf(k79).
isa(k710, k7).
is-leaf(k710).
isa(k21, k2).
is-leaf(k21).
isa(k22, k2).
is-leaf(k22).
isa(k23, k2).
is-leaf(k23).
isa(k24, k2).
is-leaf(k24).
isa(k11, k1).
is-leaf(k11).
isa(k12, k1).
is-leaf(k12).
isa(k13, k1).
is-leaf(k13).
isa(k1, keramik).
isa(k2, keramik).
isa(k3, keramik).
is-leaf(k3).
isa(k4, keramik).
is-leaf(k4).
isa(k5, keramik).
is-leaf(k5).
isa(k6, keramik).
is-leaf(k6).
isa(k7, keramik).
isa(k8, keramik).
is-leaf(k8).
isa(k9, keramik).
is-leaf(k9).

isa(k10, keramik).
is-leaf(k10).
isa(stahl, material).
isa(keramik, material).
isa(hss, material).
is-leaf(hss).

tool-num(Wkl, Mat) :-
    s-tool(Mat, Down-geo-1),
    s-angle(Down-geo-1, Wkl),
    s-position(Wkl, Mat),
    numeric-test(Wkl, Mat).

mixed-selection(Wkl, Mat) :-
    s-tool(Mat, Down-down-geo-1-1),
    s-angle(Down-down-geo-1-1, Wkl),
    s-position(Wkl, Mat),
    s-wrk(Mat, Down-down-geo-2-1),
    s-angle(Down-down-geo-2-1, Wkl),
    s-position(Wkl, Mat),
    s-lager(Mat, Geo).

h-selection(Wkl, Mat) :-
    s-tool(Mat, Down-geo-1),
    s-angle(Down-geo-1, Wkl),
    s-position(Wkl, Mat),
    s-wrk(Mat, Down-geo-2),
    s-angle(Down-geo-2, Wkl),
    s-position(Wkl, Mat).

tool-selection2(Wkl, Mat) :-
    s-wrk(Mat, Geo),
    s-angle(Geo, Wkl),
    s-position(Wkl, Mat).

s-wrk(A, B) :- is-leaf(A),
               is-leaf(B),
               t-isa(A, s1),
               t-isa(B, rund).

s-wrk(A, B) :- is-leaf(A),
               is-leaf(B),
               t-isa(A, s2),
               t-isa(B, nicht-eckig).

s-wrk(A, B) :- is-leaf(A),
               is-leaf(B),

```

```

t-isa(A, k12),
t-isa(B, rund).

s-tool(Mat5, Geo5),
s-angle(Geo5, Wkl5),
s-position(Wkl5, Mat5).

10-tool-selection(Wkl1, Wkl2) :-
s-tool(Mat1, Down-geo1-1),
s-angle(Down-geo1-1, Wkl1),
s-position(Wkl1, Mat1),
s-tool(Mat2, Down-geo2-1),
s-angle(Down-geo2-1, Wkl2),
s-position(Wkl2, Mat2),
s-tool(Mat3, Down-geo3-1),
s-angle(Down-geo3-1, Wkl3),
s-position(Wkl3, Mat3),
s-tool(Mat4, Down-geo4-1),
s-angle(Down-geo4-1, Wkl4),
s-position(Wkl4, Mat4),
s-tool(Mat5, Down-geo5-1),
s-angle(Down-geo5-1, Wkl5),
s-position(Wkl5, Mat5),
s-tool(Mat1, Down-geo1-2),
s-angle(Down-geo1-2, Wkl1),
s-position(Wkl1, Mat1),
s-tool(Mat2, Down-geo2-2),
s-angle(Down-geo2-2, Wkl2),
s-position(Wkl2, Mat2),
s-tool(Mat3, Down-geo3-2),
s-angle(Down-geo3-2, Wkl3),
s-position(Wkl3, Mat3),
s-tool(Mat4, Down-geo4-2),
s-angle(Down-geo4-2, Wkl4),
s-position(Wkl4, Mat4),
s-tool(Mat5, Down-geo5-2),
s-angle(Down-geo5-2, Wkl5),
s-position(Wkl5, Mat5).

5-tool-selection(Wkl1, Wkl2) :-
s-tool(Mat1, Geo1),
s-angle(Geo1, Wkl1),
s-position(Wkl1, Mat1),
s-tool(Mat2, Geo2),
s-angle(Geo2, Wkl2),
s-position(Wkl2, Mat2),
s-tool(Mat3, Geo3),
s-angle(Geo3, Wkl3),
s-position(Wkl3, Mat3),
s-tool(Mat4, Geo4),
s-angle(Geo4, Wkl4),
s-position(Wkl4, Mat4),

tool-selection(Wkl, Mat) :-
s-tool(Mat, Geo),
s-angle(Geo, Wkl),
s-position(Wkl, Mat).

s-lager(A, B) :- is-leaf(A),
is-leaf(B),
t-isa(A, stahl),
t-isa(B, 100).

s-lager(A, B) :- is-leaf(A),
is-leaf(B),
t-isa(A, keramik),
t-isa(B, 150).

s-lager(A, B) :- is-leaf(A),
is-leaf(B),
t-isa(A, hss),
t-isa(B, 90).

s-position(A, B) :- is-leaf(A),
is-leaf(B),
t-isa(A, stumpf),
t-isa(B, stahl).

s-position(A, B) :- is-leaf(A),
is-leaf(B),
t-isa(A, rechter),
t-isa(B, keramik).

s-position(A, B) :- is-leaf(A),
is-leaf(B),
t-isa(A, 10),
t-isa(B, k1).

s-angle(A, B) :- is-leaf(A),
is-leaf(B),
t-isa(A, viereck),
t-isa(B, 150).

s-angle(A, B) :- is-leaf(A),
is-leaf(B),
t-isa(A, viereck),
t-isa(B, 100).

s-angle(A, B) :- is-leaf(A),
is-leaf(B),
t-isa(A, dreieck),
t-isa(B, 180).

s-angle(A, B) :- is-leaf(A),
is-leaf(B),
t-isa(A, rund),

```

```
        t-isa(B, spitz).
s-tool(A, B) :- is-leaf(A),
               is-leaf(B),
               t-isa(A, s2),
               t-isa(B, eckig).
s-tool(A, B) :- is-leaf(A),
               is-leaf(B),
               t-isa(A, s5),
               t-isa(B, eckig).
s-tool(A, B) :- is-leaf(A),
               is-leaf(B),
               t-isa(A, k1),
               t-isa(B, nicht-eckig).
s-tool(A, B) :- is-leaf(A),
               is-leaf(B),
               t-isa(A, k12),
               t-isa(B, rund).
```

D Extended Abstract



In the last few years PROLOG has changed its appearance from an experimental to a more and more serious language. This is due to many people thinking about good compiling techniques and useful extensions.[1] [29] We show that the use of simple control instructions based on the Warren Abstract Machine (WAM)[29] can speed up execution efficiency enormously.

In a first part we introduce the idea of indexing and where it comes from. The difference between DB-indexing methods and those for PROLOG-like languages are discussed. Then the main advantages for indexing and their disadvantages are compared. As a result we show that indexing changes the run-time and the memory management behavior of a PROLOG emulator.

In a second part we give short theoretical background information about the general indexing problem.[11]

- First, index trees can increase to an exponential code size if constants and variables are mixed in a specific way in the argument positions of the heads of clauses. In this case a set of n clauses can be partitioned into two sets of $(n - 1)$ and $(n - 2)$ clauses, which are then used recursively to construct the child subtrees until the leaf of the index tree corresponds to only a single rule.
- Second, for any reasonable definition of “optimal”, finding an *optimal* index tree is NP-complete. This could happen, if the index scheme provides indexing of inner structures of arguments. In this case the problem of finding a minimal subset of argument positions such that two rules do not unify in all positions of this set can be shown equivalent to the set-covering problem, which is known to be NP-complete.

Several possible indexing methods, different implementation techniques and a step-by-step introduction into our RFM-indexing method are given. Existing indexing methods can be grouped as follows:

1. hardware-oriented approaches[14]
 - (a) m-in-n-coding
2. software-oriented approaches (WAM-oriented)[14]
 - (a) general WAM-indexing[29]
 - (b) complete indexing
 - (c) shallow backtracking
 - (d) quadratic indexing

We discuss three different possibilities for the implementation task:

1. horizontal transformation on PROLOG-level

2. vertical transformation down to WAM-level
3. WAM-extensions

Our implementation of an indexing method is an amalgamation of (as we think) the positive aspects of the software-oriented approaches and easy to extend with other features like assert. In order to explain our method we introduce a new graphical representation for general indexing methods. Step by step, beginning with a non-indexed example, we derive a extensible, flexible, non-first, multi-argument indexing method.

In a short section we explain, why we call our indexing method an “intelligent” one. Heuristics, which determine which argument position should be indexed, why, and in which order, are presented.

Last but not least, we show benchmark results and give ideas for further work in this area. We indicate that our method is able to handle features like higher-order operators as well as assert and retract. We also discuss the idea of extending our technique to a complete-indexing method.

References

- [1] Hassan Ait-Kaci. The WAM: A (Real) Tutorial. Report 5, Digital, Paris Research Laboratory, January 1990.
- [2] Alain Callebaut Bart Demoen, Andre Marien. Indexing PROLOG Clauses. *Journal of Logic Programming*, page 1001 ff, 1989.
- [3] H. Benker, J. Beacco, S. Bescos, M. Dorochevsky, Th. Jeffré, A. Pöhlmann, J. Noyé, B. Poterie, A. Sexton, J.C. Syre, O. Thibault, and G. Watzlawik. KCM: A Knowledge Crunching Machine. In *Proceedings of the International Symposium on Computer Architecture*, Jerusalem, May 1989.
- [4] Claude Berge. *Graphs and Hypergraphs*. North Holland Publishing-Company, 1973.
- [5] Harold Boley. A Relational/Functional Language and Its Compilation into the WAM. SEKI Report SR-90-05, Universität Kaiserslautern, Fachbereich Informatik, April 1990.
- [6] Harold Boley, editor. *Beiträge zum Arbeitstreffen über WAM-Erweiterungen am DFKI Kaiserslautern*, number 91-2, März 1991.
- [7] 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.
- [8] Harold Boley and Michael M. Richter, editors. *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, number 567 in Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, Berlin, Heidelberg, 1991.

- [9] K.L. Clark. Predicate Logic as a Computational Formalism. Report, Imperial College of Science and Technology, December 1979.
- [10] ECRC. *SEPIA PROLOG 3.0: Manual*. ICL, 1990.
- [11] 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.
- [12] Hans-Günther Hein. Adding WAM-Instructions to support Valued Clauses for the Relational/Functional Language RELFUN. SEKI Working Paper SWP-90-02, Universität Kaiserslautern, Fachbereich Informatik, December 1989.
- [13] Hans Günther Hein. WAM Indexing and Footening Techniques for RELFUN – A case study on the DNF benchmark. Discussion Paper 91-11, DFKI Kaiserslautern, August 1991.
- [14] Timothy Hickey and Shyam Mudambi. Global Compilation of Prolog. *Journal of Logic Programming*, 7:193–230, 1989.
- [15] ISO. PROLOG ISO-draft. *electronic mail*, 1992.
- [16] Robert Kowalski. Algorithm = Logic + Control. *Communications of the ACM* 22(7):424–436, July 1979.
- [17] Thomas Krause. Klassifizierte relational/funktionale Klauseln: Eine deklarative Zwischensprache zur Generierung von Register-optimierten WAM-Instruktionen. SEKI Working Paper SWP-90-04, Universität Kaiserslautern, Fachbereich Informatik, Mai 1990.
- [18] Thomas Krause. Globale Datenflussanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN. Document D-91-08, DFKI GmbH, 1991.
- [19] Thomas Krause. Program Transformations in a RELFUN subset. Diplomarbeit, Universität Kaiserslautern, FB Informatik, March 1991. Also available as DFKI Document.
- [20] Peter Kursawe. How to invent a Prolog Machine. In E. Shapiro, editor, *Third International Conference on Logic Programming (ICLP)*, LNCS 225, pages 134–148, London, July 1986. Springer Verlag.
- [21] Alan Mycroft and Richard A. O’Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23:295–307, 1984.
- [22] Sven Olof Nyström. Nywam - a WAM emulator written in LISP.
- [23] Michael Sintek. Indexing PROLOG Procedures into DAGs by Heuristic Classification. DFKI document, DFKI GmbH, Forthcoming 1992.
- [24] Werner Stein and Michael Sintek. A generalized intelligent indexing method. In *Workshop “Sprachen für KI-Anwendungen, Konzepte - Methoden - Implementierungen” in Bad Honnef, 12/92-1*. Institute of Applied Mathematics and Computer Science, University of Münster, May 1992.

-
- [25] Andrew Taylor. High Performance PROLOG Implementation through Global Analysis. In Harold Boley Michael M. Richter, editor, *International Workshop on Processing Declarative Knowledge*, 1991.
- [26] Andrew Taylor. Lips on a Mips. Technical report, University of Sidney, AU, 1991.
- [27] Peter Lodewige van Roy. *Can Logic Programming Execute as Fast as Imperative Programming*. PhD thesis, University of California, 1990.
- [28] Hans W. A Complete Indexing Scheme for WAM Based Abstract Machines. In *Programming Language Implementation and Logic Programming*, page 232 ff. 1992.
- [29] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.
- [30] David S. Warren. Database Updates in Pure Prolog. In *International Conference on Fifth Generation Computer Systems*, pages 244–253, 1984.
- [31] Maurer Wilhelm. *Übersetzerbau. Theorie, Konstruktion, Generierung*. Springer Lehrbuch, 1992.



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 published 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-92-14

Intelligent User Support in Graphical User Interfaces:

1. InCome: A System to Navigate through Interactions and Plans
Thomas Fehrle, Markus A. Thies
2. Plan-Based Graphical Help in Object-Oriented User Interfaces
Markus A. Thies, Frank Berger

22 pages

RR-92-15

Winfried Graf: Constraint-Based Graphical Layout of Multimodal Presentations
23 pages

RR-92-16

Jochen Heinsohn, Daniel Kudenko, Bernhard Nebel, Hans-Jürgen Profitlich: An Empirical Analysis of Terminological Representation Systems
38 pages

RR-92-17

Hassan Ait-Kaci, Andreas Podelski, Gert Smolka: A Feature-based Constraint System for Logic Programming with Entailment
23 pages

RR-92-18

John Nerbonne: Constraint-Based Semantics
21 pages

RR-92-19

Ralf Legleitner, Ansgar Bernardi, Christoph Klauck: PIM: Planning In Manufacturing using Skeletal Plans and Features
17 pages

RR-92-20

John Nerbonne: Representing Grammar, Meaning and Knowledge
18 pages

RR-92-21

Jörg-Peter Mohren, Jürgen Müller: Representing Spatial Relations (Part II) -The Geometrical Approach
25 pages

RR-92-22

Jörg Würtz: Unifying Cycles
24 pages

RR-92-23

Gert Smolka, Ralf Treinen: Records for Logic Programming
38 pages

RR-92-24

Gabriele Schmidt: Knowledge Acquisition from Text in a Complex Domain
20 pages

RR-92-25

Franz Schmalhofer, Ralf Bergmann, Otto Kühn, Gabriele Schmidt: Using integrated knowledge acquisition to prepare sophisticated expert plans for their re-use in novel situations
12 pages

RR-92-26

Franz Schmalhofer, Thomas Reinartz, Bidjan Tschaischian: Intelligent documentation as a catalyst for developing cooperative knowledge-based systems
16 pages

RR-92-27

Franz Schmalhofer, Jörg Thoben: The model-based construction of a case-oriented expert system
18 pages

RR-92-29

Zhaohui Wu, Ansgar Bernardi, Christoph Klauck: Skeletal Plans Reuse: A Restricted Conceptual Graph Classification Approach
13 pages

RR-92-30

Rolf Backofen, Gert Smolka
A Complete and Recursive Feature Theory
32 pages

RR-92-31

Wolfgang Wahlster
Automatic Design of Multimodal Presentations
17 pages

RR-92-33

Franz Baader: Unification Theory
22 pages

RR-92-34

Philipp Hanschke: Terminological Reasoning and Partial Inductive Definitions
23 pages

RR-92-35

Manfred Meyer:
Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a CIM Environment
18 pages

RR-92-36

Franz Baader, Philipp Hanschke:
Extensions of Concept Languages for a Mechanical Engineering Application
15 pages

RR-92-37

Philipp Hanschke: Specifying Role Interaction in Concept Languages
26 pages

RR-92-38

Philipp Hanschke, Manfred Meyer:
An Alternative to Θ -Subsumption Based on Terminological Reasoning
9 pages

RR-92-40

Philipp Hanschke, Knut Hinkelmann: Combining Terminological and Rule-based Reasoning for Abstraction Processes
17 pages

RR-92-41

Andreas Lux: A Multi-Agent Approach towards Group Scheduling
32 pages

RR-92-42

John Nerbonne:
A Feature-Based Syntax/Semantics Interface
19 pages

RR-92-43

Christoph Klauck, Jakob Mauss: A Heuristic driven Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM
17 pages

RR-92-44

Thomas Rist, Elisabeth André: Incorporating Graphics Design and Realization into the Multimodal Presentation System WIP
15 pages

RR-92-45

Elisabeth André, Thomas Rist: The Design of Illustrated Documents as a Planning Task
21 pages

RR-92-46

Elisabeth André, Wolfgang Finkler, Winfried Graf, Thomas Rist, Anne Schauder, Wolfgang Wahlster: WIP: The Automatic Synthesis of Multimodal Presentations
19 pages

RR-92-47

Frank Bomarius: A Multi-Agent Approach towards Modeling Urban Traffic Scenarios
24 pages

RR-92-48

Bernhard Nebel, Jana Koehler:
Plan Modifications versus Plan Generation: A Complexity-Theoretic Perspective
15 pages

RR-92-49

Christoph Klauck, Ralf Legleitner, Ansgar Bernardi:
Heuristic Classification for Automated CAPP
15 pages

RR-92-50

Stephan Busemann:
Generierung natürlicher Sprache
61 Seiten

RR-92-51

Hans-Jürgen Bürckert, Werner Nutt:
On Abduction and Answer Generation through Constrained Resolution
20 pages

RR-92-52

Mathias Bauer, Susanne Biundo, Dietmar Dengler, Jana Koehler, Gabriele Paul: PHI - A Logic-Based Tool for Intelligent Help Systems
14 pages

RR-92-54

Harold Boley: A Direkt Semantic Characterization of RELFUN
30 pages

RR-92-55

John Nerbonne, Joachim Laubsch, Abdel Kader Digne, Stephan Oepen: Natural Language Semantics and Compiler Technology
17 pages

RR-92-56

Armin Laux: Integrating a Modal Logic of Knowledge into Terminological Logics
34 pages

RR-92-58

Franz Baader, Bernhard Hollunder:
How to Prefer More Specific Defaults in Terminological Default Logic
31 pages

RR-92-59

Karl Schlechta and David Makinson: On Principles and Problems of Defeasible Inheritance
14 pages

RR-92-60

Karl Schlechta: Defaults, Preorder Semantics and Circumscription
18 pages

RR-93-02

Wolfgang Wahlster, Elisabeth André, Wolfgang Finkler, Hans-Jürgen Profitlich, Thomas Rist:
Plan-based Integration of Natural Language and Graphics Generation
50 pages

RR-93-03

Franz Baader, Berhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, Enrico Franconi:
An Empirical Analysis of Optimization Techniques for Terminological Representation Systems
28 pages

RR-93-04

Christoph Klauck, Johannes Schwagereit:
GGD: Graph Grammar Developer for features in CAD/CAM
13 pages

RR-93-05

Franz Baader, Klaus Schulz: Combination Techniques and Decision Problems for Disunification
29 pages

RR-93-08

Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer: COLAB: A Hybrid Knowledge Representation and Compilation Laboratory
64 pages

RR-93-09

Philipp Hanschke, Jörg Würtz:
Satisfiability of the Smallest Binary Program
8 Seiten

DFKI Technical Memos
TM-91-12

Klaus Becker, Christoph Klauck, Johannes Schwagereit: FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM
33 Seiten

TM-91-13

Knut Hinkelmann: Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14

Rainer Bleisinger, Rainer Hoch, Andreas Dengel:
ODA-based modeling for document analysis
14 pages

TM-91-15

Stefan Busemann: Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

TM-92-01

Lijuan Zhang: Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

TM-92-02

Achim Schupeta: Organizing Communication and Introspection in a Multi-Agent Blocksworld
32 pages

TM-92-03

Mona Singh:
A Cognitive Analysis of Event Structure
21 pages

TM-92-04

Jürgen Müller, Jörg Müller, Markus Pischel, Ralf Scheidhauer:
On the Representation of Temporal Knowledge
61 pages

TM-92-05

Franz Schmalhofer, Christoph Globig, Jörg Thoben:
The refitting of plans by a human expert
10 pages

TM-92-06

Otto Kühn, Franz Schmalhofer: Hierarchical skeletal plan refinement: Task- and inference structures
14 pages

TM-92-08

Anne Kilger: Realization of Tree Adjoining Grammars with Unification
27 pages

DFKI Documents
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 Ditttrich, 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-92-22

Werner Stein: Indexing Principles for Relational Languages Applied to PROLOG Code Generation
80 pages

D-92-23

Michael Herfert: Parsen und Generieren der Prolog-artigen Syntax von RELFUN
51 Seiten

D-92-24

Jürgen Müller, Donald Steiner (Hrsg.):
Kooperierende Agenten
78 Seiten

D-92-25

Martin Buchheit: Klassische Kommunikations- und Koordinationsmodelle
31 Seiten

D-92-26

Enno Toltmann:
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

D-92-28

Klaus-Peter Gores, Rainer Bleisinger: Ein Modell zur Repräsentation von Nachrichtentypen
56 Seiten

D-93-01

Philipp Hanschke, Thom Frühwirth: Terminological Reasoning with Constraint Handling Rules
12 pages

D-93-02

Gabriele Schmidt, Frank Peters, Gernod Laufkötter: User Manual of COKAM+
23 pages

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 1993

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 applicable

Indexing Principles for Relational Languages Applied to PROLOG Code Generation.
Werner Stein

D-92-22
Document