A Complete Indexing Scheme for WAM-based Abstract Machines

Werner Hans*

RWTH Aachen, Lehrstuhl für Informatik II
Ahornstraße 55, W-5100 Aachen, Germany

Abstract. This paper introduces an indexing scheme for logic programming languages, complete in considering all parameters and nested terms. Nevertheless it does not lead to a code-size explosion, because the sets of valid clauses are not encoded by special control flow instructions. Instead, some kind of identifiers of the applicable clauses are stored in (clause) sets allowing the stepwise restriction of alternative clauses during the runtime analysis of the parameters.
We present the compilation scheme to integrate this approach in existing compilers. It shows the cooperation of the new instructions, the strict separation of the indexing code from the code executing the compiled clauses, and it demonstrates the easiness of integrating a complete indexer in compilers.
The compilation scheme was originally developed for the stack-based narrowing machine of the functional logic language BABEL. We describe, however, the slightly modified indexing scheme for the usual WAM-architecture and assume Prolog as the language platform.

1 Introduction

Logic programming languages, such as Prolog, define predicates by several clauses describing the different conditions which must be fulfilled to apply the predicate. While evaluating the query (or parts of it) the described conditions are tested to determine the subset of applicable clauses. One request is their quick selection, because much time may be spent in this determination phase.

Indexing is an approach that

- improves the determination of the first applicable rule,
- separates the applicable rules from the non–applicable ones. This leads to an early reduction of the search space, being important for (or-)parallel implementations,
- offers the possibility of the detection of determinism, leading to less memory consumption, in particular of the runtime stack, because less choice point must be saved. [Hickey,Mudambi89] point out that indexing can affect the consumption of all memory areas.
- allows the fast detection of failure situations.
- enables the lazy narrowing machine (in a functional + logic environment) to select the rules by viewing only the currently evaluated parameters. This mechanism delays (possibly infinite) computations which may be irrelevant for the decision.

[Kliger,Shapiro88] construct a decision tree at compile time taking all clauses of a predicate into consideration. They translate this tree to a header for the corresponding machine code of the predicate. At runtime the generated instructions control the flow which finally reaches the jump instruction pointing at the correct clause[2]. This is a fast and complete approach, but it consumes much program space. The decision tree may need program space exponential in the number of clauses and argument positions. Consequently they choose in [Kliger,Shapiro90] decision graphs rather than decision trees to encode the possible traces of each predicate. Although this eliminates many cases of exponential blow-up of codesize, it cannot prevent it. [Korsloot,Tick91] extend this algorithm to handle "don't know" nondeterminism. Now the control flow reaches try/trust blocks. Each of them consists of instructions, to jump to the first applicable clause as well as to choose available alternatives after failure(s). As the decision tree/graph may possess many branches at every level, this may result in many try/trust blocks at its leaves.

[Hickey,Mudambi89] use a similar approach, to determine the applicable clauses. For further improvements at runtime, they consider mode–information and encode each possible invocation (situation) by specialized machine code. But this leads to further code space consumption.

All these approaches need quite complex compilers. Some authors omit complete indexing and use instead simpler kinds of it. A well-known restriction is the first–argument–top–level–indexing[Aït-Kaci91]. This restriction considers that most of the programmers have in mind the depth–first–left–to–right evaluation order of Prolog (in particular during the unification) while they write programs. This specialized clauses support this simple indexing mechanism, frequently.

But looking at the recently developed functional + logic programming languages it becomes more attractive to consider complete indexing. An alternative approach is presented in [Chakravarty,Lock91]. The authors do not statically create a choice point, whenever a predicate with more than one clause is called. Instead, they dynamically create a choice point, only in cases, an unbound variable may get alternative bindings. To ensure the correctness of their approach, the programs must meet a nonambiguity requirement, that is usual in functional programming.

To decrease the size of the decision tree, they do a program transformation including clause– and parameter–reordering. For this transformed program they compute the indexing code similar to the pattern matcher in [Peyton,Jones87]. But their approach is illegal for programs being very sensible to clause– or parameter–reordering, as Prolog, BABEL.

The reason for the exponential blow-up is given by the kind of chains connecting the clauses. That chaining is done via special control flow instructions like try, retry, and trust, grouped to try/trust blocks. This is simple and clearly fast, but the compiled code may explode encoding such inflexible (unconditional) try/trust blocks. Additionally, the intermediate analysis results of several arguments are not stored,

* The author is supported by the grant In 20/6-1 from the 'Deutsche Forschungsgemeinschaft'. e-mail: hans@zeus.informatik.rwth-aachen.de

[2] They present their ideas for FCP (Flat Concurrent Prolog).

but immediately used for branching. Therefore they are only implicit given by the paths from the root to the actual node in the decision tree. This must lead to space consuming decision trees.

Another interesting approach is taken in [Palmer,Naish91], who have proved that the generation of the decision tree is an NP-hard problem. They use so-called clause sets to store the applicable rules, which are computed dynamically. We choose their approach, too, because of its elegance and simplicity, but we focus on the compilation scheme.

This paper is organized as follows: The next section mentions the chosen indexing approach. It describes the extensions of the WAM and points out the new instructions. Section 3 deals with the compilation scheme and Section 4 gives a compilation example, demonstrating the simplicity of our scheme. In Section 5 the produced code is analyzed and some simple and impressive optimizations are presented. Section 6 mentions some restrictions that occur in real implementations and Section 7 concludes this paper presenting several runtime results and memory demands. In most of the sections we take Prolog as the source language and assume Prolog's evaluation strategy.

2 Indexing

Complete indexing divides the invocation of a predicate in a preceding analysis pass and the execution pass. During the analysis pass, the clauses are split into applicable and non–applicable ones. The applicable ones are stored in clause sets. Starting with the clause set containing the "identifiers" of all clauses, the non–applicable clauses are successively removed from it. This approach allows an independent analysis of each parameter as well as an easy examination of the components of nested terms. In terms of decision trees, this approach split the great decision tree in several less complex decision trees.

All (separate) restrictions are synthesized by the intersection of the superset[3] and the computed sets. The computed sets only consider restrictions got by a local analysis of each parameter, but the intersection of all sets delivers the desired set.

By applying Prolog's left-to-right evaluation strategy to the analysis, we only need one set for the set manipulation. This claims the new register *Alt* in the abstract machine, to hold the intermediate set.

The code of each procedure[4] is preceded by a special code for the analysis. It compares the current arguments with the formal ones and collects the different restrictions in Alt. This indexing part is eventually concluded by an instruction, which takes the evaluated clause set (*Alt*) and jumps to the first valid clause.

To profit from the analysis after failure, the choice point is extended with a new component *alt* storing the remaining clauses of the procedure. This component gets a "copy" of Alt when branching to the first clause. Its value is modified during backtracking. To capture this situation the indexing part is augmented by one further instruction, to jump to when a failure occurs. This instruction combines the actions of the usual WAM-instructions retry_me_else and trust_me.

[3] containing all defined clauses

[4] all clauses of a program with the same predicate symbol and the same arity.

In the following listing an informal description of each instruction is given:

switch_on_term X_i,(vlab,valt),(clab,calt),(flab,falt)[5]
dereferences[6] the content of the i-th argument register, inspects the dereferenced value, and jumps to the instruction labeled vlab, clab or flab, depending on the values type, i.e. whether it is a variable (unbound REF-cell), a constant- or a functor-cell, respectively. Additionally the current value in register Alt is intersected with one of the three sets valt, calt or falt with respect to the inspected data. The value of register X_i is replaced by the dereferenced value.

switch_on_constant X_i,(lab,alt),$\mathcal{A}$, $\mathcal{A} = \{(c_i, alt_i) \mid 1 \leq i \leq n\}$
expects a constant c in register X_i. If $\mathcal{A}$ contains a pair (c,alt′), Alt is intersected with alt′. Otherwise Alt is intersected with alt. In both cases the control continues at the instruction labeled lab.

The constants in the first position of all tuples in $\mathcal{A}$ are different in pairs.

switch_on_structure X_i,(lab,alt),$\mathcal{A}$ $\mathcal{A} = \{(f_i/n_i, lab_i, alt_i) \mid 1 \leq i \leq n\}$
compares the functor f/n pointed to by register X_i with the functors contained in $\mathcal{A}$. If $\mathcal{A}$ contains a triple (f/n,lab′,alt′) the *subterm* register S is set to the heap address following that functor cell. The mode is set to *read* mode and Alt and alt′ are intersected. The instruction labeled lab′ is executed next.

If $\mathcal{A}$ does not contain a valid triple, Alt is intersected with alt, and a jump to lab occurs.

The functor/arity–pairs in the first position of all triples in $\mathcal{A}$ are different in pairs[7].

try #arg
analyzes the current value of register Alt. If this register stores more than one alternative, the instruction creates a choice point similar to the WAM try_me_else instruction. The new component alt of the choice point is initialized with the current value of Alt. To determine the continuation point it accesses the label-list of the retry instruction, immediately following. If a is Alt's least element it branches to lab_a (see retry), and deletes this element a from alt. The continuation pointer of the choice point is initialized with the label of the following retry instruction.

Try needs its parameter #arg for the size alignment of the choice point.

If register Alt contains only one alternative no choice point must be created. Only the jump is performed.

If register Alt does not contain any alternative, a failure situation occurs.

retry (lab_1,...,lab_n)
is executed during backtracking to determine the next alternative. It is an amalgamation of the WAM-instructions retry_me_else and trust_me. If the component alt of the topmost choice point contains only one element "a", it behaves like trust_me and erases the choice point from the stack. It jumps further to lab_a.

[5] For simplicity we neglect the special consideration of lists.

[6] Dereferencing is done by following a possible reference chain until an unbound REF cell or a non-REF cell is reached.

[7] Conceptually it is possible to handle constants and functors in the same manner, but this division simplifies the compilation scheme given below.

Otherwise it deletes alt's least element "a" from alt and jumps to lab_a. But in both cases the computation environment must be restored. This includes the resetting of the heap, the unwinding of the trail, and the reloading of the argument registers.

In the indexing part the WAM instruction

unify_variable X_i,

is mentioned (which is executed only in read mode) as well as the unconditional *jump*.

To ensure the correct computation of the clause set, the correct initialization of register Alt is necessary. This is done by the instructions *call* and *execute*. They set Alt to $\mathbb{N}$, the set of all natural numbers[8].

3 The compilation scheme

The generation of the indexing part for each procedure consists of two parts. First of all, the necessary information for the static sets of the switch instructions has to be collected. Later, the compiler must be able to extract the currently interesting parts of the collection.

We start with the explanation of the auxiliaries (table 1).

$$\mathit{collect}\ (\ <\mathrm{p}(\mathrm{t}_{i,1},\ldots,\mathrm{t}_{i,n})\text{:--}\mathrm{body}_i \mid 1 \leq i \leq r >) := \bigcup_{1\leq i\leq r, 1\leq j\leq n} \{(\mathrm{i,j,t}_{i,j})\}$$

$$\mathit{split}(\mathcal{A}, \mathrm{j}) := \{(\mathrm{i,t}) | (\mathrm{i,j,t}) \in \mathcal{A}\}$$

$$\mathit{select}(\mathcal{A}, \mathrm{j}) := \{\ (\mathrm{i,t}_j) |\ (\mathrm{i,f}(\mathrm{t}_1,\ldots,\mathrm{t}_j,\ldots,\mathrm{t}_n)) \in \mathcal{A}\}$$

$$\mathit{typesplit}\ (\emptyset) := \emptyset$$

$$\mathit{typesplit}\ ((\mathrm{i,t})+\mathcal{A}) := \begin{cases} (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3 \cup \{(i,t)\}) & ;\text{if } \mathrm{t} \in \mathrm{Var} \\ (\mathcal{A}_1 \cup \{(i,t)\}, \mathcal{A}_2, \mathcal{A}_3) & ;\text{if } \mathrm{t} = \mathrm{c} \\ (\mathcal{A}_1, \mathcal{A}_2 \cup \{(i,t)\}, \mathcal{A}_3) & ;\text{if } \mathrm{t} = \mathrm{f}(\ \mathrm{t}_1, \ldots, \mathrm{t}_n) \end{cases}$$

with $(\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)=\mathit{typesplit}(\mathcal{A})$.

$$\mathit{transform}(\emptyset) := \emptyset$$

$$\mathit{transform}((\mathrm{i,f}(\mathrm{t}_1,\ldots,\mathrm{t}_n))+\mathcal{A}) :=$$

$$\begin{cases} (\mathcal{B} \setminus \{\mathcal{C}\} \cup \{\mathcal{C} \cup \{(\mathrm{i}, \mathrm{f}(\mathrm{t}_1, \ldots, \mathrm{t}_n))\}\} & ;\text{if } \exists \mathcal{C} \in \mathcal{B}, \mathrm{i}', \mathrm{t}'_\mathrm{j} : (\mathrm{i}', \mathrm{f}(\mathrm{t}'_1, \ldots, \mathrm{t}'_\mathrm{n})) \in \mathcal{C} \\ \mathcal{B} \cup \{\{(\mathrm{i}, \mathrm{f}(\mathrm{t}_1, \ldots, \mathrm{t}_\mathrm{n}))\}\} & ;\text{otherwise} \end{cases}$$

with $\mathcal{B}=\mathit{transform}(\mathcal{A})$.

$$\mathit{createalt}(\mathcal{A}) := \{\mathrm{i} \mid (\mathrm{i,t}) \in \mathcal{A}\}$$

Table 1. The auxiliaries

collect collects all necessary information for the generation of the indexing part. It extracts the information from the heads of the program clauses, grouped to procedures. The body of the clause is not viewed. At this point we remind that the origin of our scheme lies in BABEL. As BABEL possesses an explicit if_then_else, neither the programmer must simulate this operation by two or more clauses[9], nor the compiler has to detect and group them. *collect* is called for each procedure. To simplify later manipulation this information is represented as a set of triples. Each triple (i,j,t) contains the number of the clause i, the argument position j, and the term t (formal parameter). The following *append* predicate

append([],X_1,X_1).

append([X_1 |X_2],X_3,[X_1 |X_4]):–append(X_2,X_3,X_4).

leads to {(1,1,[]),(1,2,X_1),(1,3,X_1),(2,1,[X_1 |X_2]),(2,2,X_3),(2,3,[X_1 |X_4])}.

The remainder of the indexing code generation uses only the delivered information, allowing the clause overlapping consideration of the program.

Called with the set computed by *collect*, *split* extracts all entries of it, which describe the terms at the desired argument position in the clauses. As the argument position is always known in the context, the argument position in the entries of the set is omitted. *Select* computes a set of tuples, where in each entry of the input set the nested term is replaced by its j^{th} component. The context guarantees that all terms of the input set are nested structures.

Generating the indexing code for each argument position separately, the compiler calls *typesplit* to divide the given set[10] (of tuples) in accordance with the type of the term. An application results in three sets, one set contains all tuples with constants, the second one contains all structures, and the last one all tuples with variables.

Transform provides the information that is added as parameters to the instructions *switch_on_constant* and *switch_on_structure*. It modifies the given set being either a set of constants or a set of structures. This results in another set, each entry being a set itself and these sets contain all entries with the same constant or the same toplevel functor respective to the original set. We identify constants and structures of arity 0 in the formal description.

The function *createalt* creates the set of all possible alternatives by neglecting the terms of its input set.

Of these preparations we describe the compilation scheme (table 2). We simplify the management of labels by assuming a symbolic address space. We also assume that the i-th argument of a predicate is stored in the i-th register of the machine, denoted by X_i.

Proctrans compiles each procedure. If the procedure consists of only one clause the function *Ruletrans* is called immediately. The other case results in two parts of code: The indexing part containing mainly the switch-instructions is produced by several calls of *Switchtrans*. The other part containing the usual code for the clauses is built up by *Ruletrans*. It produces the machine code for each clause, that shall conclude with a *proceed* instruction or in the case of tail recursion with an *execute* instruction. This code contains no instruction for switch handling after failure. Because we focus on indexing no detailed description of *Ruletrans* is given.

The instructions *try* and *retry* conclude the indexing part. They direct the control flow to the first of all applicable clauses or, after failure, to the next valid one.

Each call of *Switchtrans* produces indexing code for one argument position. The behaviour of the code is comparable with the first two levels of Prolog's standard

[8] Clearly, it suffices to initialize Alt with the numbers of all rules or more restricted with the numbers of all the rules, which belongs to the called predicate.

[9] in pure Prolog

[10] this is the output of *split* (or *select*).

$\mathit{Proctrans}(<\mathrm{p}(\mathrm{t}_{i,1},\ldots,\mathrm{t}_{i,n})\text{:-}\mathrm{body}_i \mid 1 \leq i \leq r>, \mathrm{l}) :=$

p/n: $\mathit{Ruletrans}(\mathrm{p}(\mathrm{t}_{1,1},\ldots,\mathrm{t}_{1,n})\text{:-}\mathrm{body}_1,\mathrm{l})$ if r=1

$\mathit{Switchtrans}(1,\mathit{split}(\mathcal{A},1),\mathrm{l.0.1})$ if r>1

$\vdots$

$\mathit{Switchtrans}(\mathrm{n},\mathit{split}(\mathcal{A},\mathrm{n}),\mathrm{l.0.n})$

try n

retry (l.1,…,l.r)

l.1: $\mathit{Ruletrans}(\mathrm{p}(\mathrm{t}_{1,1},\ldots,\mathrm{t}_{1,n})\text{:-}\mathrm{body}_1,\mathrm{l.1})$

$\vdots$

l.r: $\mathit{Ruletrans}(\mathrm{p}(\mathrm{t}_{r,1},\ldots,\mathrm{t}_{r,n})\text{:-}\mathrm{body}_r,\mathrm{l.r})$

with $\mathcal{A}=\mathit{collect}(<\mathrm{p}(\mathrm{t}_{i,1},\ldots,\mathrm{t}_{i,n})\text{:-}\mathrm{body}_i \mid 1 \leq i \leq r>)$

$\mathit{Switchtrans}(\mathrm{i},\mathcal{A},\mathrm{l}) :=$

switch_on_term $\mathrm{X}_i,(\mathrm{l.0},\mathit{createalt}(\mathcal{A})),(\mathrm{l.1},\mathit{createalt}(\mathcal{A}' \cup \mathcal{A}''')),$
$(\mathrm{l.2},\mathit{createalt}(\mathcal{A}'' \cup \mathcal{A}'''))$

l.1: *switch_on_constant* $\mathrm{X}_i,(\mathrm{l.0},\mathit{createalt}(\mathcal{A}''')),$
$\{(\mathrm{c}_j,\mathit{createalt}(\mathcal{A}'_j \cup \mathcal{A}'''))\mid 1 \leq j \leq k\}$

l.2: *switch_on_structure* $\mathrm{X}_i,(\mathrm{l.0},\mathit{createalt}(\mathcal{A}''')),$
$\{(\mathrm{f}_j/\mathrm{m}_j,\mathrm{l.2.j},\mathit{createalt}(\mathcal{A}''_j \cup \mathcal{A}'''))\mid 1 \leq j \leq n\}$

l.2.1: $\mathit{SwitchStructtrans}(\mathcal{A}''_1,\mathcal{A}''',\mathrm{m}_1,\mathrm{l.2.1},\mathrm{l.0})$

$\vdots$

l.2.n: $\mathit{SwitchStructtrans}(\mathcal{A}''_n,\mathcal{A}''',\mathrm{m}_n,\mathrm{l.2.n},\mathrm{l.0})$

l.0:

with $(\mathcal{A}',\mathcal{A}'',\mathcal{A}''')=\mathit{typesplit}(\mathcal{A})$
$\{\mathcal{A}'_1,\ldots,\mathcal{A}'_k\}=\mathit{transform}(\mathcal{A}')$, $(\mathrm{i}'_j,\mathrm{c}_j)\in \mathcal{A}'_j(1 \leq j \leq k)$,
$\{\mathcal{A}''_1,\ldots,\mathcal{A}''_n\}=\mathit{transform}(\mathcal{A}'')$, and $(\mathrm{i}''_j,\mathrm{f}_j(\mathrm{t}_{j,1},\ldots,\mathrm{t}_{j,m_j}))\in \mathcal{A}''_j(1 \leq j \leq n)$.

$\mathit{SwitchStructtrans}(\mathcal{A},\mathcal{A}',\mathrm{n},\mathrm{l},\mathrm{lab}):=$

unify_variable $\mathrm{X}_{\mathrm{new}_1}$

$\vdots$

unify_variable $\mathrm{X}_{\mathrm{new}_n}$

$\mathit{Switchtrans}(\mathrm{new}_1,\mathit{select}(\mathcal{A},1)\cup\mathcal{A}',\mathrm{l.1})$

$\vdots$

$\mathit{Switchtrans}(\mathrm{new}_n,\mathit{select}(\mathcal{A},\mathrm{n})\cup\mathcal{A}',\mathrm{l.n})$

jump lab

Table 2. The compilation scheme

indexing approach given in[Aït-Kaci91]. The first instruction *switch_on_term* realizes a conditional branch depending on the data in register X_i. The second instruction is executed if a constant is found. Its third parameter stores all given constant symbols together with the alternatives. *Switch_on_structure* will be enriched similar. Further code is generated by *SwitchStructtrans* for each valid functor.

SwitchStructtrans creates code to analyze the components of structures. Its parameters denote

– the set containing the tuples with identical functors of the same arity,
– the set of tuples with variables at the currently viewed argument position,
– the arity of the functor
– a symbol for the label generation, and
– the goal to create a jump to at the end of this code block.

Because the components of the structure must be saved[11], the components are loaded in free registers, indexed from new_1 till new_n, temporarily. Afterwards it calls *Switchtrans* recursively for each component. The code is eventually concluded by the *jump* instruction, that skips the code of other structures that may follow.

Note that the variables, visited so far, are passed to *SwitchStructtrans*. By this, recursive calls of *Switchtrans* will not forget the corresponding clause candidates.

4 Example

To get an impression of the code produced by this scheme, we present the code of the append predicate. At the first glance the presented code looks deterrent. But the optimizations, we are dealing with in the next section, show that many of the produced instructions are not needed anyway. The superfluous instructions are marked by the indexed plus signs in the first column. The different indices correspond to different degrees of optimizations.

	append:	*switch_on_term*	X_1,(l.13,{1,2}),(l.2,{1}),(l.3,{2})	1^{st} arg.
$+_4$	l.2 :	*switch_on_constant*	X_1,(l.13,{}),{([],{1})}	
$+_4$	l.3 :	*switch_on_structure*	X_1,(l.13,{}),{(./2,l.4,{2})}	
$+_3$	l.4 :	*unify_variable*	X_4	$[\mathrm{X}_1 \mid \mathrm{X}_2]$
$+_3$	l.5 :	*unify_variable*	X_5	
$+_2$	l.6 :	*switch_on_term*	X_4,(l.9,{2}),(l.7,{2}),(l.8,{2})	X_1
$+_1$	l.7 :	*switch_on_constant*	X_4,(l.9,{2}),{}	
$+_1$	l.8 :	*switch_on_structure*	X_4,(l.9,{2}),{}	
$+_2$	l.9 :	*switch_on_term*	X_5,(l.12,{2}),(l.10,{2}),(l.11,{2})	X_2
$+_1$	l.10 :	*switch_on_constant*	X_4,(l.12,{2}),{}	
$+_1$	l.11 :	*switch_on_structure*	X_4,(l.12,{2}),{}	
$+_3$	l.12 :	*jump*	l.13	
$+_2$	l.13 :	*switch_on_term*	X_2,(l.16,{1,2}),(l.14,{1,2}),(l.15,{1,2})	2^{nd} arg.
$+_1$	l.14 :	*switch_on_constant*	X_2,(l.16,{1,2}),{}	
$+_1$	l.15 :	*switch_on_structure*	X_2,(l.16,{1,2}),{}	
	l.16 :	*switch_on_term*	X_3,(l.28,{1,2}),(l.17,{1}),(l.18,{1,2})	3^{rd} arg.
$+_1$	l.17 :	*switch_on_constant*	X_3,(l.28,{1}),{}	
$+_4$	l.18 :	*switch_on_structure*	X_3,(l.28,{1}),{(./2,l.19,{1,2})}	
$+_3$	l.19 :	*unify_variable*	X_4	$[\mathrm{X}_1 \mid \mathrm{X}_4]$
$+_3$	l.20 :	*unify_variable*	X_5	
$+_2$	l.21 :	*switch_on_term*	X_4,(l.24,{1,2}),(l.22,{1,2}),(l.23,{1,2})	X_1
$+_1$	l.22 :	*switch_on_constant*	X_4,(l.24,{1,2}),{}	

[11] the subterm register may be changed at runtime by other switch_on_structure commands.

$+_1$	l.23:	*switch_on_structure*	X_4,(l.24,{1,2}),{}	
$+_2$	l.24:	*switch_on_term*	X_5,(l.27,{1,2}),(l.25,{1,2}),(l.26,{1,2})	X_4
$+_1$	l.25:	*switch_on_constant*	X_4,(l.27,{1,2}),{}	
$+_1$	l.26:	*switch_on_structure*	X_4,(l.27,{1,2}),{}	
$+_3$	l.27:	*jump*	l.28	
	l.28:	*try*	3	
	l.29:	*retry*	(l.x_1,l.x_2)	
	l.x_1:	*Ruletrans*(append([],X_1,X_1):–),...)		
	l.x_2:	*Ruletrans*(append([X_1|X_2],X_3,[X_1|X_4]):–append(X_2,X_3,X_4)),...)		

The symbolic labels are given in a more readable form and the last column of the table denotes the analyzed arguments or subterms of them.

5 Simple Optimization

When we consider the last example, we recognize that the code for the indexing analysis can grow very fast with the complexity of the formal parameters of the predicate (but within linear limits). This is undesirable, if the engine has to decide only between few alternatives. Such situations claim for optimizations to handle them in a better manner.

An observation shows that all switch_on_constant/structures with an empty third parameter are superfluous[12].($+_1$)

A second source for optimization is given if all formal parameters at a particular argument position are variables. Obviously all analysis is useless, because no restrictions will be possible. Such a situation can be treated by *Switchtrans* with a test $\mathcal{A}' = \mathcal{A}'' = \emptyset$. The same test removes code for testing variables in common subexpressions ($+_2$), too.

If we look at the indexing part of [X_1|X_2], we can remove the three remaining lines l.4, l.5, and l.12. *SwitchStructtrans* generates this instructions. We can optimize this part of code generation by integrating a check, testing whether indexing code will be produced by the *Switchtrans* invocations. If no code is produced, the unify_variable and jump instructions can be suppressed, too. But we must augment *Switchtrans*: If *SwitchStructtrans* called for a functor f_j, produces no code, the label l.2.j has to be replaced by l.0 in the switch_on_structure instruction ($+_3$).

Adding some heuristics to the optimization, we can possibly do further reductions in code size. If we assume the completeness of the definitions of the predicates in a program, we can remove switch_on_constant and switch_on_structure[13] instructions, if their third parameter contains only one entry. If the assumption does not hold, the rejection of the wrong clauses is done in the usual manner by backtracking. Applied to the second example the lines marked by $+_4$ can be discarded and the indexing code shrinks from 29 to 4 lines.

All the optimizations can be integrated in the compiler in an easy manner.

[12] The deletion of instructions may require the correction of some labels.

[13] possibly together with the code produced by *SwitchStructtrans*.

6 Implementation

This section deals with the implementation aspects of our approach. Complete indexing had demanded extensions in both the compiler and the WAM. One important problem bases on the representation of the introduced sets, which are not sufficiently supported in usual programming environments, such as C. If we neglect the compilation time of programs, as we do, sets can simply be realized as linear lists in the compiler. The translation of the compilation scheme in a concrete programming language is straight forward and does not lead to any problems.

But we cannot neglect time in the WAM, because the indexing overhead competes with the original rejection mechanism. Therefore the sets are realized by fixed sized bitmaps, which are supported very well by nearly every hardware processor. This allows an easy and fast access and manipulation at runtime on the one hand, but this prohibits general sets on the other hand. The current implementation uses 32-bit bitmaps and thus they could store up to 32 elements. Procedures with more than 32 defined clauses have to be divided into groups containing 32 clauses at most. These groups are tied in the usual way by the WAM-like instructions *try_me_else*, *retry_me_else* and *trust_me*.

The representation by bitmaps allows an easy test of failure situations[14]. Therefore all switch instructions are extended by the simple 0 test that optimizes the analysis with a fast detection of failure situations.

It is also possible to implement dynamic bitmaps, which contain an entry storing the length of the bitmap[Palmer,Naish91]. But to keep costs of bitmap manipulation down, we choose the fixed representation. Dealing with predicates with many clauses, instructions dealing with dynamic bitmaps become more important. We think that the best way is given by having both kinds of instructions. Then the compiler can choose the best class of instructions.

7 Results

The runtime results are given for the abstract machine of the functional logic language BABEL, which is a slightly modified WAM. The main difference is found in the parameter passing mechanism. For a formal description of both the language BABEL and the abstract machine the curious reader may look at [Moreno,Rodríguez89] [Loogen,Winkler91].

The current implementation consists of a compiler translating BABEL programs into code for the abstract machine. The compiler is written in C, and supports all the optimizations mentioned so far. The code delivered by the compiler is executed by an emulator, written in C, too.

The different mechanisms of clause selection are compared in the tables 3 and 4. Table 3 compares the execution times[15] of the following examples, written in BABEL mostly in a functional way:

[14] One has to check the bitmaps to 0.

[15] All examples have been executed on a Sun-Sparc-Station 1+.

fib N : a recursive definition of the Fibonacci function applied to N

rev N : a naive definition of the function reversing a list of length N

quicksort N: a recursive quicksort algorithm applied to a list of length N

queens N : the problem of positioning N queens on a NxN chess-board, so that no queen threatens any other.

popul N : is a database program reporting the surface and population of N countries. It computes all pairs of countries with population densities differing at most 5%.

program	no indexing	complete indexing	ratio
fib 20	1500 msec	1570 msec	105%
rev 30	35 msec	31 msec	89%
quicksort 30	50 msec	49 msec	98%
queens 8	3810 msec	3730 msec	98%
popul 25	290 msec	100 msec	38%

Table 3. Runtimes with and without Indexing

program	no indexing	complete indexing	ratio
fib 20	21891 − 10945 = 10946	10946 − 0 = 10946	50%
rev 30	496 − 465 = 31	0 − 0 = 0	0%
quicksort 30	641 − 484 = 157	0 − 0 = 0	0%
queens 8	28882 − 28882 = 0	7657 − 7657 = 0	27%
popul 25	676 − 676 = 0	26 − 26 = 0	3.8%

Table 4. Behaviour of Choice Points with and without Indexing

The execution times of the first three examples describe the computation times till the first solution is found. Note that although each query has only one result the remaining parts of the computation tree, that are not cut by indexing, are not visited so far. The other two examples calculate all solutions whereby the whole computation tree is traversed.

Most of the examples show some speed up. Someone may be disappointed about the small speedups. This is due to the fact that the examples contain mostly predicates with only few clauses. A more thankful example is the last one. It shows which improvements complete indexing contains. Here we reach a speedup of 3.

Table 4 focuses on the behaviour of the choice points. The entries in the second and third column are given in the form "$a - b = c$". a denotes the number of choice points, that are created at runtime, b denotes the deleted ones, and c denotes the difference. The entries in the ratio column are related to a. We see that the extension to complete indexing decreases the number of choice points significant. Often this results in a strong reduction of space consumption. In particular the runtime stack shrinks more often after a successful predicate call, because fewer choice points prohibit the deletion of environments.

[Hickey,Mudambi89] give a detailed explanation that fewer choice points lead to less trailing and fewer demand of heap space. In the BABEL machine we also need fewer savings of data stack entries. This saves both runtime stack space and time for copy operations. Table 5 gives a comprehensive survey of the possible savings

program	no indexing	complete indexing	ratio
fib 20	963264 cells	963264 cells	100%
rev 30	15172 cells	924 cells	6.1%
quicksort 30	20132 cells	856 cells	4.3%
queens 8	3192 cells	1748 cells	55%
popul 25	388 cells	296 cells	76%

Table 5. Runtime Stack Space Consumption with and without Indexing

of stack space. We recognize that the savings differ extremely, but in the average we may expect the saving of 50%. A good candidate for complete indexing is the quicksort function saving more than 95% stack memory. The Fibonacci function, however, shows that complete indexing cannot guarantee memory savings.

8 Conclusion

We have presented an alternative approach of complete indexing. The formal specification has shown the simplicity of the evaluation of the necessary information and its translation into indexing code for the Warren Abstract Machine.

Because of the (small) overhead of analyzing the whole arguments of a predicate, we cannot expect the same speed-ups as we get by an extended pattern-matcher[16], which will need more program space for doing the same complex analysis. But even in the case of small programs with few clauses we recognize some runtime speed-ups[17] in comparison with a translation without indexing.

Another important point of view is the saving of memory as we have demonstrated in the last section. Especially in functional + logic environments the number of choice points should be reduced as far as possible, to be able for tail recursion optimization and utilization of determinism. Such saving will decrease the overhead of memory management and will indirectly improve the runtime behaviour.

References

[Aït-Kaci91] H. Aït-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Logic Programming Series. Cambridge, MA, 1991

[16] as introduced in [Peyton,Jones87] for the functional environment

[17] in the BABEL environment

[Chakravarty,Lock91] M. M. T. Chakravarty and H. C. R. Lock, The Implementation of Lazy Narrowing, PLILP 91, *Lecture Notes in Computer Science*, 528, Springer Verlag, Aug. 1991

[Hickey,Mudambi89] T. Hickey and Sh. Mudambi, Global compilation of Prolog, *Journal on Logic Programming*, Vol. 7, 1989

[Kliger,Shapiro88] S. Kliger and E. Shapiro, A Decision Tree Compilation Algorithm for FCP(—,:,?), *Proceedings of the Fifth International Conference of Logic Programming and Symposium of Logic Programming*, MIT Press, 1988

[Kliger,Shapiro90] S. Kliger and E. Shapiro, From Decision Trees to Decision Graphs, *North American Conference on Logic Programming*, Austin, MIT Press, October 1990

[Korsloot,Tick91] M. Korsloot and E. Tick, Compilation Techniques for Nondeterminate Flat Concurrent Logic Programming Languages, *Logic Programming: Proceedings of the Eighth International Conference*, MIT Press, June 1991

[Loogen91] R. Loogen, From Reduction Machines to Narrowing Machines, In CCPSD, TAPSOFT 91, *Lecture Notes in Computer Science*, 494, Springer Verlag, 1991

[Loogen,Winkler91] R. Loogen and S. Winkler, Dynamic Detection of Determinism in Functional Logic Languages, PLILP 91, *Lecture Notes in Computer Science*, 528, Springer Verlag, 1991

[Moreno,Rodríguez89] J.J. Moreno-Navarro and M. Rodríguez-Artalejo, *Logic Programming with Functions and Predicates: The Language BABEL*, Technical Report DIA/89/3, Universidad Complutense, Madrid 1989

[Palmer,Naish91] D. Palmer and L. Naish, NUA-Prolog: An Extension to the WAM for Parallel Andorra, *Logic Programming: Proceedings of the Eighth International Conference*, MIT Press, June 1991

[Peyton,Jones87] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987

[Warren83] D.H.D. Warren, *An Abstract PROLOG Instruction Set*, Technical Note 309, SRI International, Menlo Park, California, 1983