Indexing PROLOG Procedures into DAGs by Heuristic Classification

Michael Sintek
DFKI
Postfach 2080
67608 Kaiserslautern
Germany

December 15, 1993

Abstract

This paper first gives an overview of standard PROLOG indexing and then shows, in a step-by-step manner, how it can be improved by slightly extending the WAM indexing instruction set to allow indexing on multiple arguments. Heuristics are described that overcome the difficulty of computing the indexing WAM code. In order to become independent from a concrete WAM instruction set, an abstract graphical representation based on DAGs (called DAXes) is introduced.

The paper includes a COMMON LISP listing of the main heuristics implemented; the algorithms were developed for RELFUN, a relational-plus-functional language, but can easily be used in arbitrary PROLOG implementations.

The ideas described in this paper were first presented at the Workshop "Sprachen für KI-Anwendungen, Konzepte – Methoden – Implementierungen" 1992 in Bad Honnef [SS92]. This paper is part of a collaborative work together with Werner Stein [Ste92].

Contents

I An Introduction to PROLOG Indexing 3

1 PROLOG and its Compilation into the WAM 3

2 Compiling a Single Clause 3

3 Combining Multiple Clauses Into a Procedure 4

4 Standard PROLOG Indexing 6

5 Motivation for Extensions of the Standard PROLOG Indexing 9

II DAXes: Indexing Information Represented in Specialized DAGs 10

6 Clauses and Fails 10

7 Choice Points 10

8 Index Instructions 11

9 Combining the DAX Components 11
9.1 `less` 11
9.2 `f` 12

III Extensions of the Standard PROLOG Indexing 13

10 Looking at Other Approaches 13
10.1 Hardware Oriented Approaches 13
10.1.1 m-in-n-Coding 13
10.2 Software Oriented Approaches 14
10.2.1 Complete Indexing 14
10.3 Index Assistant Functions 16
10.3.1 Shallow Backtracking 17
10.3.2 Quadratic Indexing 17

11 Our Approach 18
11.1 Using Arguments Other Than the First (1N) 19
11.2 Using More Than One Argument (MBN, MDN, and MN) 21

11.2.1 Breadth Oriented (MBN) 21
11.2.2 Depth Oriented (MDN) 22
11.2.3 Breadth and Depth Oriented (MN) 23
11.3 Allowing Variables in Index Blocks (1V and MV) 26
11.3.1 The 1V-Algorithm 27
11.3.2 The Final Result: The MV-Algorithm 28

12 Future Extensions 30
12.1 Using Additional Information 30
12.2 Assert 30
12.3 Compiling Higher Order PROLOG Extensions 30

IV Indexing in RELFUN 32

13 The RELFUN Implementation Structure 32

14 Compilation Phases 33
14.1 A Classifier with Indexing Heuristics 33
14.2 A Code Generator with Indexing Heuristics 35

15 Summary: Heuristic Classification 36

V Appendix 37

A User Commands 37

B Sample Session 38

C Benchmarks 45
C.1 Benchmark Results 45
C.2 Benchmark Sources 47
C.2.1 nrev Benchmark 47
C.2.2 dnf Benchmark 47
C.2.3 NET Benchmark 48

D Implementation of the Heuristics 51

Part I
An Introduction to PROLOG Indexing

1 PROLOG and its Compilation into the WAM

This paper will not give a complete description of compiling PROLOG into the WAM (Warren Abstract Machine); only those topics will be covered that are relevant to indexing. For more details on the WAM, refer to [War83], [GLLO85], [AK90], and [Nys85].

The WAM instruction set contains the following groups of instructions:

- instructions for register manipulations and unification
- control instructions (for "calling" subprocedures)
- choice instructions for combining clauses into a procedure (see section 3)
- indexing instructions
- instructions for extralogicals (such as the cut)

2 Compiling a Single Clause

The compilation of a single clause is not affected by the standard indexing method and the enhanced indexing methods described in this paper. In case you are not familiar with the WAM, the following small examples will give you an idea of how clauses are compiled.

Consider the following clause:

```
less(0,N).
```

Here is the WAM assembler code for the procedure:

```
less/2
    get_constant 0, X1
    proceed
```

`less/2` is the entry label for the procedure. The `/2` is needed no distinguish the binary relation `less` from, say, a unary procedure with the same name.

`get_constant` is one of the instructions used for unification. In this case, the register `X1`, which always contains the first argument passed to a procedure, is unified with the constant `0`: If `X1` is a free variable, `X1` is bound to the constant `0`; if `X1` is bound, this instruction "fails" iff `X1` is not bound to `0`, otherwise no action is taken.

For the (really anonymous) variable in the second argument no instruction is necessary.

The `proceed` instruction simply marks the end of a procedure. It acts quite similarly to the `return` instruction in conventional machine languages.

Compiling a rule is almost as simple as compiling a fact; the assembler code sequences for the head and the body are concatenated:

```
less(s(M), s(N)) :- less(M, N).
```

Here is the WAM assembler code for the rule:

```
less/2
    allocate 0              % allocate a new environment on the stack

    get_structure s/1, X1   % head: less(s(
    unify_variable X3       %              M),
    get_structure s/1, X2   %                  s(
    unify_variable X4       %                    N)) :-

    put_value X3, X1        % body:
    put_value X4, X2
    call less/2, 0          %                            less(M, N).

    deallocate              % remove the environment frame
    proceed
```

3 Combining Multiple Clauses Into a Procedure

The two clauses in the previous section define the binary relation `less`:

```
less(0,N).
less(s(M), s(N)) :- less(M, N).
```

The WAM code sequences for these two clauses can be combined without any changes to form the WAM code for the complete procedure:

```
less/2
    try_me_else 2
    get_constant 0, X1
    proceed

2   trust_me_else_fail
    allocate 0

    get_structure s/1, X1   % less(s(
    unify_variable X3       %        M),
    get_structure s/1, X2   %            s(
    unify_variable X4       %              N)) :-

    put_value X3, X1        %
    put_value X4, X2
    call less/2, 0          %                      less(M, N).

    deallocate
    proceed
```

Three WAM instructions are needed for combining clauses in this way:

`try_me_else` *L:* allocate a new choice point frame on the stack setting its next clause field to *L*

`retry_me_else` *L:* having backtracked to the current choice point, reset all the necessary information from it and update its next clause field to *L*

`trust_me_else_fail` *L:* having backtracked to the current choice point, reset all the necessary information from it, discard it, and reset the latest choice point (the `B` register) to its predecessor

It is not necessary for the reader to understand the way these instructions work internally. It is only important to realize that for all queries and calling procedures always *all* clauses of a procedure are ultimately "tried".

For instance, the query `less(0,s(0))` compiles to

```
    put_constant 0, X1
    put_structure s/1, X2
    unify_constant 0
    execute less/2
```

It first tries the fact (succeeding) and on backtracking tries the rule (failing).

Preparing the use of indexing *header* code in the next section, let us note that `try` L, `retry` L, `trust` L can be used instead of `try_me_else` L, `retry_me_else` L, and `trust_me_else_fail` L by the following equivalence:

```
        try A                      try_me_else V1
        retry B1               A   ...
        ...                    V1  retry_me_else V2
        retry Bn               B1  ...
        trust C
                               ...
    A   ...
    B1  ...                    Vn  retry_me_else W
    ...                        Bn  ...
    Bn  ...                    W   trust_me_else_fail
    C   ...                    C   ...
```

4 Standard PROLOG Indexing

If all arguments in a query or a calling predicate are variables, then there is clearly no better way to proceed other than in the above way. On the other hand, when some of the arguments are at least partially instantiated, that information can be used to skip all (or at least some of) those clauses that do not fit these arguments. In analogy to databases, techniques to achieve this are summarized as “indexing” techniques.

The main difference between database and PROLOG indexing is that the former handles a *set* of items while the latter deals with a (textually ordered) *sequence* of items (since PROLOG clauses are tried from top to bottom).

The standard PROLOG indexing method described in [War83], [GLLO85], and [AK90] uses the first argument of each procedure for indexing.

In the `less` example, the first clause has to be tried only if the first argument is the constant `0` or a free variable. Analogously, the second clause has only to be tried if the first argument is a unary structure with functor `s` or is a free variable.

The WAM instruction set must therefore include an instruction to determine the type of an argument. This instruction is called `switch_on_term`. It takes as many arguments as there are types in PROLOG (e.g. constants, structures, lists, and empty lists) plus one argument for free variables:

`switch_on_term` *Const, Struct, List, Nil, Var*.

All these arguments are labels to jump at if the first procedure argument has the corresponding type.

In case of constants and structures, the constants and the functors can also be used for indexing, thus two more switching instructions are used: `switch_on_constant` N, T and `switch_on_structure` N, T where T is a hash table of size N containing entries of the form *constant:label* or *structure/arity:label*. Actual constants and structures not appearing in the hash table lead to failure.

Replacing the try instructions by these switching instructions in the `less` example, the following WAM assembler code results:

```
less/2
    switch_on_term const, struct, fail, fail, var
const   % X1 must here be *some* constant
    switch_on_constant 1, {0:1}     % jump to clause 1 if X1 = 0
struct  % X1 must here be *some* structure
    switch_on_structure 1, {s/1:2}  % jump to clause 2 if X1 = s(...)
    % else fail
var          % jump to both clauses if X1 is a free variable:
    try 1    % first try clause with label 1,
    trust 2  % then the clause with label 2

1   get_constant 0, X1
    proceed

2   allocate 0
    get_structure s/1, X1   % less(s(
    unify_variable X3       %        M),
    get_structure s/1, X2   %            s(
    unify_variable X4       %              N)) :-
    put_value X3, X1        %
    put_value X4, X2
    call less/2, 0          %                      less(M, N).
    deallocate
    proceed
```

Hassan Aït-Kaci in [AK90] called this the *three-level-indexing scheme*:

level		WAM instructions
I	discrimination on type (constant, structure, list, empty list, and variable)	`switch_on_term`
II	discrimination on value (only for constants and structures)	`switch_on_constant` `switch_on_structure`
III	enumeration of clauses	`try`, `retry`, `trust`

If the first argument of a procedure contains variables, one has to divide the procedure into several "blocks" or "partitions"[1], i.e. maximal three-level-indexable subportions of a procedure either having a variable as the first argument (one-clause blocks) or not (general blocks). The following procedure has to be split into four blocks:

```
f(1,a).   % block 1
f(2,b).

f(X,X).   % block 2

f(X,d).   % block 3

f(3,e).   % block 4
f(4,f).
```

Blocks 1 and 4 can be compiled using the above described indexing instructions, blocks 2 and 3 are compiled straight forward. The four blocks are then glued together by the `try`, `retry`, and `trust` instructions:

```
f/2 try block1
    retry block2
    retry block3
    trust block4
```

Together with the discrimination on name and arity, which can also be viewed as part of the indexing, we now have a *five-level-indexing scheme*:

level		WAM instructions
N	discrimination on name and arity	`call` and `execute`
B	enumeration of blocks	`try`, `retry`, `trust`
I	discrimination on type (constant, structure, list, empty list, and variable)	`switch_on_term`
II	discrimination on value (only for constants and structures)	`switch_on_constant` `switch_on_structure`
III	enumeration of clauses	`try`, `retry`, `trust`

[1] both terms are used interchangeably in this paper

5 Motivation for Extensions of the Standard PROLOG Indexing

The standard indexing method is only useful for procedures with a database-like structure, i.e. the first argument is a key (or at least a quasi-key: practically all constants are different, there a hardly any variables):

```
p(c1, ...) :- ... .
p(c2, ...) :- ... .
       ...
p(cn, ...) :- ... .
```

Thus the standard indexing method does not work in the following cases:

1. the quasi-key is not the first argument of the procedure
2. the procedure can be split into several blocks each having another argument as a quasi-key
3. the quasi-key is spread over several arguments
4. there is more than one argument (group) that could serve as a quasi-key (this is important if the argument that is best suited for indexing is rarely instantiated in calls)
5. some combinations of cases 1–3 with case 4

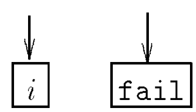
Part II

DAXes: Indexing Information Represented in Specialized DAGs

In order to avoid further elaboration on concrete WAM indexing instructions, an abstract graphical representation of the indexing instructions will be used, namely DAXes: **d**irected **a**cyclic digraphs for inde**x**ing. The following sections describe the various DAX components.

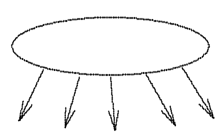
6 Clauses and Fails

For all indexing methods proposed in this paper, the WAM code for a single clause is not relevant. Therefore, a clause is represented by a box containing only the number (label) of the clause. Similarly, a fail is represented by a box containing `fail`.

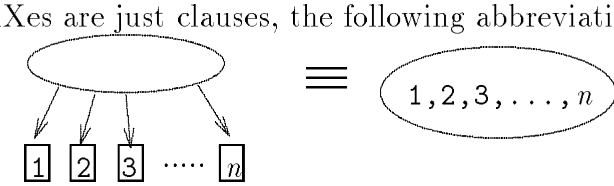


7 Choice Points

When combining multiple clauses into a procedure, they are connected via the `try_me_else`, `retry_me_else`, `trust_me` or, equivalently, the `try`, `retry`, `trust` instructions (see sections 3 and 4). Such a choice point is abstractly represented by an oval with (left-to-right ordered) outgoing arrows:



If all sub-DAXes are just clauses, the following abbreviation is used:



8 Index Instructions

The index instructions have the following graphical representations:

- `switch_on_term` *Const, Struct, List, Nil, Var*:

type *n*				
Const	Struct	List	Nil	Var

  The `type` *n* means switching on the type of the *n*th argument[2] (see section 11.1).

- `switch_on_constant` *N, T*:

const *n*					
c1	*c2*	*c3*	...	*cN*	else

  where the hash table *T* has the *N* entries *c1, c2, ..., cN*; the `else` label will be explained in section 11.3

- `switch_on_structure` *N, T*:

struct *n*					
s1	*s2*	*s3*	...	*sN*	else

  where the hash table *T* has the *N* entries *s1, s2, ..., sN*

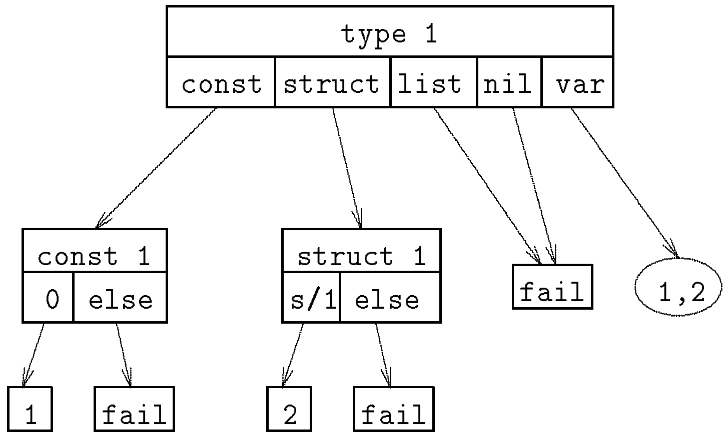
9 Combining the DAX Components

In order to show how to combine the introduced DAX components, the two examples of section 4, `less` and `f`, are used.

9.1 less

```
less(0,N).                           % clause 1
less(s(M), s(N)) :- less(M, N).      % clause 2
```

[2]thus an extension of the standard WAM switching instructions is needed on the concrete level: either (as in the KCM [BBB+89]) add an additional argument to all three switching instructions, or (as in our approach) add one new instruction (`set_index_number` *n*; see appendix B) to change the value of an index register (`IX`) which is looked up by the switch instructions



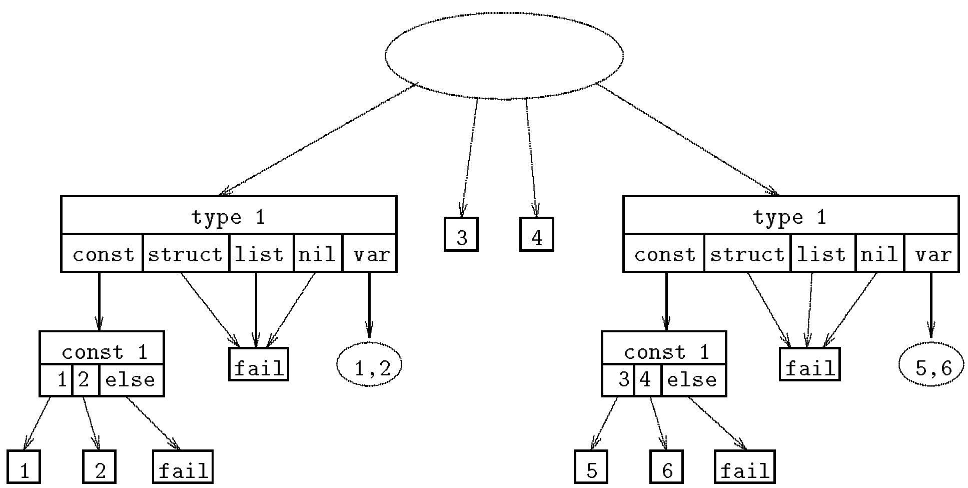
9.2 f

```
f(1,a).   % clause 1
f(2,b).   % clause 2

f(X,X).   % clause 3

f(X,d).   % clause 4

f(3,e).   % clause 5
f(4,f).   % clause 6
```



Part III
Extensions of the Standard PROLOG Indexing

10 Looking at Other Approaches

In this section we provide an overview of several indexing schemes which is a slightly revised version of section 6 in [Ste92]. They can be distinguished into *hardware oriented* and *software oriented* approaches.

The hardware oriented approaches are based on database techniques. A hash-function returns, for a given query, a set of clauses as potential matches. This is done separately from the compilation of the program, so clauses[3] (maybe a very large number of clauses) can be stored separately (e.g. externally).

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

10.1 Hardware Oriented Approaches

Several indexing methods are based on bit-matrix representations of clauses in a procedure. They are *field encoding*, *superimposed coding with embedded position and variables*, and *superimposed coding with external variables* [HM89].

All these are based on the principle of *n-in-m-coding* which is described in the next section.

10.1.1 m-in-n-Coding

In this method the value of an attribute is compressed into a binary word of *width* n with a fixed 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 three possibilities to do this are proposed.

The main advantage of this method is that one can currently construct hardware that handles up to 8.000 clauses and more in the presented manner. Together with the linear searching hash-function one reaches 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 stored separately (externally) from the small index code and only single rules are loaded.

[3] mainly facts

10.2 Software Oriented Approaches

In contrast to the hardware oriented approaches, the software oriented approaches do not use a hash-function returning a *set* of potentially matching clauses, but the program flow sequentially *enumerates* all those clauses. For this reason the index code and the clause code become scattered over the program code.

In section 4 standard WAM indexing was explained. A much more complex indexing mechanism, complete indexing, is introduced in the next section.

10.2.1 Complete Indexing

In [HM89] 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`[4]. If we assume that the program `p` is also called with other modes, the compiler will produce other specialized procedures for these modes.

The transformed source program 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`$(T_1,\ldots,T_n,Z_1,\ldots,Z_m):-$
2. $P_1,\ldots,P_r$
3. $Z_1=S_1,\ldots,Z_m=S_m,B_1,\ldots,B_s.$

Where:

$$T_i \equiv \text{arguments with mode } \textit{constant}$$

$$S_i \equiv \text{arguments with mode } \textit{don't know}$$

$$Z_i \equiv \text{new variables not yet occurring in the clause}$$

[4] `c` stands for *constant* and `d` for *don't know*

$$P_i \equiv \begin{cases} \text{Primitives:} \\ \text{goals without side effects and} \\ \text{whose parameters are known to} \\ \text{be ground after head unification} \end{cases}$$

$$B_i \equiv \begin{cases} \text{either a non-primitive goal or} \\ \text{causing side effect or with un-} \\ \text{bound arguments after head uni-} \\ \text{fication} \end{cases}$$

The generated indexing code is in some sense also a three level indexing (c.f. section 4) of the following form:

level	
I	indexing head code
II	indexing primitive code
III	enumeration of clauses

The first one is a sequential 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 WAM branch instructions (e.g. `if_gt`, `if_eq`, `if_le`), so control jumps to a given label based e.g. on arithmetical comparisons.

The indexing bodies are compiled with the standard WAM techniques.

level	instruction	arguments
I	`g_switch`	2: argument-number and list of tuples (atom link)

II	`if_gt`
	`if_eq`
	`if_le`
	`atomic`
	`functor`
	...
		2–3:
		test-arguments (1–2) and true link

III	`try`
	`retry`
	`trust`
		1: label

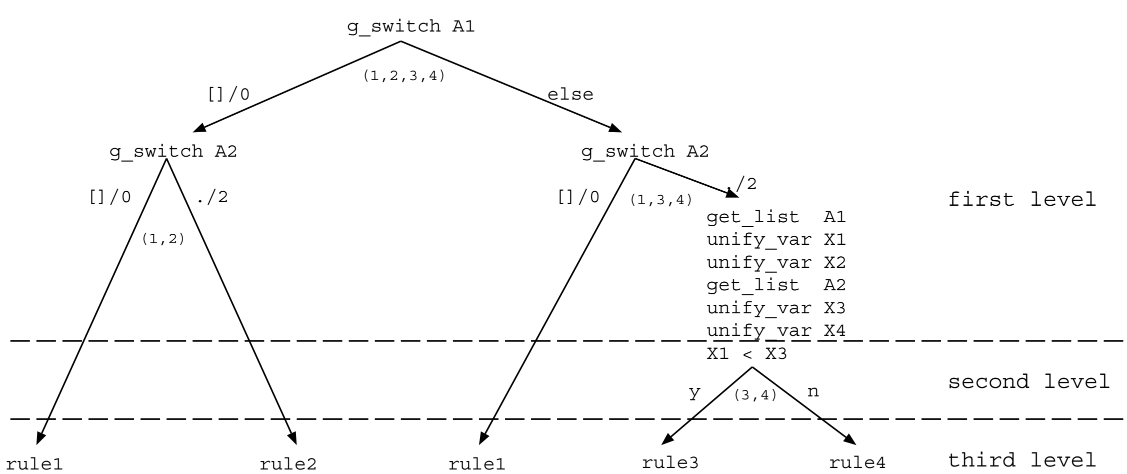
Example:

```
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) :- X1=L.
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:



10.3 Index Assistant Functions

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

10.3.1 Shallow Backtracking

This approach can only be applied to primitive deterministic[5] procedures.

While unification in the head and the primitive body code takes place, only a link to the next alternative clause is needed as backtrack information because no heap variables are bound, no non-primitive goal in the body will be called, and no side effects will occur. On the other hand, after successful unification in 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.

10.3.2 Quadratic Indexing

Another approach for 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).

[5] *primitive deterministic* is an extended definition of *head deterministic* which looks not only at the clause heads but also at the primitive instructions at the beginning of the bodies

11 Our Approach

The next sections will describe our approach. Instead of directly presenting our final indexing technique, its components are introduced in order of increasing complexity.

The following part of a PROLOG program[6] (a normalizer producing DNFs of propositional formulas) is used to demonstrate our heuristics for generating index trees:

```
norm(X, X) :- literal(X).
norm(or(X, Y), or(X, Y)) :- literal(X), literal(Y).
norm(and(X, Y), and(X, Y)) :- literal(X), literal(Y).

norm(or(X, Y), or(X1, Y)) :-
        literal(Y),
        norm(X, X1).

norm(or(X, or(Y, Z)), W) :-
        norm(or(or(X, Y), Z), W).

norm(or(X, and(Y1, Y2)), or(X1, Y12)) :-
        norm(X, X1),
        norm(and(Y1, Y2), Y12).

norm(and(X, Y), and(X1, Y)) :-
        literal(Y),
        norm(X, X1).

norm(and(X, and(Y, Z)), W) :-
        norm(and(and(X, Y), Z), W).

norm(and(X, or(Y1, Y2)), and(X1, Y12)) :-
        norm(X, X1),
        norm(or(Y1, Y2), Y12).
```

Only the following information (entirely extracted from the clause heads[7]) is used for the index tree generation; all algorithms in this paper can easily be

[6]see also appendices B and C.2.2

[7]if a variable in the head is directly bound to a constant or structure in the body before any other subgoals, this information can also be used (e.g. in `p(X) :- X = 5, q(6)`); anyway, RELFUN's normalizer would move such body goals into the head (e.g. obtaining `p(5) :- q(6)`)

extended to use additional information such as inner structure arguments and "guards"[8] (see section 12.1):

#		Arg 1		Arg 2	
1	*norm*(	X	,	X	)
2	*norm*(	or/2	,	or/2	)
3	*norm*(	and/2	,	and/2	)
4	*norm*(	or/2	,	or/2	)
5	*norm*(	or/2	,	W	)
6	*norm*(	or/2	,	or/2	)
7	*norm*(	and/2	,	and/2	)
8	*norm*(	and/2	,	W	)
9	*norm*(	and/2	,	and/2	)

The following sections describe the heuristics for our indexing techniques:

- 1N-Algorithm: One Argument / No Variables
- MN-Algorithm: Multiple Arguments / No Variables
  - MBN-Algorithm: MN-Algorithm / Breadth Oriented
  - MDN-Algorithm: MN-Algorithm / Depth Oriented
- 1V-Algorithm: One Argument / Variables
- MV-Algorithm: Multiple Arguments / Variables

11.1 Using Arguments Other Than the First (1N)

In this first generalization of the standard indexing technique (indexing on the first argument) only one variable-free argument column in each indexing partition is used (this argument column need not be the same in all partitions).

The heuristics for finding the partitions is the following simple greedy (don't-care-choice) algorithm (*1N-Algorithm*):

1. For each argument column i, count the number of non-variable arguments down to the first variable or the end of the procedure ($NV[i]$)
2. $maxNV := \max_i(NV[i])$
3. If $maxNV = 0$ then use the first clause as a separate partition (without indexing) else
   - $maxCOLS := \{i | NV[i] = maxNV\}$

[8] side-effect free builtins (<,>,...)

- if $maxCOLS = \{k\}$ then $COL := k$ else choose $COL \in maxCOLS$ with the most selective[9] elements
- use the first $maxNV$ clauses as a partition and index them on the COL^{th} argument

4. If any clauses are left go to 1 to form further partitions else stop

Using this algorithm on the example, the following two partitions are formed:

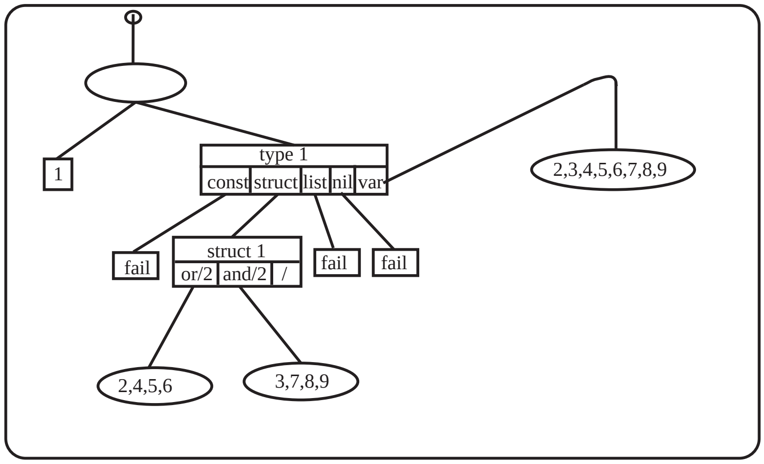
1. - $NV[1] = NV[2] = 0$
   - $maxNV = 0$
   - use first clause without indexing
   - go to 1 with clauses 2 – 9

2. - $NV[1] = 8$, $NV[2] = 3$
   - $maxNV = 8$
   - $maxCOLS = \{1\}$
   - use clauses 2 – 9 with indexing on 1^{st} argument

#		Arg 1		Arg 2		Idx
1	*norm*(	X	,	X	)	-
2	*norm*(	or/2	,	or/2	)	
3	*norm*(	and/2	,	and/2	)	
4	*norm*(	or/2	,	or/2	)	
5	*norm*(	or/2	,	W	)	1
6	*norm*(	or/2	,	or/2	)	
7	*norm*(	and/2	,	and/2	)	
8	*norm*(	and/2	,	W	)	
9	*norm*(	and/2	,	and/2	)	

Resulting index tree ("`/`" in `else` field is used here as a shortcut for a pointer to `fail`; arcs are directed in the natural top-to-down order):

[9] selectivity is the number of different constants and functors



11.2 Using More Than One Argument (MBN, MDN, and MN)

Multiple arguments can be used in two different ways for indexing:

1. When the indexing argument is unbound, use the "best" of the remaining ones (e.g., if in the above example *norm* is called with the first argument unbound, try indexing on the second) (⇒ *index tree breadth, MBN-Algorithm*)

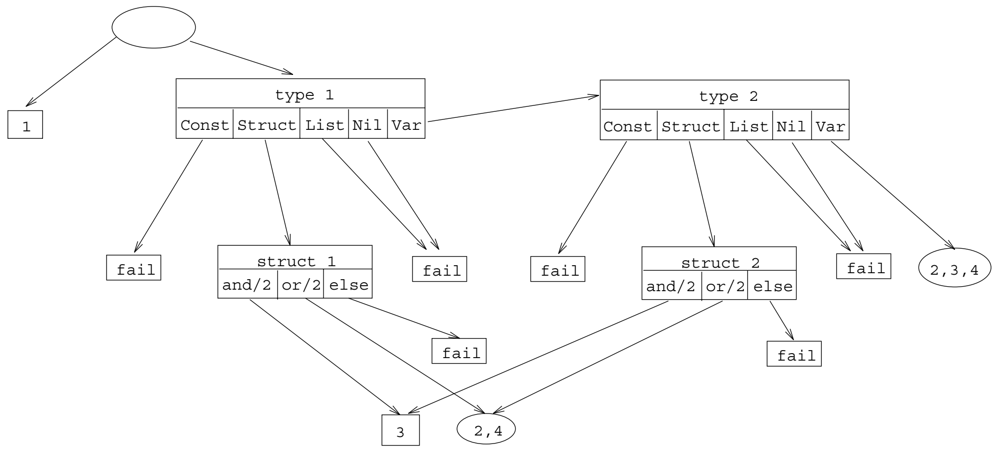
2. When the argument that can be used for indexing selects many clauses, view these clauses as a new procedure and index it recursively (e.g., if *norm* is called with and/2 as the first argument, form a procedure from clauses 3,7,8,9 and index it on argument column 2 (for the second partition)) (⇒ *index tree depth, MDN-Algorithm*)

The *MBN-Algorithm* together with the *MDN-Algorithm* form the *MN-Algorithm*, which is explained in detail in section 11.2.3. The results of the *MBN-Algorithm* and *MDN-Algorithm* applied to the *norm* example should be intuitively clear and are presented in the next two sections.

11.2.1 Breadth Oriented (MBN)

For simplicity, we consider only the following part of the *norm* example:

#		Arg 1		Arg 2		Idx
1	*norm*(	X	,	X	)	-
2	*norm*(	or/2	,	or/2	)	
3	*norm*(	and/2	,	and/2	)	1+2
4	*norm*(	or/2	,	or/2	)	



Note that the `Var` case of the `type 1` node points ('over') to the `type 2` node ('breadth') under the assumption that the second query argument may be non-variable.

That the `struct 1` and `struct 2` nodes have the same outgoing arrows[10] is a consequence of the example's structurally identical first and second arguments[11].

11.2.2 Depth Oriented (MDN)

For simplicity, we consider only the following part of the *norm* example:

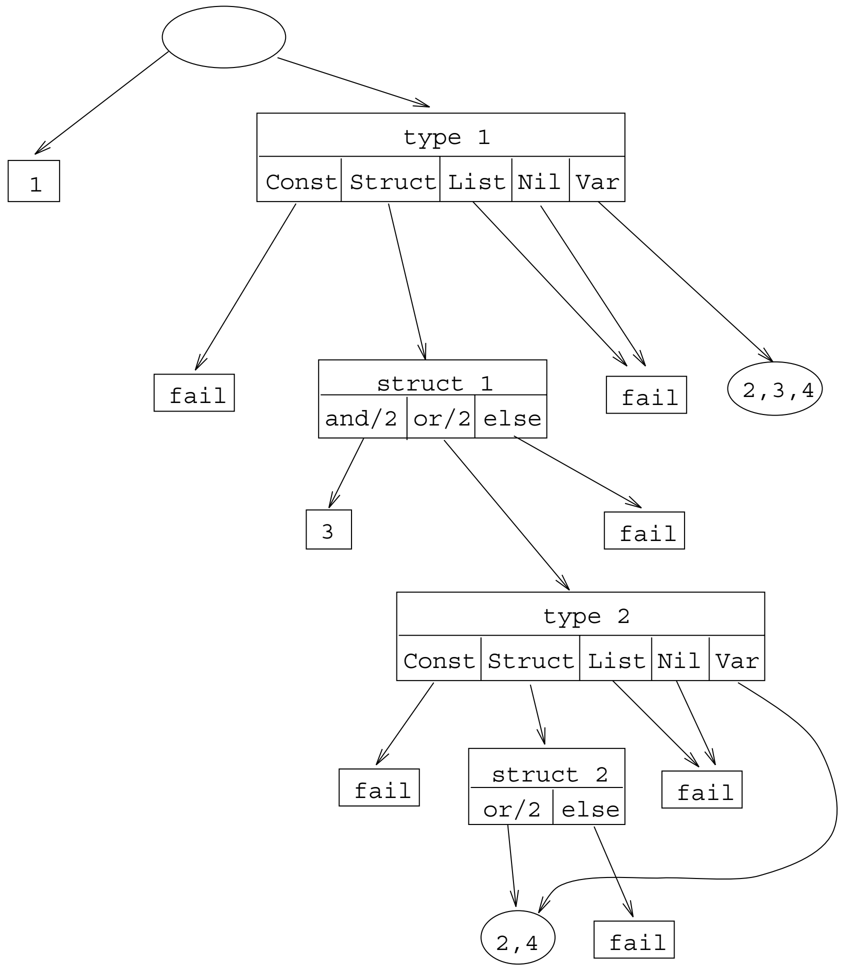
#		Arg 1		Arg 2		Idx
1	*norm*(	X	,	X	)	-
2	*norm*(	or/2	,	or/2	)	
3	*norm*(	and/2	,	and/2	)	1
4	*norm*(	or/2	,	or/2	)	

`or/2` occurs two times in the first argument column; viewing the selected clauses as a new procedure:

#		Arg 2		Idx
2	*norm*(	or/2	)	(2)
4	*norm*(	or/2	)	

[10] `and/2` and `or/2` of the hash table are presented here in the opposite order of earlier examples, which if of course immaterial

[11] in future DAXes *layout* will occasionally enforce copying; in our *implementation*, identical sub-DAXes are always shared (see [Ste92])



Note that the `or/2` case of the `struct 1` node points ('down') to the `type 2` node ('depth') under the assumption that the second query argument may further index the `or/2`-sub-procedure.

That this assumption is false (clauses 2 and 4 cannot be discriminated) is again due to the structurally identical first and second arguments of the example.

11.2.3 Breadth and Depth Oriented (MN)

The following algorithm (*MN-Algorithm*) combines the MNB- and MND-Algorithms:

1. For each argument column i, create a list $NL[i]$ where $NL[i]$ is the longest prefix of column i without variables

2. If $\forall_i\ NL[i] = ()$ then use the first clause as a separate partition (without indexing) else

   - sort the $NL[i]$ in descending order (w.r.t. their length) into the list SL

- $maxNL$:= length of first element in SL
- $lastCol$:= position of last column in SL with length $\geq maxNL \cdot c$ with $c \approx 0.7$ (this means that in order to enlarge the index tree breadth the partition size may be reduced, e.g. by at most 30% $\equiv c = 0.7$)[12];
  $maxNL'$:= length of this column;
  SL' = first $lastCol$ elements of SL;
  reorder SL' w.r.t. selectivity[13]
- create a partition consisting of the first $maxNL'$ clauses; index the argument columns in SL' ($\Rightarrow$ index tree breadth)
- for each constant/functor occurring multiply in one argument column of this partition do
  - form a procedure containing all selected clauses and the remaining argument columns in SL' (only columns to the right of the current one)
  - apply the MN-Algorithm recursively to this procedure ($\Rightarrow$ index tree depth)

3. If any clauses are left go to 1 else stop

MN-Algorithm applied to *norm* example:

1.
   - $NL[1] = NL[2] = ()$
   - use clause 1 as first partition
2.
   - $NL[1] = (or/2, and/2, or/2, \ldots, and/2)$
   - $NL[2] = (or/2, and/2, or/2)$
   - $SL = (NL[1], NL[2])$
   - $maxNL = 8$
   - $lastCol = 1$ ($NL[2]$ is too short, thus index tree breadth = 1)
   - $SL' = (NL[1])$
   - second partition consists of clauses 2 – 9, indexing takes place on first argument
   - and/2 occurs four times in indexing column:
     - form procedure from selected clauses:

[12] of course this constant could be easily changed
[13] see section 11.1

#	Arg 2
3	*norm*(and/2)
7	*norm*(and/2)
8	*norm*(W)
9	*norm*(and/2)

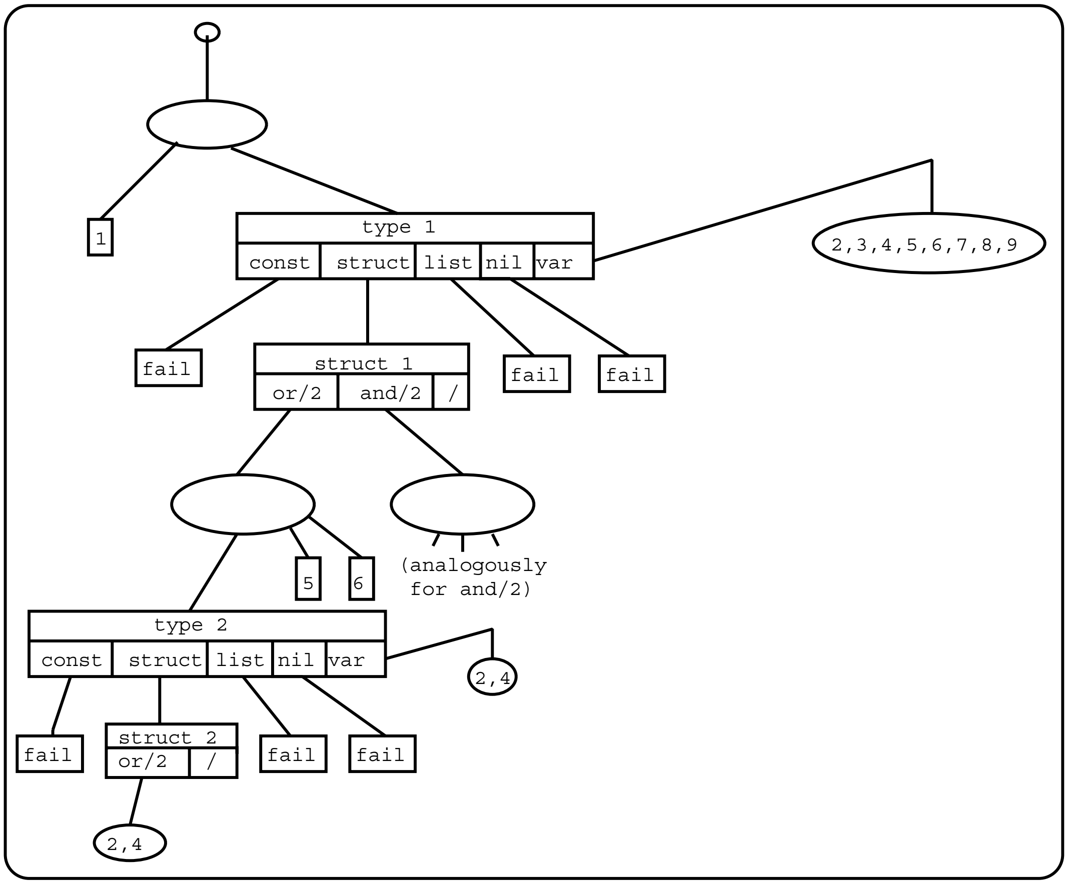
– applying MN-Algorithm to this procedure:

#	Arg 2	Idx
3 7	*norm*(and/2) *norm*(and/2)	(2)
8	*norm*(W)	-
9	*norm*(and/2)	-

- or/2 occurs four times in indexing column; result analogously to and/2:

#	Arg 2	Idx
2 4	*norm*(or/2) *norm*(or/2)	(2)
5	*norm*(W)	-
6	*norm*(or/2)	-

Resulting index tree:



11.3 Allowing Variables in Index Blocks (1V and MV)

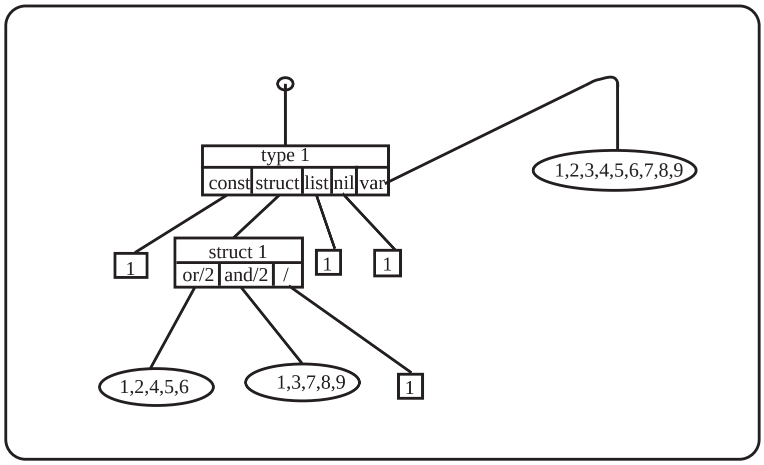
In order to obtain larger and thus more efficient partitions (w.r.t. time), indexed argument columns should be allowed to contain some variables. If, for example, an argument column contains the sequence (1,2,X,2,1), it makes sense to form a single partition from all 5 clauses; if a 1 is presented to this partition, only the clauses 1,3,5 have to be tried. If a constant other than 1 or 2 or any structure or list is presented to this partition, the third clause has to be tried. The standard `switch_on_constant` and `switch_on_structure` instructions cannot handle this situation which made it necessary to add the `else` argument to these instructions.

The algorithms for generating index trees with variables allowed in partitions (1V- and MV-Algorithms) can easily be obtained from the 1N-Algorithm and MN-Algorithm by simply replacing the restriction "no variables" by "at most a number BVS and percentage BVP % of variables".

BVS is called the *block variable size* and specifies the maximal number of variables an argument column of a partition is allowed to contain; BVP is the maximal portion (in %) of variables in a partition's argument column.

11.3.1 The 1V-Algorithm

The 1V-Algorithm is subsumed be the MV-Algorithm; only the result of using it on our *norm* example is presented[14] (this can be regarded as being obtained from the '1N' DAX in section 11.1 by propagating the branch for clause 1 down to the leaves of the second partition, overwriting fail nodes):



[14] BVS and BVP unrestricted

11.3.2 The Final Result: The MV-Algorithm

1. For each argument column i, create a list $NL[i]$ where $NL[i]$ is the longest prefix of column i with at most a number BVS and percentage BVP % of variables

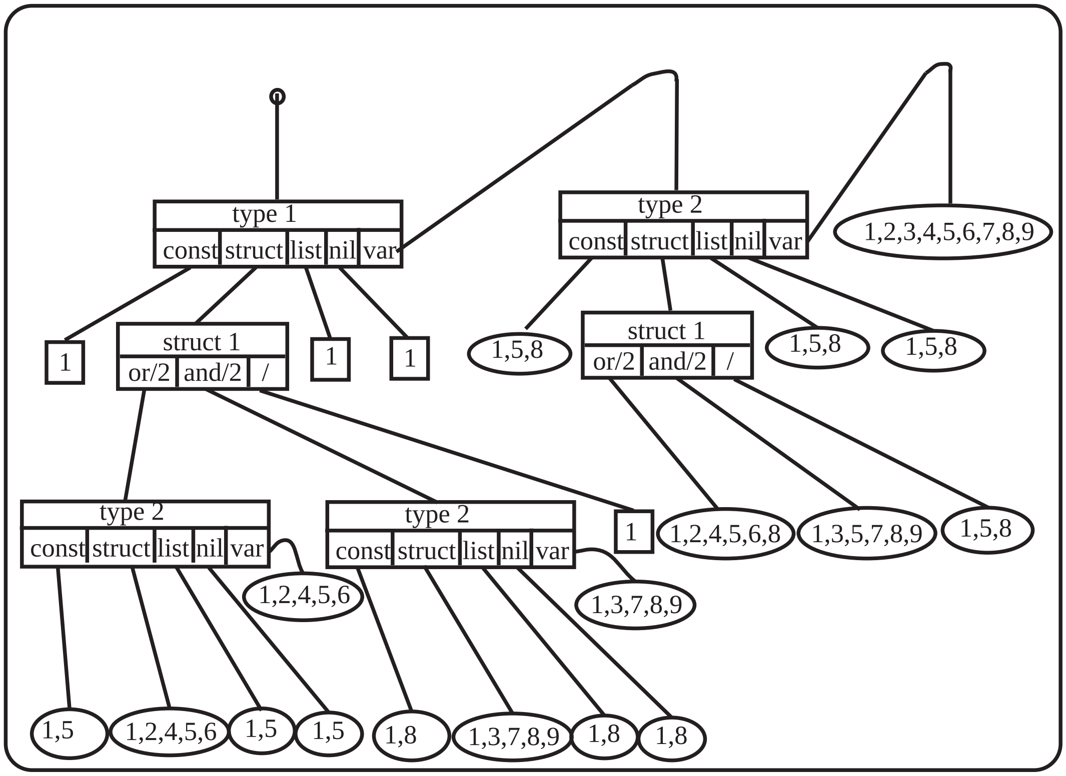
2. If $\forall_i \; NL[i] = ()$ then use the first clause as a separate partition (without indexing) else

   - sort the $NL[i]$ in descending order (w.r.t. their length) into the list SL
   - $maxNL$:= length of first element in SL
   - $lastCol$:= position of last column in SL with length $\geq maxNL \cdot c$ with $c \approx 0.7$;
     $maxNL'$:= length of this column;
     SL' = first $lastCol$ elements of SL;
     reorder SL' w.r.t. selectivity[15]
   - create a partition consisting of the first $maxNL'$ clauses; index the argument columns in SL' ($\Rightarrow$ index tree breadth)
   - for each constant/functor occurring multiply in one argument column of this partition do
     - form a procedure containing all selected clauses and the remaining argument columns in SL' (only columns to the right of the current one)
     - apply the MV-Algorithm recursively to this procedure ($\Rightarrow$ index tree depth)

3. If any clauses are left go to 1 else stop

Result of using the MV-Algorithm on our *norm* example:

[15]see section 11.1



In the above DAX, some sub-DAXes were pruned in order to reduce memory consumption. This pruning is performed by the pruning algorithm explained in [Ste92][16].

The benchmarks in appendix C give you an impression of the efficiency gains of the MV-Algorithm.

[16] the pruning can be influenced by the `indexing :max-args <n>` and `indexing :max-depth <n>` commands in RELFUN which are described in appendix A

12 Future Extensions

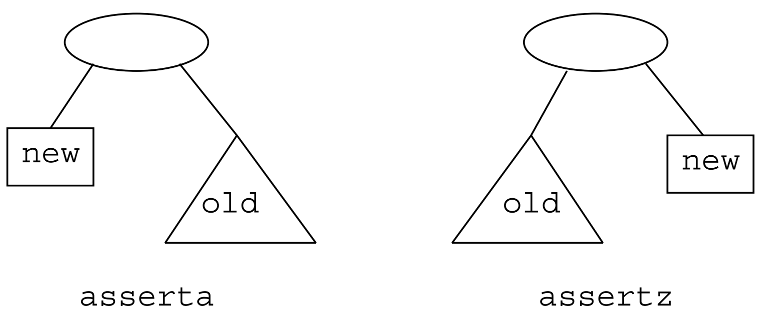
12.1 Using Additional Information

In addition to constants, functors, and lists (as described in section 11), the following indexing information can be used:

- inner structure arguments: the above heuristics do not have to be changed; simply form pseudo-argument columns of inner structure positions
- guards: side-effect free builtins can be extracted from a clause and mixed with the indexing code (c.f. section 10.2.1)
- modes (declared or inferred): can be used to exclude output argument columns and to prefer input argument columns

12.2 Assert

Instead of recompiling a procedure when additional clauses are asserted at its front or end, one can simply add the new clauses at the top of the index tree:



This method results in a loss of (time) efficiency when too many clauses are asserted because these new clauses are not indexed. Still, in that case only the header code for the index tree has to be reorganized; the old clauses themselves need not to be recompiled.

12.3 Compiling Higher Order PROLOG Extensions

In [Bol90] Harold Boley described how to reduce higher-order RELFUN clauses to constant-operator clauses.

The second-order characteristics of the constant-operator fact

```
transitive(ancestor).
```

is dependent on `ancestor`'s use as a first-order relation:

```
Rel(A,C) :- transitive(Rel), Rel(A,B), Rel(B,C).
```

Higher-order procedures like this cannot be directly compiled into the WAM, but a simple transformation of all clause heads and goals allows compilation:

$$h_0(h_1, \ldots, h_k) \rightarrow \mathtt{ap}(h_0, h_1, \ldots, h_k)$$

For the above example, this transformation results in[17]

```
ap(transitive, ancestor).
ap(Rel,A,C) :- ap(transitive,Rel), ap(Rel,A,B), ap(Rel,B,C).
```

With the standard PROLOG indexing, a significant loss of efficiency results because indexing on only the first argument selects the clauses just by their procedure name but does not look at their (real) arguments. The MN- and MV-Algorithms overcome this problem by looking at all arguments (see section 11.2).

[17] a more efficient alternative to this transformation is implemented as part of RELFUN's compilation laboratory

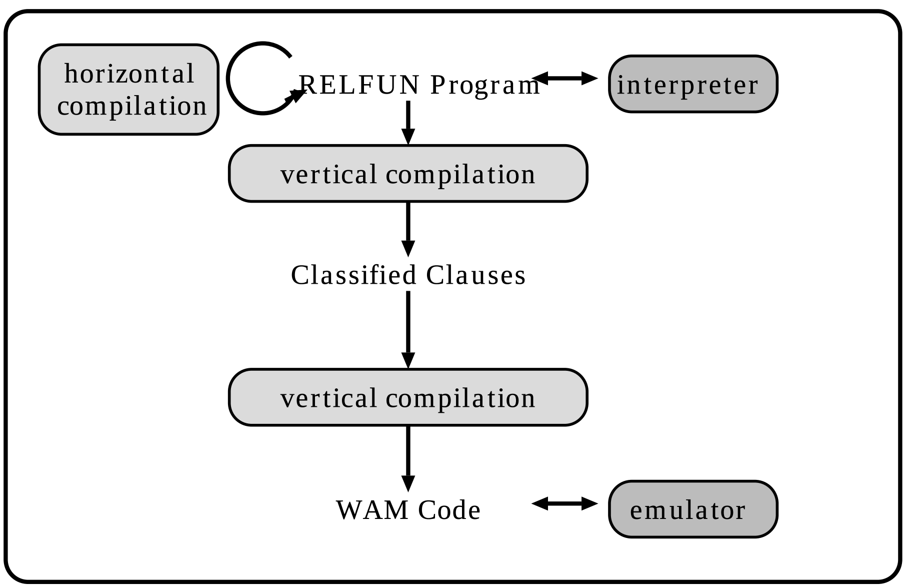
Part IV
Indexing in RELFUN

13 The RELFUN Implementation Structure

Although RELFUN provides both relational and functional clauses [Bol90], for the purpose of indexing it can be regarded as a kind of PROLOG since indexing affects the clause head and perhaps some body premises ("guards"), but never the (functional) foot.

The compilation task is divided into several horizontal[18] and vertical[19] 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.

One of the most important features of the RELFUN 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 [BEHK91, Kra90, Kra91] 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 this intermediate language. So one modification had to take place in the first vertical compilation step between the RELFUN program and the classified clauses.

[18] source to source
[19] source to code

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 ([Nys85]), a LISP implementation of the WAM ([War83]), good for rapid prototyping and experimental extensions. It was changed for handling RELFUN's functional extensions by Hans-Günther Hein (see [Hei89]).

14 Compilation Phases

14.1 A Classifier with Indexing Heuristics

The result of the MV-Algorithm that enriches the classified clauses by heuristic indexing information is described by the following EBNF:

```
classified-procedure ::=
  (proc <name>/<n>                    ; <n> is the arity
        <number-of-clauses>
        <indexing>
        <classified-clause-1>
        ...
        <classified-clause-n> )

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

iblock ::= <pblock> | <sblock>

pblock ::= (pblock <rblock> { <sblock> | <1block> }+ )

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>)

1block ::= (1block <clauses> { arg-col }+ )

sblock ::= (sblock <rblock> <seqind> [ <pblock> ] )

seqind ::= (seqind { <seqind-arg> }+ )

seqind-arg ::= (arg <arg-number>
                    (info <inhomogenity>)
                    <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> ] )
```

Explanations:

- iblock = indexed block
- pblock = partition block
- sblock = standard index block
- 1block = 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

For further details and an example, refer to appendices B and D.

14.2 A Code Generator with Indexing Heuristics

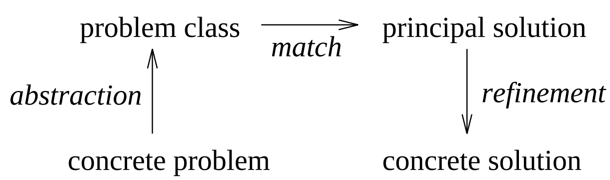
Code generation, the second part of our implementation, is working below the level of the classified clauses and is described in detail in [Ste92]. Its main task is the generation of indexing WAM code from the indexing information in the classified clauses.

15 Summary: Heuristic Classification

There is a more global sense (than that of section 14.1) in which this paper combines heuristics and classification, providing a good scheme for this summary section.

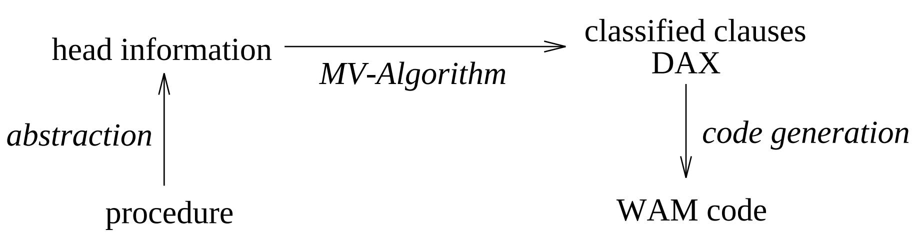
In [Cla85] *heuristic classification* has been identified as a widespread problem solving method. Heuristic classification is comprised of three main phases:

1. *abstraction* from a concrete, particular problem description to a problem class,
2. *heuristic match* of a principal solution (method) to the problem class, and
3. *refinement* of the principal solution to a concrete solution for the concrete problem.



These phases can be correlated with the phases in our indexing scheme:

1. abstraction from a RELFUN procedure resulting in the relevant head information, the argument columns (see section 11 and the function `icl.mk-it-head` ("make index type head" in appendix D),
2. applying the MV-Algorithm (or one of the other heuristics) resulting in a DAX, and
3. using the code generator to produce the concrete solution, the WAM code.



Part V
Appendix

A User Commands

Since indexing should be automatic the index structure is hidden from the RELFUN user. The only command 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/functor block to <no> (BVS[20], *block variable size*, see section 11.3)
- `:max-depth` <no> sets the maximal depth of the index tree to <no> (*index tree depth*, see section 11.2)
- `:max-args` <no> sets the maximal breadth of the index tree to <no> (*index tree breadth*, see section 11.2)
- `:debug on (off)`: for internal use only

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
```

[20] BVP cannot be changed by the user

```
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 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[21] which produces the disjunctive normal form of a logic formula with the operators 'and', 'or' and 'not' (here written as a, o, and n).

We begin our example with the RELFUN program of dnf[22] and its head information:

```
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).
```

[21] cf. section 11 and appendix C.2.2

[22] the only difference to standard PROLOG here being the use of square brackets instead of round parentheses for structures

Head information:

#		Arg 1		Arg 2	
1	*dnf*(	X	,	X	)
2	*dnf*(	o/2	,	o/2	)
3	*dnf*(	a/2	,	a/2	)
4	*dnf*(	n/1	,	W	)
5	*dnf*(	n/1	,	W	)
6	*dnf*(	n/1	,	W	)
7	*dnf*(	o/2	,	W	)
8	*dnf*(	a/2	,	a/2	)
9	*dnf*(	a/2	,	a/2	)
10	*dnf*(	a/2	,	W	)
11	*dnf*(	a/2	,	W	)
12	*dnf*(	a/2	,	W	)

Classified clauses (indexing 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)
(arg
 2
 (var x)
 (struct o 2)
 (var w)))
         (seqind
          (arg
           2
           (info 1)
           (const)
           (struct ((o 2)
    (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)) )
```

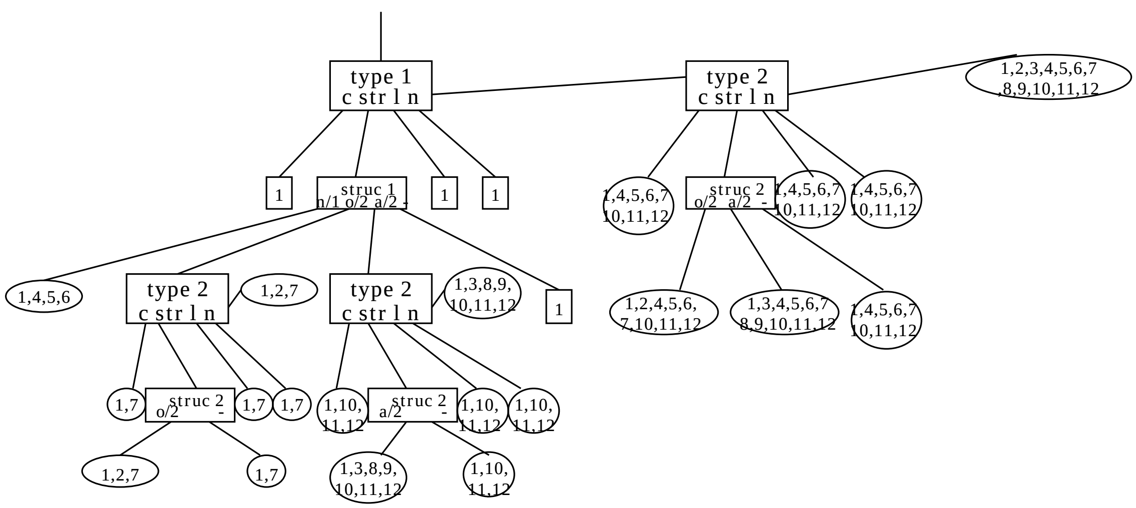
```
      (list)
      (nil)
      (other (clauses 1 4 5 6 7 10 11 12)) ) ) ) ) )
  (fun*den  ; clauses part omitted
   .....
  ))
```

The indexing switches had the following values:

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

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 dnf/2 clauses is of the following form:



The resulting index code[23] 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"
```

[23] an instruction `inst(arg1, ..., argN)` is internally written as `(inst arg1 ...argN)`, i.e. in LISP syntax

```
(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               ; WAM code for clauses omitted
....
2
....
```

C Benchmarks

C.1 Benchmark Results

The next table gives an overview of three benchmarks[24]:

1. the first benchmark is the well known naive reverse benchmark

2. the second benchmark (dnf) is the complete program from section 11 and appendix B

3. the third test is the NET DATALOG benchmark; NET is an automatically generated (from a constraint net) tool-selection program for an NC-program generator [BHH+91]; its task is to select a cutting tool for turning a given workpiece on a CNC-lathe machine

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.

[24]these benchmark results are not very exact, since run-time was taken by hand (our emulator has no run-time measure predicate).

benchmark name	target hardware	time
nrev : well known naive reverse benchmark 6 lines arity of procedures: 2-3		
	SUN 4 125 MB RAM (Lucid) no indexing	13 sec
	SUN 4 125 MB RAM (Lucid) indexing	7 sec
dnf : tool from Hans-Günther Hein (see [Hei93]) 105 lines arity of procedures: 2-3		
	IVORY LISP-BOARD (Symbolics) no indexing	84 sec
	IVORY LISP-BOARD (Symbolics) indexing	24 sec
	SUN 4 125 MB RAM (Lucid) no indexing	425 sec
	SUN 4 125 MB RAM (Lucid) indexing	120 sec
NET : (author: Frank Steinle) 312 lines arity of procedures 2-3		
	IVORY LISP-BOARD (Symbolics) no indexing	288 sec
	IVORY LISP-BOARD (Symbolics) indexing	15 sec
	SUN 4 125 MB RAM (Lucid)) no indexing	1460 sec
	SUN 4 125 MB RAM (Lucid) indexing	72 sec

C.2 Benchmark Sources

These are the listings of the benchmarks used in section C.1.

C.2.1 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.2.2 dnf Benchmark

This benchmark was called with the procedure `go4`. Only the time for finding the first solution was 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).
```

C.2.3 NET Benchmark

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

```
t-isa(X, X).
t-isa(X, Y) :- tt-isa(X, Y).

tt-isa(X, Y) :- isa(X, Y).
tt-isa(X, Y) :- isa(X, Z), tt-isa(Z, Y).

isa(90, rechter).
is-leaf(90).
isa(0, spitz).
is-leaf(0).
isa(10, spitz).
is-leaf(10).
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(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).


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


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 Implementation of the Heuristics

```
; ----------
; selectors:
; ----------

; CG:
; ---

(defmacro s-var-name (term-classification)
  `(cadar ,term-classification))


; ICL:
; ----

; classified procedure:

(defun icl.s-iblock-from-class-proc (classified-procedure)
  (cadr (cadddr classified-procedure)))

; iblock:

(defun icl.s-iblock-type (iblock)
  ; nil, pblock, sblock, 1block
  (car iblock))


; pblock:

(defun icl.s-rblock-from-pblock (pblock)
  (cadr pblock))

(defun icl.s-iblock-list-from-pblock (pblock)
  (cddr pblock)) ; cannot be another pblock or rblock!


; sblock:

(defun icl.s-rblock-from-sblock (sblock)
  (cadr sblock))

(defun icl.s-seqind-arg-list-from-sblock (sblock)
  (cdaddr sblock))

(defun icl.s-iblock-from-sblock (sblock)
  (cadddr sblock))


; 1block:

(defun icl.s-clause-from-1block (1block)
  (cadadr 1block))

(defun icl.s-arg-col-list-from-1block (1block)
  (cddr 1block))


; rblock:

(defun icl.s-clauses-from-rblock (rblock)
  (cdadr rblock))
```

```
(defun icl.s-arg-col-list-from-rblock (rblock)
  (cddr rblock))


; arg-col:

(defun icl.s-arg-no-from-arg-col (arg-col)
  (cadr arg-col))

(defun icl.s-it-list-from-arg-col (arg-col)
  (cddr arg-col))


; seqind-arg:

(defun icl.s-arg-no-from-seqind-arg (seqind-arg)
  (cadr seqind-arg))

(defun icl.s-info-from-seqind-arg (seqind-arg)
  (caddr seqind-arg))

(defun icl.s-constant-list-from-seqind-arg (seqind-arg) ; -> element list
  (cdr (cadddr seqind-arg)))

(defun icl.s-structure-list-from-seqind-arg (seqind-arg) ; -> element list
  (cdar (cddddr seqind-arg)))

(defun icl.s-list-from-seqind-arg (seqind-arg) ; -> 1 element
  (cadr (cddddr seqind-arg)))

(defun icl.s-nil-from-seqind-arg (seqind-arg) ; -> 1 element
  (caddr (cddddr seqind-arg)))

(defun icl.s-other-from-seqind-arg (seqind-arg) ; -> 1 element
  (cadddr (cddddr seqind-arg)))

(defun icl.s-var-from-raw-seqind-arg (seqind-arg) ; -> 1 element
  (cadr (cddddr seqind-arg)))


; element (in constant list, structure list, or list, nil):

(defun icl.s-element-name-from-element (element) ; doesn't make sense
  (car element))                                 ; on list, nil, other

(defun icl.s-clauses-from-element (element)
  (cdadr element))

(defun icl.s-iblock-from-element (element)
  (caddr element))



; ---------------
; mk-index-struct
; ---------------

(defun mk-index-struct (procedure-name clause-count list-of-clauses)
  (cons 'indexing
        (when (>=  clause-count idx.*min-no-of-proc-clauses*)
               (let ((it-heads (mapcar #'icl.mk-it-head list-of-clauses)))
                   (when (car it-heads) ; args exist
                         (let* ((rblock (icl.gen-rblock it-heads))
```

```
                              (iblock (icl.gen-iblock rblock)))
                             (icl.nil-or-list iblock)))))))



; --------------------
; make index type head
; --------------------

(defun icl.mk-it-head (clause)
  (let ((head-chunk (car (s-cg-chunks clause))))
       (icl.mk-it-head2
        (s-cg-arglist_classification
         (s-cg-fac_list (s-cg-chunk_head_literal head-chunk)))
        (icl.get-it-bindings (s-cg-chunk_hd_cgfpl head-chunk)))))

(defun icl.mk-it-head2 (old-head it-bindings)
  (unless (null old-head)
          (cons
           (let ((index-type (icl.g-index-type (car old-head))))
                (cond ((eq (car index-type) 'var)
                       (cond ((cdr (assoc (cadr index-type) it-bindings)))
                             (T index-type)))
                      (T index-type)))
           (icl.mk-it-head2 (cdr old-head) it-bindings))))



; get index type bindings

(defun icl.get-it-bindings (guards*fpl) ; fpl = first premise literal
  (mapcan #'icl.get-it-binding guards*fpl))

(defun icl.get-it-binding (guard)
  ; returns (<it>) or nil
  (when (consp guard) ; ignore constant "first_premise_literals"
        (when (eq (s-cg-functor guard) 'is)
              (let ((arglist (s-cg-arglist_classification guard)))
                   (when (arg-var-p (car arglist))
                         (cons (cons (s-var-name (car arglist))
                                     (icl.g-index-type (cadr arglist)))
                               nil))))))


; generate index types (only basic types: var, const, struct)

(defun icl.g-it-const (term)
  (when (atom term)
        (list 'const term)))

(defun icl.g-it-var (term)
  (when (arg-var-p term)
        (list 'var (s-var-name term))))

(defun icl.g-it-struct (term)
  (when (cg-inst-p term)
        (list 'struct
              (cg-s-inst-functor term)
              (length (cg-s-inst-funargs term)))))

(defun icl.g-index-type (term)
  (cond ((icl.g-it-const term))
        ((icl.g-it-var term))
         ((icl.g-it-struct term))
```

```
        (T (error "icl.g-index-type: unknown type ~A" term))))


; index types type tests ...

(defun icl.it-const-p (it)
  (eq (car it) 'const))

(defun icl.it-var-p (it)
  (eq (car it) 'var))

(defun icl.it-struct-p (it)
  (eq (car it) 'struct))

(defun icl.it-p (it) T) ; needed in 'icl.arg-col-statistics'

(defun icl.it-not-index-p (it) ; change this if additional var-like
  (icl.it-var-p it))           ; types are added !

(defun icl.it-index-p (it)
  (not (icl.it-not-index-p it)))


(defun icl.it-element (it)
  (if (null (cddr it))
      (cadr it)   ; element is an atom
      (cdr it)))  ; element is a list


; type transformations

(defun icl.id (it)
  it)

(defun icl.var-anonym (it) ; anonymize variables: (var x) -> (var _)
  (if (icl.it-var-p it)
      `(var _)
      it))



; --------------------------
; generate rblock (raw block)
; --------------------------

(defun icl.gen-rblock (it-heads)
  (cons 'rblock (cons (cons 'clauses (icl.numbers 1 (length it-heads)))
                      (icl.gen-arg-col-tags
                       (icl.swap-rows-and-cols it-heads)))))

(defun icl.gen-arg-col-tags (arg-cols &optional (no 1))
  (unless (null arg-cols)
          (cons (cons 'arg (cons no (car arg-cols)))
                (icl.gen-arg-col-tags (cdr arg-cols) (1+ no)))))



; ---------------------
; generate rblock*rblock
; ---------------------

(defun icl.gen-rblock*rblock (rblock len)
  (let* ((clauses (icl.s-clauses-from-rblock rblock))
         (clauses*clauses (get-first-n-elements-and-rest len clauses))
```

```
        (arg-cols (icl.s-arg-col-list-from-rblock rblock))
         (arg-nos (mapcar #'icl.s-arg-no-from-arg-col arg-cols))
        (splitted-arg-cols (multiple-splitting
                            len
                            (mapcar #'icl.s-it-list-from-arg-col arg-cols)))
        (arg-cols1 (mapcar #'car splitted-arg-cols))
        (arg-cols2 (mapcar #'cdr splitted-arg-cols))
        (rblock1 (cons 'rblock
                       (cons (cons 'clauses (car clauses*clauses))
                             (icl.add-arg-tags arg-nos arg-cols1))))
        (rblock2 (cons 'rblock
                       (cons (cons 'clauses (cdr clauses*clauses))
                             (icl.add-arg-tags arg-nos arg-cols2)))))
       (cons rblock1 rblock2)))


(defun icl.add-arg-tags (arg-nos arg-cols)
  (mapcar #'(lambda (arg-no arg-col)
                    (cons 'arg (cons arg-no arg-col)))
          arg-nos arg-cols))



; ---------------------------------------
; block analysis: icl.analyze-all-arg-cols
; ---------------------------------------

(defun icl.analyze-arg-col (it-list len max-no-of-vars max-portion-of-vars)
  (let ((pos 1)
        (itl it-list)
        (l nil)
        (max-pos 0)
        (max-list nil)
        (no-of-vars 0))
       (loop
        (when (null itl) (return (cons max-pos max-list)))
        (when (icl.it-not-index-p (car itl))
              (set-inc no-of-vars)
              (when (or (> no-of-vars max-no-of-vars)
                        (> (/ (float no-of-vars) len)
                           max-portion-of-vars))
                    (return (cons max-pos max-list))))
        (let ((var-portion (/ (float no-of-vars) pos)))
             (set-cons var-portion l)
             (when (<= var-portion max-portion-of-vars)
                   (setq max-pos pos
                         max-list l)))
        (set-inc pos)
        (set-cdr+ itl))))



(defun icl.analyze-all-arg-cols (arg-col-list
                                 no-of-clauses
                                 max-no-of-vars
                                 max-portion-of-vars
                                 min-block-portion)

  ; returns: - (1) for 1blocks
  ;          - (len . nil/t-list) for sblocks
  ;            where a t in the nil/t-list stands for a useful argument

  (let ((analyzed-arg-cols
        (mapcar #'(lambda (arg-col)
```

```
                         (icl.analyze-arg-col (cddr arg-col)
                                              no-of-clauses
                                              max-no-of-vars
                                               max-portion-of-vars))
               arg-col-list)))
       (let ((max-len (apply #'max (mapcar #'car analyzed-arg-cols))))
            (cond
             ((< max-len 2) '(1))
             (T (let ((min-len (truncate (* max-len min-block-portion))))
                     (icl.find-last-optimum
                      analyzed-arg-cols
                      (length analyzed-arg-cols)
                      (if (< min-len 2) 2 min-len)
                      max-len
                      max-portion-of-vars)))))))


(defun icl.find-last-optimum (analyzed-arg-cols no-of-arg-cols min-len max-len
                                              max-portion-of-vars)
  (do ((pos max-len (1- pos))
       (arg-cols analyzed-arg-cols)
       (opt-pos max-len)
       (opt-useful-arg-cols nil)
       (optimum 0))

      ((or (< pos min-len)
           (= optimum no-of-arg-cols))
       (cons opt-pos opt-useful-arg-cols))

      (let* ((cars*cdrs (mapcar #'(lambda (arg-col)
                                   (icl.pl-car*cdr arg-col pos 1))
                               arg-cols))
             (useful-arg-cols (mapcar #'(lambda (p)
                                              (<= p max-portion-of-vars))
                                     (mapcar #'car cars*cdrs)))
             (no-of-useful-arg-cols (count-if #'(lambda (x) x)
                                            useful-arg-cols)))
            (setq arg-cols (mapcar #'cdr cars*cdrs))
            (when (> no-of-useful-arg-cols optimum)
                  (setq optimum no-of-useful-arg-cols
                        opt-useful-arg-cols useful-arg-cols
                        opt-pos pos)))))


(defun icl.pl-car*cdr (plist pos &optional default)
  ; car/cdr of partial list (len . list)
  (cond ((> pos (car plist)) (cons default plist))
        ((<= pos 0) (cons nil plist))
        (T (cons (cadr plist)
                 (cons (1- (car plist)) (cddr plist))))))



; ------------------------------------
; generate iblock (indexed block) or nil
; ------------------------------------

(defun icl.gen-iblock (rblock)
  (let ((no-of-clauses (length (icl.s-clauses-from-rblock rblock))))
       (when (> no-of-clauses 1)
             (let ((pblock (icl.gen-pblock rblock no-of-clauses)))
                  (if (null (cdddr pblock))
                      (caddr pblock)   ; simplify pblocks with only 1 partition
                      pblock)))))
```

```
; ------------------------------------------
; heuristics for generating pblock partitions
; ------------------------------------------

(defun icl.max-no-of-vars (no-of-clauses)
  (if (<= no-of-clauses idx.*max-no-of-vars*)
      (1- no-of-clauses)
      idx.*max-no-of-vars*))


(defun icl.max-portion-of-vars (no-of-clauses)
  (if (<= no-of-clauses idx.*max-no-of-vars*)
      0.99
      0.75))

(defun icl.min-block-portion (no-of-clauses)
  0.7)



; -----------------------------------
; generate pblock (partitioned block)
; -----------------------------------

(defun icl.gen-pblock (rblock no-of-clauses) ; -> pblock
  (cons 'pblock (cons rblock
                      (icl.gen-pblock-partitions rblock no-of-clauses))))

(defun icl.gen-pblock-partitions (rblock no-of-clauses)
  (when (> no-of-clauses 0)
        (let ((len*nil/t-list (icl.analyze-all-arg-cols
                               (icl.s-arg-col-list-from-rblock rblock)
                               no-of-clauses
                               (icl.max-no-of-vars no-of-clauses)
                               (icl.max-portion-of-vars no-of-clauses)
                               (icl.min-block-portion no-of-clauses))))
             (let ((rblock*rblock (icl.gen-rblock*rblock
                                   rblock (car len*nil/t-list))))
                  (cons (icl.gen-sblock (car rblock*rblock)
                                        (car len*nil/t-list)
                                        (cdr len*nil/t-list))
                        (icl.gen-pblock-partitions
                         (cdr rblock*rblock)
                         (- no-of-clauses (car len*nil/t-list))))))))

; ---------------
; generate sblock
; ---------------

(defun icl.gen-sblock (rblock len nil/t-list) ; -> sblock

  ; 1a. return 1block

  (cond
   ((= len 1)
    (cons '1block
          (cdr rblock)))


  ; 1b. create and return normal sblock

   (T (let* ((clauses (icl.s-clauses-from-rblock rblock))
```

```
           (arg-col-list (icl.s-arg-col-list-from-rblock rblock)))

  ; 2. select 'constant'/'variable' argument columns

          (let ((constant-arg-cols
                 (mapcan #'(lambda (useful arg-col)
                                  (when useful (list arg-col)))
                         nil/t-list arg-col-list)))

               (let ((variable-arg-cols
                      (mapcan #'(lambda (useful arg-col)
                                       (unless useful (list arg-col)))
                              nil/t-list arg-col-list)))

  ; 3. create seqind structure

                    (let ((seqind-structure
                           (icl.gen-seqind constant-arg-cols
                                           variable-arg-cols
                                           clauses)))

  ; 4. create indexed rest block (from variable-arg-cols)

                         (let ((indexed-rest-block
                                (when (and variable-arg-cols
                                           (> (length clauses) 1))
                                      (cons (icl.gen-iblock
                                             (cons 'rblock
                                                   (cons
                                                    (cons 'clauses clauses)
                                                    variable-arg-cols)))
                                            nil))))
  ; 5. build sblock

                              (cons 'sblock
                                    (cons rblock
                                          (cons
                                           seqind-structure
                                           indexed-rest-block))))))))))


(defun icl.arg-col-statistics (arg-col
                               clauses
                               &optional (predicate #'icl.it-p)
                                         (it-transform #'icl.id))

  ; create an assoc list for an argument column of the form
  ; ((<it> . <clauses>) ...) where <it> is of the form
  ; (const <c>) ...
  ; predicate should be #'icl.it-[not-]index-p ...
  ; it-transform should be #'icl.id or #'icl.var-anonym

  (cond ((null arg-col) nil)
        ((not (funcall predicate (car arg-col)))
         (icl.arg-col-statistics (cdr arg-col) (cdr clauses)
                                 predicate it-transform))
        (T (let* ((rest-args
                   (icl.arg-col-statistics (cdr arg-col) (cdr clauses)
                                           predicate it-transform))
                  (clause (car clauses))
                  (index-arg (funcall it-transform (car arg-col)))
                  (index-arg*clauses (assoc index-arg rest-args
                                            :test #'equal)))
```

```
                (acons index-arg (cons clause (cdr index-arg*clauses))
                       (delete index-arg*clauses
                               rest-args))))))


(defun icl.gen-seqind (tagged-arg-cols additional-arg-cols clauses)

  ; sequential indexing

  (let* ((seqind-args
          (sort (mapcar #'(lambda (t-a-c)
                                  (icl.gen-seqind-arg t-a-c clauses))
                        tagged-arg-cols)
                #'(lambda (a b)
                          ; change this for better heuristics!!
                          (> (car (cdaddr a)) (car (cdaddr b))))))
         (sorted-tagged-arg-cols
          (icl.sort-tagged-arg-cols
           tagged-arg-cols
           (mapcar #'cadr seqind-args))))
        (cons 'seqind
              (maplist #'(lambda (rest-seqinds rest-t-a-c)
                                 (icl.extend-seqind clauses
                                                    (car rest-seqinds)
                                                    (append
                                                     (cdr rest-t-a-c)
                                                     additional-arg-cols)))
                       seqind-args
                       sorted-tagged-arg-cols))))


(defun icl.sort-tagged-arg-cols (tagged-arg-cols numbers)
  ; sort tagged-arg-cols the same way the numbers are sorted
  (mapcar #'(lambda (n)
                    (find-if #'(lambda (t-a-c)
                                       (= (cadr t-a-c) n))
                             tagged-arg-cols))
          numbers))


(defun icl.gen-seqind-arg (tagged-arg-col clauses)
  (let ((type-table (icl.type-collect
                     (icl.arg-col-statistics
                      (cddr tagged-arg-col)
                      clauses
                      #'icl.it-p
                      #'icl.var-anonym))))
  (cons 'arg
        (cons (cadr tagged-arg-col)
              (cons (list 'info (icl.compute-weight-of-const-arg-col
                                 type-table))
                    type-table)))))


(defun icl.compute-weight-of-const-arg-col (type-table)
  ; simply count number of different constants/structures
  (+ (length (cdar type-table))
     (length (cdadr type-table))))


(defun icl.type-collect (stat-table)
  ; only for constants, structures and vars;
  ; returns const*struct*var
  ; subtypes handled by icl.extend-seqind
```

```
  (let ((constants nil)
        (structures nil)
        (vars nil))
       (dolist (it*clauses stat-table)
               (let* ((it (car it*clauses))
                      (element (icl.it-element it))
                      (clauses (cdr it*clauses))
                      (tagged-clauses (cons 'clauses clauses))
                      (element**tagged-clauses (list element tagged-clauses)))
                     (cond ((icl.it-const-p it)
                            (set-cons element**tagged-clauses constants))
                            ((icl.it-struct-p it)
                            (set-cons element**tagged-clauses structures))
                           ((icl.it-var-p it)
                            (setq vars (cons tagged-clauses nil)))
                           (T (error "icl.type-collect: unknown type: ~A"
                                     it)))))
       (list (cons 'const (nreverse constants))
             (cons 'struct (nreverse structures))
             (cons 'var vars))))



(defun icl.gen-constants*nil (constants)
  (let ((empty-list
         (find-if #'(lambda (constant)
                           (null (icl.s-element-name-from-element constant)))
                  constants)))
       (cons (delete empty-list constants :test #'equal)
             (cdr empty-list))))


(defun icl.gen-structures*list (structures)
  (let ((list
         (find-if #'(lambda (structure)
                           (equal (icl.s-element-name-from-element structure)
                                  '(cns 2)))
                  structures)))
       (cons (delete list structures :test #'equal)
             (cdr list))))


(defun icl.extend-seqind (org-clauses seqind rest-tagged-arg-cols)
  ; add new iblocks for multiply orruring elements
  ; and split constants and structures for subtypes (nil, list)
  (let* ((arg-no (icl.s-arg-no-from-seqind-arg seqind))
         (info (icl.s-info-from-seqind-arg seqind))
         (constants (icl.s-constant-list-from-seqind-arg seqind))
         (structures (icl.s-structure-list-from-seqind-arg seqind))
         (vars (icl.s-var-from-raw-seqind-arg seqind))
         (var-clauses (icl.s-clauses-from-element vars))
         (ext-constants (icl.extend-seqind-elements
                         constants
                         rest-tagged-arg-cols
                         org-clauses
                         var-clauses))
         (ext-structures (icl.extend-seqind-elements
                          structures
                          rest-tagged-arg-cols
                          org-clauses
                          var-clauses))
         (constants*nil (icl.gen-constants*nil ext-constants))
         (structures*list (icl.gen-structures*list ext-structures)))
        (cons
```

```
         'arg
         (cons
          arg-no
          (cons
           info
           (cons
            (cons 'const (car constants*nil))
            (cons
             (cons 'struct (car structures*list))
             (cons (cons 'list (cdr structures*list))
                   (cons (cons 'nil (cdr constants*nil))
                         (when var-clauses
                               (cons
                                 (cons 'other
                                       (cdr (icl.extend-seqind-element
                                             vars
                                             rest-tagged-arg-cols
                                             org-clauses
                                             nil)))
                                nil)))))))))))


(defun icl.extend-seqind-elements (elements rest-t-a-c org-clauses var-clauses)
  (mapcar #'(lambda (element)
                    (icl.extend-seqind-element
                     element rest-t-a-c org-clauses var-clauses))
          elements))


(defun icl.extend-seqind-element (element rest-t-a-c org-clauses var-clauses)
  (let ((clauses (sort (append (icl.s-clauses-from-element element)
                               (copy-list var-clauses)) ; sort is destructive!
                       #'<)))
       (cons (icl.s-element-name-from-element element)
             (cons (cons 'clauses clauses)
                   (when rest-t-a-c
                         (icl.nil-or-list
                          (icl.gen-iblock
                           (icl.gen-rblock-for-seqind
                            org-clauses clauses rest-t-a-c)))))))))


(defun icl.gen-rblock-for-seqind (org-clauses clauses tagged-arg-cols)
  (cons 'rblock
        (cons
         (cons 'clauses clauses)
         (mapcar #'(lambda (tagged-arg-col)
                           (cons
                            'arg
                            (cons
                             (cadr tagged-arg-col)
                             (mapcan #'(lambda (it clause)
                                                (when (member clause clauses)
                                                      (cons it nil)))
                                     (cddr tagged-arg-col)
                                     org-clauses))))
                 tagged-arg-cols))))


; -------------------
; auxiliary functions
; -------------------
```

```
(defun icl.swap-rows-and-cols (lists)
  (apply #'mapcar (cons #'list lists)))

(defun icl.numbers (start end)
  (unless (> start end)
          (cons start (icl.numbers (1+ start) end))))

(defun icl.nil-or-list (l)
  (when l (cons l nil)))
```

References

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

[BBB+89] 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. May 1989.

[BEHK91] 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.

[BHH+91] Harold Boley, Philipp Hanschke, Martin Harm, Knut Hinkelmann, Thomas Labisch, Manfred Meyer, Jörg Müller, Thomas Oltzen, Michael Sintek, Werner Stein, and Frank Steinle. *μCAD2NC: A Declarative Lathe-Workplanning Model Transforming CAD-like Geometries into Abstract NC Programs.* DFKI Document D-91-15, DFKI GmbH, November 1991.

[Bol90] Harold Boley. A Relational/Functional Language and Its Compilation into the WAM. SEKI Report SR-90-05, Universität Kaiserslautern, Fachbereich Informatik, April 1990.

[Cla85] J. Clancey, W. Heuristic Classification. *Artificial Intelligence*, 27:289–350, 1985.

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

[Hei89] 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.

[Hei93] Hans-Günter Hein. Propagation Techniques in WAM-based Architectures — The FIDO-III Approach. DFKI Technical Memo TM-93-04, DFKI GmbH, October 1993.

[HM89] Timothy Hickey and Shyam Mudambi. Global Compilation of Prolog. *Journal of Logic Programming*, 7:193–230, 1989.

[Kra90] 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.

[Kra91] Thomas Krause. Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN. Diplomarbeit, DFKI D-91-08, Universität Kaiserslautern, FB Informatik, Postfach 3049, D-6750 Kaiserslautern, März 1991.

[Nys85] Sven Olof Nystrøm. NyWam - A WAM Emulator Written in LISP. 1985.

[SS92] 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.

[Ste92] Werner Stein. Indexing Principles for Relational Languages Applied to PROLOG Code Generation. Technical Report Document D-92-22, DFKI GmbH, 1992.

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