

Compiling Large Disjunctions

Henk Vandecasteele

Bart Demoen

Gerda Janssens

Report CW 295, July, 2000



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Compiling Large Disjunctions*

Henk Vandecasteele

Bart Demoen

Gerda Janssens

Report CW 295, July, 2000

Department of Computer Science, K.U.Leuven

Abstract

In the context of data mining and inductive logic programming (ILP), large sets of queries must be evaluated against a set of examples. One particularly effective optimisation is to organise these sets of queries as a query pack, yielding speedup factors of up to 20. This was achieved through compiling a query pack for an enhanced WAM. The weak point in this story is that compilation of huge packs - they are represented as a nested disjunction with several thousands of goals and a nesting depth of more than 10 - with the usual techniques is too inefficient to give such learning systems also a high overall speedup when using the pack technology. Therefore, the compilation of large disjunctions with many variables had to be improved. In this paper, we investigate the problems current Prolog compilers have when dealing with such huge disjunctions. We propose some remedies and discuss the trade-offs. In particular the paper presents a new classification algorithm for variables - which is almost linear in the case of ILP - and a numbering scheme for variables that reduces the total number of variables. The paper also discusses an issue in the generated WAM code related to precise garbage collection. We have implemented a prototype compiler and present some preliminary figures. The paper also considers the possibility of incrementally compiling growing disjunctions/packs which is particularly interesting in the context of inductive learning.

*Presented at the Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages at CL2000, Imperial College, London

Compiling large disjunctions

Henk Vandecasteele, Bart Demoen, and Gerda Janssens

*Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Leuven, Belgium
{henkv,bmd,gerda}@cs.kuleuven.ac.be*

Abstract

In the context of data mining and inductive logic programming (ILP), large sets of queries must be evaluated against a set of examples. One particularly effective optimisation is to organise these sets of queries as a query pack, yielding speedup factors of up to 20. This was achieved through compiling a query pack for an enhanced WAM. The weak point in this story is that compilation of huge packs - they are represented as a nested disjunction with several thousands of goals and a nesting depth of more than 10 - with the usual techniques is too inefficient to give such learning systems also a high overall speedup when using the pack technology. Therefore, the compilation of large disjunctions with many variables had to be improved. In this paper, we investigate the problems current Prolog compilers have when dealing with such huge disjunctions. We propose some remedies and discuss the trade-offs. In particular the paper presents a new classification algorithm for variables - which is almost linear in the case of ILP - and a numbering scheme for variables that reduces the total number of variables. The paper also discusses an issue in the generated WAM code related to precise garbage collection. We have implemented a prototype compiler and present some preliminary figures. The paper also considers the possibility of incrementally compiling growing disjunctions/packs which is particularly interesting in the context of inductive learning.

1 Introduction

When inductive logic programming (ILP) is used for data mining purposes, one generates lots of hypotheses and checks whether they are valid with respect to a database (a set of examples) in order to discover the "best" hypothesis. In the case of the ILP systems TILDE and WARMR developed by the K.U.Leuven group, a hypothesis is actually a Prolog query and it is executed with respect to a Prolog program, describing general knowledge about the problem domain and also containing the actual examples. Moreover, the ILP queries have some

specific properties. First, a query is just a conjunction of atoms and different queries can have the same atoms at the start of the conjunction, i.e. they can have a common prefix. Next, during the data mining process valid queries are refined, i.e. some atoms are added at the end of the query. In both cases, it would be useful that common parts are only executed once. Current Prolog systems have no direct support for this, while tabling is no real solution.

ILPROLOG is a Prolog system that was recently developed purposefully to provide special support for optimisations and for requirements to ILP systems, in particular the validation of hypotheses. ILPROLOG is WAM-based and currently uses the XSB [14] compiler for the generation of the abstract machine code and a reader implemented by Koen De Bosschere [5]. The kernel is written in C using state of the art technology: its roots are in [8]. In order to improve the evaluation of a set of queries, we group them together in a so-called *query pack*, which is similar to a Prolog disjunction but its execution is tuned towards ILP as mentioned in Section 2. In [7] an extension to WAM supporting the execution of query packs is explained in detail. [3] reports how ILPROLOG has been used by the ILP systems TILDE [2] and WARMR [6] and how they benefit from the query pack support. Experimental results show that the evaluation of a set of queries speeds up by a factor of 20 when query packs are used. However, when query packs are used, they must be generated and compiled and this reduces the efficiency gain. During experiments for [3] we observed that it sometimes takes more time to compile the query packs than to execute them. In some cases this meant a speed loss instead of a speedup. While, with the new compiler described here, we have a net efficiency gain of a factor 4, which is considerable.

Having identified the compilation of query packs as a bottleneck in our system, we reconsidered the compilation of (large) disjunctions as this covers also the issues for query packs. The relation between query packs and disjunctions is explained in more detail in Section 2. Section 3.1 discusses the general issues of compiling a disjunction, while section 3.2 describes the specific problems arising when compiling large disjunctions. Section 4 describes the new compiler which solves the problems. Section 5 revisits reachability, information needed during garbage collection, in the context of packs because our implementation does not respect the order of branches. Finally, Section 6 discusses the incremental compilation of packs.

2 Query Packs and Disjunctions

As mentioned above ILP systems usually generate a large amount of queries. These systems want to know for each of these queries whether it fails or succeeds. Almost always these queries have common prefixes. Executing these queries as query packs [3] is a technique that combines these queries, such that

common prefixes are executed only once. Syntactically, query packs are similar to Prolog disjunctions. When grouping two queries with a common prefix, the query pack consists of the common prefix followed by a disjunction with two branches, where each branch corresponds to the part where the queries differ. Generalisation for grouping more queries is straightforward when disjunctions are also nested. In the following example the queries `q1`, `q2`, `q3` and `q4` are grouped as `qd`.

```

q1 :- a,b,c,d.
q2 :- a,b,e,f.
q3 :- a,b,e,g,h.
q4 :- a,b,i.

qd :- a,b, ( c,d ; e, ( f ; g, h ) ; i ).

```

Using disjunctions avoids recomputation of common prefixes such as `a,b` and `a,b,e` in the previous example. In the context of ILP we can go further: for each original query, we only want to know whether it succeeds or not ¹. Due to backtracking, normal Prolog execution detects all successes. Using cuts (or some of its special variants) does not help as it could prevent backtracking into branches that have not yet succeeded. Therefore, a dedicated execution for such disjunctions, then called *query packs*, is needed. The basic idea is that as soon as a branch succeeds, it will no longer be considered on backtracking. Suppose that in the above example the `c,d` branch (corresponding to the query `a,b,c,d`) succeeds, but the calls to `e` and `i` fail and that we backtrack to `b`. Now, it is as if the `c,d` branch is not there any longer and only the other alternatives have to be tried. The paper [7] describes possible implementations and shows that an extension of the WAM leads to better results than a meta-interpreter or a source-to-source transformation. This has been confirmed by the experiments for [3] where also the need of a good query pack compiler was observed. Note that the WAM implementation possibly changes the order in which the remaining branches are considered. For the example above, it might be the case that after backtracking to `b` the branch with `i` is considered before the originally second one.

In order to get a better idea of for example typical TILDE query packs, we report the following properties for a set of query packs taken from the experiment on the Mutagenesis data set [12,3]:

- *Nbq* is the number of queries in a pack.
- *Nbv* is the number of variables in a pack (as computed by `numbervars`).
- *MaxL(F)* with *MaxL* the maximum length of the queries (i.e. the maximum number of atoms in the queries of the pack) and *F* the fraction of the queries

¹ we are not interested in answer substitutions, neither in multiple successes.

Pack	Nbq	Nbv	MaxL(F)	MinL(F)	Nbd	MaxBr(F)	MinBr(F)	Br(F)
pack1	46	141	3(100)	3(100)	1	46(100)	46(100)	46(100)
pack2	432	818	6(64)	4(11)	8	57(12)	48(12)	56(50)
pack4	3616	6306	9(80)	7(2)	64	59 (5)	57(56)	57(56)
pack6	12616	21546	12(86)	10(0.6)	216	76(0.5)	59(69)	59(69)
pack7	1498	2667	7(74)	5(3)	27	58(7)	52(4)	57(44)
pack8	16744	28905	10(78)	7(0.3)	295	60(1)	57(39)	58(46)
pack9	39706	68024	11(82)	8(0.1)	689	66(0.1)	58(49)	58(48)

Table 1
Some properties of typical query packs.

that do have the maximum length.

- $MinL(F)$ with MinL the minimum length of the queries in the pack and F the fraction of the queries that do have the minimum length.
- Nbd is the number of disjunctions in a query pack (where e.g. (a;b;c;d) is counted as one disjunction with a branching factor of four or having four branches).
- $MaxBr(F)$ is the maximum number of branches MaxBr in the disjunctions, with F the fraction of the queries that have the maximum branching factor.
- $MinBr(F)$ is the minimum number of branches in the disjunctions with F the corresponding fraction.
- $Br(F)$ is the branching factor Br that corresponds to the largest fraction (F) of disjunctions.

3 Compiling disjunctions

3.1 General issues in compiling a disjunction

Standard WAM does not prescribe how to compile clauses containing an explicit disjunction - the ; control construct. E.g. [1] does not mention such disjunctions at all. As a consequence, Prolog implementations based on WAM have designed their own particular way of dealing with the disjunction. One approach is to introduce a **new predicate** for the disjunction and source-transform the disjunction away into a sequence of clauses, as exemplified in the following:

a :- b, (c ; d), e.

is transformed to:

a :- b, newpred, e.

```
newpred :- c.
newpred :- d.
```

This approach simplifies in many ways the rest of the implementation- see later, but also has its pitfalls: when the branches of the disjunction contain a cut (!/0), the new predicate must contain an ancestor cut, which means that the transformed program is no longer ISO-compliant. Also, if the goals `a,b,c,d` and `e` contain variables, one would like to give the `newpred` as few arguments as possible: this poses a complexity issue and is one of the reasons why in our setting, this is not an optimal choice. Next problem is that the choice point created for `newpred`, contains most often more slots than the choice point created for an inlined compilation of the disjunction (see later). Finally, the transformation in the above example leads to the creation (at runtime) of 2 environment frames instead of one, which is a time and space loss.

The other solution to compiling a disjunction is to **inline** it: this is done for instance in systems like XSB [14] and Yap [15]. Since `ILPROLOG` is currently based on the XSB compiler, we have inherited the approach. For a clause like

Example 3.1

```
a :- b, (c ; d), e.
```

The following code is generated:

```
allocate 2
call b/0
trymeorelse 3
call c/0
jump 4
label 3
trustmeorelsefail
call d/0
label 4
deallex e/0
```

This approach doesn't suffer from problems with larger choice points, more environments or with cuts, but is tricky: optimal classification of variables in temporary and permanent is not obvious, the notion of the first occurrence of a (permanent) variable needs to be re-investigated (shown in an example later) and even the permanent variable numbering (assignment to an environment slot) is no longer obvious.

- **Example 3.2**

```
a :- (b(X) ; c(X)). % X can be void
a :- (b(X) ; c(X,X)). % X can be temporary
```

- **Example 3.3**

```
a :- (b(X), c(X) ; d(Y), e(Y)).
```

can be rewritten to:

```
a :- (b(X), c(X) ; d(X), e(X)).
```

showing that X and Y (of the original clause) share the same slot.

- **Example 3.4**

```
a :- (b(X) ; c), d(X).
```

The occurrence of X in d(X) is not a first occurrence, even though following the second branch of the disjunction, it wasn't initialised - meaning the compiler must emit code that initialises X in the second branch.

Implementations have not always gotten this right: SICStus Prolog [13] does not give the singleton warning for Example 3.2 and XSB used to choke on Example 3.4. Neither Yap nor XSB make X and Y share the same slot in Example 3.3. Some of the problems compiling disjunction inlined disappear if the branches don't meet at the right. That's probably why M-Prolog used to restrict the usage of disjunction to non-meeting branches.

Another problem with inlined disjunction is related to precise garbage collection: systems like Yap and ILPROLOG put at each continuation point in the code the information which permanent variables are live at that point. This is similar to having pointer maps as described in [4]. So, the actual code for Example 3.1 is

```
allocate 2
call a/0
active_yvar 0
trymeorelse 3
active_yvar 0
call b/0
active_yvar 0
jump 4
label 3
trustmeorelsefail
call c/0
active_yvar 0
label 4
deallex d/0
```

where the pseudo-instruction *active_yvar* has as arguments some description² of which permanent variables are active at that (continuation) point. If the *(re)trymeorelse* instruction makes sure that the continuation pointer saved in its choicepoint points to the *active_yvar* instruction following the *trymeorelse*, one can have a uniform view - during marking - of reachability.

² might be through a table.

	Yap-2.2.1	Yap-2.2.1	XSB	XSB	SICStus-3.8.2
	time	permvars	time	permvars	time
pack1	0.01	2	0.57	2	0.12
pack2	0.26	13	24.39	13	2.31
pack4	*	*	1641.9	83	60.25
pack6	*	*	-	-	597.44
pack7	2.37	38	282.29	38	14.24
pack8	*	*	-	-	1194.6
pack9	*	*	-	-	M

Table 2
Current Prolog systems and large disjunctions

All these potential pitfalls related to the inlined compilation of disjunction have at least one implementor made regret this choice [10].

3.2 Issues in compiling a large disjunction

The problems current compilers have with large disjunctions, are mainly that

- (i) the complexity of the overall compilation is non-linear in the size of the disjunction
- (ii) large disjunctions contain too many permanent variables. This is a problem because Prolog implementations typically have a fixed maximum number of permanent variables, because it affects all kinds of design decisions. Just think about get/put/unify instructions that refer to permanent variables by their number and which have a fixed format for that number (8/16 bits).

Both problems will be addressed in section 4. Here we give some empirical data for our claims in the form of a table which contains for a number of systems, some compilation characteristics for a set of packs taken from the Mutagenesis data set [12,3].

The measurements were performed on a Pentium II, 266MHz, 128Mb. Timings are in seconds. In the table, a * means that execution was aborted because of a segmentation violation, M means that the process got out of memory, and - means it took just too long to wait for. The times for Yap and SICStus include read, compile and load. For XSB they include read, compile and produce a file with the abstract machine code. The timings of XSB were obtained by executing a slightly adapted XSB compiler under ILPROLOG . This doesn't affect the number of permvars. Using this XSB compiler under ILPROLOG is 2 to 4 times faster than using the original within the XSB-system.

The table clearly shows that ordinary compiler technology is inadequate for compiling large disjunctions. In Section 4.5, Table 3, we'll show the figures for the new compiler.

4 The new compiler

It turns out that current compilers do not cope adequately with such large clauses with many disjunctions and many variables. A major issue is the time-complexity of the algorithm for classifying the variables. In our case it is essential that such an algorithm is linear in the number of variables occurring in the system. Due to the huge amount of variables (quickly approaching hundred thousand variables, see Table 1) it is essential to determine the exact scope of the variables, so that the space for the variables in the environment can be reduced as shown in Example 3.3. It is obvious that correctness of the generated WAM-code should be preserved, in particular special care should be taken for the initialisation problem as shown in Example 3.4. A new ILPROLOG compiler is being developed that tackles the above problems, as is explained in the next subsections. This compiler is derived from the compiler used in [9].

4.1 *Classifying variables for disjunctions that do not meet*

In the context of the ILP applications, there definitely was a need for a better algorithm for classification of variables: variables are either permanent, temporary or void. Moreover, we can focus on clauses with disjunctions that do not meet, i.e. the nested disjunctions in a clause form a tree and no disjunction is followed by another call. The *classification algorithm* should recognise that the same variables that occur in different branches are actually different variables. In addition, for register allocation and other optimisations, the number of occurrences of variables should be counted, not only for temporary variables, but also for permanent variables. The algorithm should be blazingly fast, to be able to handle query packs with loads of variables and many branches.

The ILPROLOG compiler is currently written in Prolog. Its classification algorithm first gathers the relevant information and then the classification can be done. The classification algorithm is based on some abstract syntax of the clause, generated in some earlier phase of the compiler: each occurrence of a variable is replaced by a term `var(Orig, New)` where `Orig` is the original variable in the clause and `New` is a fresh variable which is different for each occurrence of the variable in the original clause. During the classification algorithm the `Orig` variable will be used to store temporary information. The `New` variable will contain the final result, namely the type of the variable – void, temporary or permanent – and the number of occurrences of the variable.

As an example we will use:

Example 4.1

q:- a(X), (a(X, Y);a(X, Y, Y)).

The outcome of the algorithm should classify X as a permanent variable. The first occurrence of Y as a void variable, the last two occurrences of Y as a temporary variable.

The input for the algorithm described here for example 4.1 is:

Example 4.2

q:- a(var(X, New₁), (a(var(X, New₂), var(Y, New₃));
a(var(X, New₄), var(Y, New₅), var(Y, New₆))).

The classification algorithm will parse the tree of disjunctions depth first, from left to right. During this traversal each branch in a disjunction gets a unique number. This number always grows during the parsing. Since a branch in a disjunction may contain a disjunction as well, we get a tree structure. Given the numbering scheme above each occurrence of a variable in the clause will have a *branch number*, namely the number of the current branch. We also define a *root set*, this is the set of all numbers on the branches from the root of the clause to the branch the variable occurs in, thus including its *branch number*.

For tracking information on permanent and temporary variables the classification algorithm divides clauses into *chunks* which are also numbered. The idea is that variables occurring in only one chunk are not permanent. When defining chunks, we make a distinction between Prolog builtins that are inlined by the underlying Prolog system and builtins that are defined by Prolog clauses. For chunks, the latter are considered as ordinary (Prolog) calls, and the former are simply called builtins. When scanning a clause from left to right, the first chunk is the part of the clause beginning with the head of the clause and ending with the first call in the body (if any). Note that there can be inline builtins between the head and the first call. The next chunks are parts of the clause starting just after a previous chunk and ending with a call (if any).

The chunks are also numbered depth first, from left to right in the tree of disjunctions. Actually one can use the chunk number of the first chunk of a branch as the *branch number* for that branch.

During the parsing of the clause the variables `Orig` are instantiated to a list of terms of the format `branch(Occurrences, branchNr)`. The same variable occurring in two different branches can be viewed as different variables whenever there is no occurrence of the same variable in a common branch on the path from the root of the clause. For each such *different* variable a term `branch(Occurrences, branchNr)` will be added to the list.

On the first occurrence of a variable the `Orig` variable is instantiated as

`[branch(Occurrences, BranchNr)]` with `BranchNr` its *branch number*, and we store the `New` variable in a data structure contained in `Occurrences`. On subsequent occurrences of this variable we have to check whether the `BranchNr` is a member of the *root set* of that occurrence. If this is the case we have found a next occurrence of the same variable and we add the `New` variable to the data structure in the variable `Occurrences`. In case `BranchNr` is not a member of the *root set* we have found an occurrence of the variable which is syntactically identical, but is actually another variable. In this case we add a new term `branch(Occurrences, BranchNr)` to the list in variable `Orig`. Moreover, since we are numbering depth first, from left to right we will never find another occurrence of a variable which belongs to a term `branch(Occurrences, BranchNr)` which was is not the last created one. Therefore we adapt this list in variable `Orig` using destructive assignment (`setarg/3`): we put the new term in front of the list. Then we always have to add subsequent occurrences of a variable in the first term of the list, or add a new term to the list.

Inside the variable `Occurrences` we have again a data structure to keep information necessary to make the difference between void, temporary an permanent variables. This `Occurrences` variable is again a list of terms `chunk(NewVars, ChunkNr)`. Whenever we find an occurrence of a variable we check if the number of the `Chunk` where the variable is found is equal to the `ChunkNr` of the first term of the list `Occurrences`. If this is the case, we add the `New` variable to the list `NewVars`, again in front of the list using `setarg/3`. If this `chunkNr` is not equal, we create a new term `chunk(NewVars, ChunkNr)`, which is put in front of the list `Occurrences` again using `setarg/3`. Afterwards we can safely check only the first element of the list `Occurrences` since `chunk` numbers are increasing.

After parsing the first `Branch` of the code of Example 4.1 we have the following instantiation of variables:

Example 4.3

```
X = [branch([chunk([New2], 2), chunk([New1], 1)], 1)],
Y = [branch([chunk([New2], 2)], 2)]
```

The final instantiation if the variables after complete parsing:

Example 4.4

```
X=[branch([chunk([New4], 3), chunk([New2], 2), chunk([New1], 1)], 1)],
Y=[branch([chunk([New5, New6], 3)], 3), branch([chunk([New2], 2)], 2)]
```

Given the data structures build during the traversal of the complete clause with its disjunction tree, the next step is the actual classification of the variables. The result of the traversal of the clause returns a list of the variables in the clause. For each of these variables we consider the following cases:

- If the variable has several `branch/2` terms then each of these terms corresponds to a different variable.

- If a `branch/2` term contains only one occurrence of the variable, we classify the variable as `void` and `New` is instantiated to the term `void`.
- If the list contained in the first argument of the `branch/2` term has only one element, then the variable occurred only within one chunk of the clause implying that it is a temporary variable. All `New` variables of all occurrences within `Occurrences` of this `branch/2` term are unified and instantiated to the term `temp(_, Count, _)`, with `Count` the number of occurrences of this variable. The other two arguments will be used later in the compiler, mainly for register allocation.
- If the list contained in the first argument of the `branch/2` term has more than one list, we have a permanent variable. Similarly, all `New` variables are unified and instantiated to the term `perm(_, Count, _)`.

To allow the garbage collector to collect the data structures generated during the classification we overwrite the `Orig` variable with the atom `[]`. The output of the algorithm for Example 4.1 is then:

Example 4.5

```
q:- a(var([],perm(1)),
      (a(var([],perm(1)),var([],void));
       a(var([],perm(1)),var([],temp(1)),var([],temp(1))))).
```

This concludes the classification algorithm for variables for disjunctions that do not meet. Remains to say a word on time complexity. All operations done at each occurrence of a variable are constant time except one. Namely the test whether the current *root set* contains the *branch number* of the first occurrence. The size of the root set is bounded by the depth of the tree of the disjunctive clause. Then if d is the depth of the tree, n the number of occurrences of variables and m a measure for the size of the clause we have that the time complexity is of the order $n*d + m$. Since d will always be rather small compared to n and m , the algorithm behaves almost linear.

4.2 Classifying variables for disjunctions that meet

The algorithm described above does not work in case the branches in a disjunction meet again. The structure in terms of disjunctions is then no more a tree. Luckily the large clauses we had to deal with the disjunctive branches did not meet, but it would be nice to have an algorithm for general clauses that behaves almost linear as well.

4.2.1 Motivation

An example of a program problematic for the previous algorithm is:

Example 4.6

```
p:- (a(X); b(X)), c(X).
```

The variable X would be regarded as two different variables in the two branches of the disjunction, but unfortunately the occurrence of X in the call to $c/1$ makes them the same variable. This does not look too difficult to fix. But imagine a situation like:

Example 4.7

$p:- (a(X); (g, (b(X); c(X)), (d(X);e(X))))$.

For this program the analysis should decide that the variable X actually represents two variables. One variable that occurs in the call to $a/1$, and all other occurrences of X is the other variable. One of the difficulties is to recognise at the time of seeing the term $d(X)$ that the occurrence of X in $b/1$ and $c/1$ are now the same variable. While the occurrence of X in $a/1$ is another variable.

4.2.2 The algorithm

The solution to the problem is to use a different numbering scheme for the branches. Situations like in Example 4.7 can be overcome by using branch numbers in the disjunctions with calls to $d/1$ and $e/1$ which are smaller than the branch numbers in the disjunctions with calls to $b/1$ and $c/1$, but larger than the number of the branch starting at the call to $a/1$. For this purpose we need a numbering scheme in which we can always find a number between two given numbers: Rational numbers would be perfect, but we have used reals! Then we have at each point in the algorithm in addition to the number of the current branch a *delta*. All numbers in descendants of this branch should take numbers between the current branch number and the current branch number plus *delta*. Whenever we encounter a disjunction that meets we use numbers within that disjunction between the current branch number plus *delta/2* and the current branch number plus *delta*. All branches after the disjunction will get numbers between the current branch number and the current branch number plus *delta/2*. While in the previous algorithm branch numbers always grow during the algorithm, here these numbers can get smaller during the process.

Whenever we encounter an occurrence of a variable where the current branch number is smaller than the branch number of the variable in the first term in the **Orig** variable, we detected a variable created in a disjunction that appears in the code after the branches of the disjunction have met. Possibly we have to merge different terms in the **Orig**, because the algorithm decided at an earlier stage that it has found different variables. Not all of the terms in **Orig** variable will have to be merged, only the ones where its corresponding branch number is larger than the current.

The problem that arises now is the new branch number that should go with these merged branches. In case of Example 4.7 it should be the branch number of the branch starting at call $g/0$. Given the current representation

this number is quite hard to find! Instead of keeping the branch number of the first occurrence of the variable we keep the complete *root set* including the branch number of the first occurrence of the variable. Then finding the new *root set* of branches that must be merged is quite easy. We just have to remove all branch numbers from the old *root set* that are larger than the branch number of the new occurrence of the variable.

Computing these branch numbers for Example 4.7 is quite technical but we worked it out for the technical readers among you. We annotated the clause with the tuples **b(BranchNr, Delta)** at each start of a branch:

Example 4.8

b(0,256) p:-
(b(128,64) a(X); **(b(192,64)** g, **(b(224,16)** b(X); **b(240,16)** c(X)),
(b(208,8) d(X); **b(216,8)** e(X)))).

It is important to note that the branch numbers in the disjunction containing d and e are smaller than those in the disjunction containing b and c, but larger than the number of the branch starting at a. Meaning that X in a(X) is a different variable.

When working with this new numbering scheme, keeping *root sets* instead of a branch number in each term in list `Orig` and merging terms when detecting shrinking branch numbers we can solve the problem. The remainder of the algorithm, parsing the clause and final classification, remains the same.

With respect to time-complexity this algorithm is not as good as the previous one. Merging different terms in the list `Orig` cannot be done in constant time. Currently this is linear in the number of occurrences of the variables involved in the operation. Considering that this part of the compiler is only used for hand-written code, meaning relatively small clauses, we do not really care about this potential quadratic behaviour.

4.3 Reusing slots for permanent variables in the environment

The usual option for assigning slots in the environment is to assign a slot in the environment whenever a permanent variable is detected. This can be done during analysis of the variables and results in a one on one mapping between permanent variables and environment slots. For a disjunctive clause one could use the same slot for different variables of which the lifetimes do not overlap. This can be seen in Example 3.3. For handwritten code such reuse will only result in minor advantage, but in our case such reuse is essential since the computer-generated clauses we are dealing with will exceed 65536 (2^{16}) permanent variables. When reusing slots in the environment, this can easily be reduced to less than 100 simultaneous permanent variables. Without complicated data structures or another pass through the clause it is quite difficult to assign slot numbers in the classification phase of the program.

Therefore we choose to assign slots to the permanent variables dynamically during the generation of code.

In case of disjunctive clauses where branches never meet the scheme is quite simple. During the generation of code the compiler remembers the next available slot in the environment. Whenever the first occurrence of a permanent variable is encountered this number is used and the next number to be used is incremented by one. If the compiler finds a disjunction, the next number to be used can be the same for all branches. Indeed when the first occurrence of a permanent variable is found within one branch, this permanent variable will be surely dead whenever another branch in the disjunction is executed and can be safely reused.

In case of disjunctions that do meet, we can never be sure that permanent variables that occurred the first time in a branch will not be used in another branch. An occurrence of this variable in the code after the branches have met, could link another branch. For this purpose we count the occurrences of all permanent variables in a branch. If this count is not equal to the total number of occurrences – we collect this information in the classification phase, then the slot allocated for this permanent variable cannot be safely used by other branches in the disjunction. Still we use the same method as above and we start with the same slot in the environment to be used first for the different branches of disjunctions. Next to that we have a list of forbidden slots in the environment: The slots that were used for permanent variables in previous branches and not all occurrences of that permanent variables were seen in that branch.

4.4 The initialisation problem for disjunctions that meet

Exactly those permanent variables from the previous sections where:

- the first occurrence was seen by the compiler in some branch of a disjunction, and
- not all of the occurrences were seen by the compiler in that very same branch cause problems with initialisation. Since not all occurrences were seen in the same branch of the disjunction, there must be an occurrence in some code after the disjunction, when the different branches of the disjunction have met. Of course during execution one never knows which branch of the disjunction has been executed before the code with these extra occurrences. In these extra occurrence one must assume that the variable is initialised, if the wrong branch was taken it is not.

For this reason we need some bookkeeping for all such variables that:

- occur in a some branch of a disjunction,
- do appear in some code after the disjunction,

	XSB time	XSB #perm	SICStus time	ILPROLOG time	ILPROLOG new #perm	ILPROLOG classic #perm
pack1	0.57	2	0.12	0.05 (0.05)	2	2
pack2	24.39	13	2.31	0.66 (0.56)	7	13
pack4	1641.9	83	60.25	10.62 (5.77)	9	83
pack6	-	-	597.44	117.21 (20.69)	11	257
pack7	282.29	38	14.24	2.8 (1.84)	8	38
pack8	-	-	1194.6	234.25 (28.25)	11	365
pack9	-	-	M	1445.47 (57.06)	14	822

Table 3

Our new ILPROLOG compiler compared to the XSB compiler we used before

- do not appear in the code before the disjunction,
- and do not appear in another branch of the disjunction.

WAM code should be generated to initialise this permanent variable in this other branch where it does not appear. Of course at the end of the WAM-code for this branch, when it is sure that the branch will not fail.

4.5 Evaluation

In Table 3 we can see a comparison between the XSB compiler [14] we used before in our ILPROLOG system and our new compiler. We included the SICStus [13] timings for comparison as well. For both systems reading the highly disjunctive clause from file, compiling and write a file with the WAM code.

For the new ILPROLOG compiler the time between brackets is the compilation time only: the reader is responsible for the difference between the two numbers. It is relevant for us to have this number, as eventually, the connection between the new compiler and the system will be without files. And yes, we can improve the reader within the ILPROLOG system a lot!

From the table we can see that the compiler has a very reasonable behaviour concerning execution time. The main reason for this is the difference in time-complexity for the classification of variables when disjunctions are involved.

Another striking aspect is the number of slots needed in the environment. When using a classic scheme the number of slots needed goes up very rapidly (last column). Note that theses are the same as in the XSB permvars column. When reusing slots this number of slots can be reduced to a very small amount (column labeled new #permvars). Remark that many Prolog-systems get into trouble when having more than 256 permanent variables. This would be the

case for pack number 6, 8 and 9.

5 Precise reachability revisited for packs

During garbage collection of the heap pointers from within the environment to the heap should be used in the marking phase. Unfortunately there could be permanent variables within the environment that are already dead! Even worse permanent variables that have not been initialised, ILPROLOG does not clean the environment on creation, could contain invalid information. For this purpose we use the technique of “active yvars” introduced in Section 3.1. After each call we have an extra opcode which has as a parameter all permanent variables alive after the call. During garbage collection the algorithm can find this information using the continuation pointer in the environments. In the case of disjunctions we need also information on the permanent variables alive at the time of creation of the choicepoint. Thus a *try_me_else* instruction needs “active yvars” as well. Moreover within normal disjunction we can improve the precision of the garbage collection by adding an “active yvar” to *retry_me_else* instructions as well, because some permanent variables alive at in the first alternatives of the disjunctions could be dead in later branches of the disjunctions, as is the case in the second branch of Example 5.1 with variable X.

Example 5.1

$p:- p(X), (q(X); d; e).$

Unfortunately during execution of packs the order of execution of the different branches is not known in advance, as discussed in Section 2.

Example 5.2

$p:- p(X), (q(X); r(X); s).$

for instance in Example 5.2 branch *s* could be executed before branch *r(X)*. This means we cannot shrink the set of permanent variables, when we move from one branch to another in the disjunction. So we have to stick to the set of permanent variables alive at the time of starting the disjunction with possible imprecise garbage collection.

6 Incremental compilation of query packs

Even though the current query pack compilation times are acceptable and scale well, it remains worthwhile looking at speeding up the compilation process. Apart from the obvious *rewrite it in C*, we want to explore the possibility of incrementally compiling query packs: During a particular learning session usually a sequence of packs of queries needs to be evaluated, one at a time. The packs in this sequence are not independent. The n-th pack is basically

the (n-1)-th pack from which some branches have been pruned and to which the remaining branches are extended. E.g. pack 1 could be³:

a, (b, (c ; d) ; e, (f ; g) ; h)

and pack 2 could be:

a, (b, d, (x ; y) ; e , f, (u ; v) ; h, (i ; j))

The idea of incrementally compiling the pack is to transform the code for pack *i* into the code for pack (*i*+1). We just point out some of the issues one has to take care of:

- One of the things that can happen by extending a branch, is that a last occurrence of a permanent variable, is no longer last: in `ILPROLOG` this poses no particular problem as variables are always initialised on the heap, so `ILPROLOG` doesn't have a `PUT_UNSAFE_VALUE` instruction. If it would have had one, then it would be a matter of being able to find out which `PUT_UNSAFE_VALUE` instruction needs to be changed into a `PUT_YVAL`.
- In the inductive learning setting of `TILDE`, each branch of the pack ends with a call to a predicate that reports which branch succeeded. This call has as argument an indication of the branch which might need to change for the new pack.
- Cutting a branch away (from a certain point) might delete a set of permanent variables from the pack: it would be best if the incremental compiler knows about it so that these slots can be reused. While doing such reuse, the idea is not to touch the assignments of surviving permanent variables. This means that an incrementally compiled pack could potentially use more than the minimal environment. However, the allocation policy of the compiler described earlier, is such that slots of removed variables are either used by other variables (in parallel branches) or these slots have a highest number that can be freed without extra cost of reallocating other variables. This is true because, within one branch, first occurrences of permanent variables are numbered sequentially. Then permanent variables initialised in deleted branches, must have the highest number.
- Typically when a branch is extended, the extension will contain variables that occurred already. This means that the classification of a variable might need to be changed from *void* or *temporary* to *permanent*. It seems better to avoid such change of classification and let the pack compiler classify every variable as permanent, even if `WAM` would classify it differently.
- Reachability information about the permanent variables must also be adapted.

³ we keep using the notation for disjunction for convenience

In order to make this incremental compilation easier, it is important that there is cooperation with the pack generation phase: currently, the generation of the (extensions to a) pack is totally separated from the compilation process itself. However, the generation phase *knows* that it builds an extension with variable that existed already, because it makes such decisions on the basis of the language bias. The idea is clear: the generation phase can convey such information to the compiler, so that true incrementality can be achieved.

7 Conclusion and Future Work

The speedups obtained by packs are impressive, and especially their scalability - now that we have the better compiler - is most comforting. However, other techniques must be incorporated in ILPROLOG to make other parts of the system scale well: support for very large datasets, fast switching between examples and better indexing than Prolog systems usually provide. Especially the latter is of the kind that could *counteract* the speedups of packs. However, we anticipate that there will always be a net gain of packs.

Acknowledgement

We thank the people from the learning group, in particular Hendrik Blockeel and Jan Ramon for teasing us with such nice problems. The third author was for this research partly funded by the project FWO-VI. G.0246.99 and partly by the Esprit project Aladin 28623.

References

- [1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991. See also: <http://www.isg.sfu.ca/~hak/documents/wam.html>.
- [2] H. Blockeel and L. De Raedt. *Top-down Induction of First Order Logical Decision Trees* Journal of Artificial Intelligence, Vol. 101, 1-2, pp. 285-297, June 1998.
- [3] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. *Executing Query Packs in ILP* Accepted for the 10th International Conference on Inductive Logic Programming (ILP 2000), July 2000.
- [4] P. Branquart and J. Lewi. A scheme of storage allocation and garbage collection for ALGOL 68. Proceedings of the IFIP Working Conference on ALGOL 68 Implementation, July 1970.

- [5] Koen De Bosschere. <http://www.elis.rug.ac.be/~kdb/>
- [6] L. Dehaspe and H. Toivonen. *Discovery of frequent Datalog patterns* Data Mining and Knowledge Discovery 3(1): 7-36, 1999.
- [7] B. Demoen, G. Janssens, H. Vandecasteele. *Executing Query Flocks for ILP* Proceedings of BENELOG'99, Maastricht, 5 November 1999. Also available on the web: http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=18794
- [8] B. Demoen, P-L. Nguyen. *So many WAM variations, so little time*. To appear in the Proceedings of CL2000, London, July, 2000
- [9] H. Vandecasteele. Compiling Prolog to ANSI C. In Y. Deville, editor, *Proceedings of the Eight Benelux Workshop on Logic Programming*, 1996.
- [10] Vitor S. Costa - private communication about Yap - beginning 2000.
- [11] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.
- [12] A. Srinivasan, S.H. Muggleton, and R.D. King. Comparing the use of background knowledge by inductive logic programming systems. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, 1995.
- [13] <http://www.sics.se/sicstus/>
- [14] <http://xsb.sourceforge.net/>
- [15] <http://www.ncc.up.pt/~vsc/Yap/>