Compilation of Compound Terms in Prolog

Micha Meier

ECRC-ECRC-95-12

technical report ECRC-ECRC-95-12

Compilation of Compound Terms in Prolog

Micha Meier

European Computer-Industry Research Centre GmbH (Forschungszentrum)
Arabellastrasse 17
D-81925 Munich
Germany
Tel. +49 89 9 26 99-0
Fax. +49 89 9 26 99-170
Tlx. 52 69 10

For more information please contact : michaecrc.de

Abstract

The execution of a compiled Prolog program can spend a significant amount of time in the unification of compound terms. We show that in the Warren Abstract Machine [7], the approach to compile this unification may be unnecessarily inefficient. When we analyse what are the redundant operations that the WAM executes, we can see that the inefficiency is mainly caused by the *breadth-first* approach to traverse the structures during the unification. We present here a method to compile the unification of compound terms which is based on a *depth-first* approach and show that it is both more general and more efficient than that of the original WAM. Furthermore we present a more efficient approach to compile compound terms in the body and also describe several possible optimizations. Our method was used in the implementation of the SEPIA system [4] developed at ECRC.

1 Introduction

The execution of Prolog programs consists mainly of procedure invocation and unification. The unification itself is estimated to account for about 70 % - 80 % of the execution time. The unification of compound terms, i.e. expressions $f(arg_1, \ldots, arg_n)$, may consume a considerable amount of time, because during the unification these compound terms have to be constructed or scanned. For example, on a set of 15 large benchmark programs measured with an emulator-based Prolog system [4], about 25 % of all emulated abstract instructions accounted for the unification of compound terms.

The WAM [7] presents a complete and compact scheme to compile compound terms, both in head and subgoal arguments. We have analyzed the WAM behavior during the unification of compound terms and have found that sometimes it is not efficient enough, because redundant operations are executed. We will first describe the problems in the WAM and then present a different scheme which tries to avoid them. The reader is assumed to be fluent in the WAM, for a good introduction see e.g. [1].

2 Compound Argument Unification in the WAM

In compiled Prolog, the compound arguments are handled differently in the head and in the body of a clause. A compound term in the clause body is compiled into a sequence of instructions that actually create this term on the heap and put a reference to it into an argument register. A compound term in the head is compiled into instructions which perform the unification with the corresponding argument in the register. The unification is a general two-way matching procedure, which can result in binding variables both in the caller and in the callee. Therefore, the unification instructions must be able to both decompose the caller argument and check if it matches with the clause head, and to create the representation of a compound term on the heap in case that the corresponding slot in the caller argument is a variable. This is the so-called *structure copying* as opposed to *structure sharing* where new structures are not explicitly created, but shared instead.

2.1 Head Arguments

A compound head argument represents a tree structure which has to be traversed during the unification with the caller. The main contribution of the WAM here is the introduction of the **mode** flag to which some instructions are sensitive. This makes it possible to generate only one instruction per source item and the bytecode is very compact. If the procedure argument is being decomposed in the unification, that is, when its main functor is identical with the main functor of the head argument, the machine is said to be in **read** mode. Conversely, if the caller argument is a variable, the machine is in **write** mode where the structure will be constructed on the heap. The **unify** instructions that are emitted for the arguments of the structure depend on the current mode: in the **read** mode they test the type of the caller argument, in the **write** mode the argument is created on the heap. The arguments of a structure are unified in the same mode as the structure functor. When the structure is being constructed on the heap, its arguments are pushed using the heap top pointer **H**, when an existing structure is decomposed, the **S** pointer is used to point to the currently unified argument. For example, the head structure in

```
p(f(g(...), h(...), k(...))) :- ...
```

is in the WAM compiled into

```
get_structure     f/3, A1
unify_variable    X1
unify_variable    X2
unify_variable    X3
get_structure     g/n1, X1
unify_ ...        <arguments of g>
...
get_structure     h/n2, X2
unify_ ...        <arguments of h>
...
get_structure     k/n3, X3
unify_ ...        <arguments of k>
...
```

If this procedure is called with **?- p(f(g(a), B, C))**, the first and second block of **unify** instructions is executed in **read** mode because the head compound term matches the corresponding functors **f/3** and $\mathbf{g/n_1}$ in the caller. The remaining two blocks of **unify** instructions are executed in **write** mode, because the structures $\mathbf{h/n_2}$ and $\mathbf{k/n_3}$ must be constructed and bound to **B** and **C** respectively.

Let us consider this example from another point of view. We can see that the code generated by the WAM is the same as the code for (when **=/2** is expanded in-line)

p(f(X1, X2, X3)) :- X1 = g(...), X2 = h(...), X3 = k(...).

where **X1**, **X2** and **X3** are temporary variables. The unification of the main structure is completely separated from the unification of its compound arguments. As the unification of the temporaries can be written in any order, the traversal of the rest of the tree can be made in any manner, as long as this way of assigning compound arguments to temporaries is kept.

This model is very easy to understand and efficient enough to implement on bytecode emulators, however it has several inherent problems:

- The **write** mode is not propagated to the substructures. Whenever a structure is unified in **write** mode, it is clear that all its substructures will also be unified in **write** mode. The WAM does not use this property, in a substructure in **write** mode the following is executed:
  - the instruction **unify_variable** pushes a new free variable on the heap and stores a pointer to it in a temporary variable
  - The instruction **get_structure** later dereferences this variable, finds out that it is uninstantiated (although this is known at compile time),

and thus it sets the **write** mode. The variable is then bound to a new structure on the heap and for this binding even a trail test has to be performed.

Several memory accesses, dereferencing, tag test and trail test are redundant. This is the consequence of the fact that the unification of a term is completely separated from the unification of its compound arguments.

- References to all substructures are saved into temporary variables. Although Prolog programmers do not seem to like deeply nested structures (except for tail-recursive ones like lists), often their arity might be quite high [3] and so many temporaries are necessary. This increases the number of used registers, or causes additional memory accesses for temporaries allocated in memory.
- Every **unify** instruction must test the **mode** bit. In fact every **unify** instruction is just a shorthand for two different instructions, one **read** and one **write**. In native code implementations, and especially on processors with an instruction pipeline, these frequent tests break the instruction flow and slow down the execution. Most commercial systems split the **unify** instructions explicitly to avoid the tests.
- In a term like **f(g(a), h(b), i(c))**, the code starts with

get_structure	**f/3,** $\mathbf{A}_i$
unify_variable	$\mathbf{X}_1$
unify_variable	$\mathbf{X}_2$
unify_variable	$\mathbf{X}_3$

  If the caller argument is **f(g(b), B, C)**, the unification of **g(a)** and **g(b)** fails and so the two **unify_variable** instructions were useless. The same effect could be achieved by just saving a *pointer* to the second argument.
- If the information about the instantiations of the arguments is available, e.g. in the form of mode declarations, it can be used only if the **unify** instructions are explicitly divided into **read** and **write** instructions. The compiler can then suppress generation of code that would never be executed.
- This sort of breadth-first traversal can be always simulated using a bounded depth-first traversal, however the converse is not true. Therefore the depth-first is more general and a better candidate for the compilation technique.

2.2 Body Arguments

In the WAM, compound terms in subgoal arguments are compiled using the same **unify** instructions like in the clause head. The only difference is that the structures are built *bottom-up*, without the **get** instructions, so that dereferencing and trail tests are avoided. The other problems remain, though. Although it is known that the instructions will be executed in **write** mode, each **unify** instruction tests the mode flag. It is also necessary to use temporary variables to build the structure. However, it is known at compile time that the structure will be created in consecutive locations at the heap top, and all pointer offsets are fixed. Therefore, no temporary variables are really necessary to create the structure.

3 Depth-First Method

Our method has the following differences compared to the WAM:

- an explicit distinction between **read** and **write** instructions is made, the compiler generates separate **read** and **write** mode sequences,
- nested structures are unified in a *depth-first* manner,
- no code is duplicated, the execution can switch from the **read** to the **write** sequence and back,
- the body compound arguments are built *top-down* in a *breadth-first* manner, no temporary variables are necessary.

3.1 Head Arguments

While the WAM saves all compound arguments into temporary variables, in the depth-first approach it is enough to save a *pointer* to the argument following a compound one. No temporary variable is needed for the last structure argument. Note for example, that no temporaries are necessary to unify a list containing only constants. The temporaries can be viewed as a return stack whose depth is known at compile time. When the unification of a substructure is finished, the execution proceeds with the next argument whose address is saved in the corresponding temporary variable.

Since the two sequences are now compiled separately, the **get_structure** and **get_list** instructions are slightly modified. For the **write** mode the execution continues in sequence, for **read** mode a branch to a given label is executed. At the end of the **write** mode sequence there is a **branch** instruction that causes a branch to the end of the **read** mode sequence.

Even when the two sequences are split, it is not possible to avoid some tests. Since part of the structure may be unified in **read** and part in **write** mode, branches from one sequence to the other one might be necessary (unless one wants to duplicate the code, but this is exactly what we try to avoid when compiling into native code). The basic observation is as follows: If a structure is unified in **read** mode, all its parent and sibling structures are also unified in **read** mode. Therefore, when the unification of this structure is finished, the execution continues directly in the **read** mode sequence. This is the *upward propagation* of the **read** mode. The compound arguments of a structure, however, might have to be unified in **write** mode if they are matched with a free variable. Hence at the *beginning* of the unification of a structure in **read** mode it is necessary to perform a test and possibly branch into the **write** mode sequence.

If a structure is unified in **write** mode, all its substructures are also unified in **write** mode, i.e. the **write** mode is propagated *downwards*, but its parent structure might be unified in **read** mode. Therefore, at the end of each structure unified in **write** mode the system has to test if a branch back into the **read** mode sequence has to be made. The condition when to jump from the **write** mode sequence back to the **read** sequence is as follows: at the end of every compound term the address of the next argument to unify is obtained from a temporary variable. The tag of this variable stores a flag which decides whether to jump back to the **read** mode sequence or whether to continue with the next instruction.

In **write** mode the whole structure skeleton has to be pushed on the heap at once since the structure arguments are not unified and accessed consecutively. Therefore the **write_structure** pushes the functor on the heap and increments **H** by the arity of the structure. To access structure arguments, both in **read** and **write** mode the **S** register is used. The **read** and **write** mode instruction sequences are very similar except that the former contains the **read** and the latter the **write** variants of the unification instructions.

Below we present an example how to implement our method in the WAM. Note that for bytecode emulators these instructions are too low-level, several instructions could be merged into one to avoid the dispatching overhead. For native code, on the other hand, the generated code must be postprocessed to remove redundant operations.

read_down $\mathbf{X}_i$
This instruction is generated before a compound term which is not the last argument. A reference to the next argument is stored into the temporary.
$\mathbf{X}_i = \mathbf{S + 1;}$

write_down $\mathbf{X}_i$
Like **read_down**, but the mode **write** is marked into the tag of the temporary.
$\mathbf{X}_i = \mathbf{tag\_struct(S + 1);}$

read_structure F
A substructure in **read** mode. If the term pointed to by **S** is a structure pointer and the functor of the structure is equal to F, **S** is set to point to its first argument, else a failure occurs.

```
if (tag(*S) != tag_struct ||
    S = val(*S), *S++ != F)
    Fail;
```

write_structure F
A substructure in **write** mode. A pointer to a new structure skeleton with functor F is stored in the location pointed to by **S** and **S** is set to point to the first argument of the new structure skeleton.

```
*S = tag_struct(H);
H++ = F;
S = H;
H += arity(F);
```

read_up X_i
This instruction is generated at the end of a structure which is itself a (not last) argument of a compound term. The **S** pointer is restored from the temporary.

```
S = Xi;
```

write_up X_i
This instruction is generated at the end of a structure in **write** mode which is not the last argument. The **S** pointer is restored from the temporary.

```
S = val(Xi);
```

read_list
Corresponds to **read_structure** for lists.

```
if (tag(*S) == tag_list)
    S = val(*S);
else
    Fail;
```

write_list
Corresponds to **write_structure** for lists.

```
*S = tag_list(H);
S = H;
H += 2;
```

read_test LabW
This instruction precedes a substructure in **read** mode. **S** is dereferenced. If the result of dereferencing is a free variable, it is trailed if necessary and a jump to **LabW** is executed.

```
S = Deref(S);
if (tag(S) == tag_ref) {
    trail(S);
    P = LabW;
}
```

write_test X_i, LabR
This instruction is generated after a **write_up** instruction. If the tag of X_i is not **write**, a jump to LabR is executed.

```
if (tag(Xi) != tag_struct)
    P = LabR;
```

read_constant C
A constant argument of a compound term in **read** mode. The contents of **S** is dereferenced and unified with **C**. After a successful unification **S** is incremented.

```
if (Unify(*Deref(S), C))
    S++;
else
    Fail;
```

write_constant C
A constant argument of a compound term in **write** mode. **C** is stored into the location pointed to by **S** and **S** is incremented.
***S++ = C;**

3.1.1 Compilation Example

Note that the WAM uses less abstract instructions, however they are more complicated than ours. When expanded, the WAM code is always longer.

f(g(X), b)

Our Code

```
        get_structure   Ai, LabR
        write_down      X1
W1:     write_structure g/1
        write_variable  Y1
        write_up        X1
        write_test      X1, R1
        write_constant  b
        branch          Ends
LabR:   read_down       X1
        read_test       W1
        read_structure  g/1
        read_variable   Y1
        read_up         X1
R1:     read_constant   b
Ends:
```

WAM Code

```
get_structure   Ai
unify_variable  X1
unify_constant  b
get_structure   g/1, X1
unify_variable  Y1
```

As we can see, the code generated for **read** and **write** mode is very similar, temporary variables are the same in both sequences and the correspondence of labels for branches between the two sequences is straightforward.

3.2 Body Arguments

Unlike head arguments, the compound terms to be constructed in the clause body are known at compile time, they are going to be pushed onto consecutive heap locations. Therefore, when the compound term is created top-down, in a breadth-first manner, one pointer is completely sufficient to build it, no temporary variables are necessary.

The instructions work as follows: the generation starts with the topmost structure, first the functor is pushed at the heap top using the **H** pointer, then the structure arguments are pushed, followed by the next substructure etc. Whenever a compound subargument is encountered, a structure pointer to it is pushed on the heap and then the next argument is built. Thus each new structure is put at the end of a queue and built when it is at its beginning.

put_structure F, A_i This instruction corresponds to the beginning of a compound argument of a body subgoal. A functor is pushed on the heap, a structure pointer to it is stored into A_i.

A_i = tag_struct(H);

***H++ = F;**

push_constant C The constant C is pushed on the heap. Note that C can also be a functor of a compound term.

***H++ = C;**

push_structure Offset This instruction corresponds to a compound subargument. A structure pointer to **H + Offset** is pushed on the heap top.

***H = tag_struct(H + Offset);**

H++;

push_list Offset This instruction corresponds to a compound subargument which is a list.

***H = tag_list(H + Offset);**

H++;

An example of a body goal compilation:

test :- do(parse(s(np, vp), [birds, fly], [])).

Our Code		**WAM Code** (from [7])	
put_structure	parse/3, A_1	put_structure	s/2, X_2
push_structure	3	unify_constant	np
push_list	5	unify_constant	vp
push_constant	[]	put_list	X_4
push_constant	s/2	unify_constant	fly

push_constant	**np**	**unify_nil**	
push_constant	**vp**	**put_list**	$\mathbf{X}_3$
push_constant	**birds**	**unify_constant**	**birds**
push_list	**1**	**unify_value**	$\mathbf{X}_4$
push_constant	**fly**	**put_structure**	**parse/3,** $\mathbf{A}_1$
push_constant	[]	**unify_value**	$\mathbf{X}_2$
execute	**do/1**	**unify_value**	$\mathbf{X}_3$
		unify_nil	
		execute	**do/1**

It can be seen that to achieve the same effect the WAM needs 3 temporary variables and more code than our method.

4 Optimizations

The above presented methods are the basic schemata only, there are many optimizations to be applied on them.

- In the sequence of **up**, **down** and **test** abstract instructions in the generated native code usually several machine instructions can be omitted, which is easy to do in a peephole optimizer.

- The **unify_local_value** WAM instruction has only its **write** counterpart, in the **read** sequence the normal **read_value** instruction is used.

- When the head structure contains void variables, it is often possible to omit part of the **read** sequence, provided that no nonvoid argument follows the void one(s). For example, the term

  [_|_]

  can be simply compiled into

```
          get_list      A_i, Ends
          write_void    2
 Ends:
```

- Ground compound terms in the clause body should not be compiled into a sequence of instructions - even with a *structure copying* method they can be shared. The compound term can be created once at compile time and in the clause body the **put_constant** instruction just puts a corresponding structure or list pointer to it.

  Similarly, ground compound terms in the head should be constructed at compile time and treated like constants in the **write** sequence, e.g. **f(g(a), g(b))** is compiled into

```
          get_structure    f/2, A_i, LR
          write_constant   < g(a) >
          write_constant   < g(b) >
          branch           Ends
 LR:      read_down        X_1
          ...
 Ends:
```

- If the compound argument is the indexed one, a branch directly to the **read** sequence can be executed:

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```

	Our Code			WAM Code	
La:	switch_on_type	A_1, list:Ll, nil:Ln, default:fail		switch_on_type	A_1, list:Ll, nil:Ln, default:fail
Lv:	get_nil	A_1	Lv:	get_nil	A_1
Ln:	get_value	A_2, A_3	Ln:	get_value	A_2, A_3
	proceed			proceed	
	get_list	A_1, Ll	Ll:	get_list	A_1
	write_variable	X_4		unify_variable	X_4
	write_variable	A_1		unify_variable	A_1
	branch	E1		get_list	A_3
Ll:	read_variable	X_4		unify_value	X_4
	read_variable	A_1		unify_variable	A_3
E1:	get_list	A_3, Lr		execute	append/3
	write_value	X_4			
	write_variable	A_3			
	branch	La			
Lr:	read_value	X_4			
	read_variable	A_3			
	branch	La			

- When more information is available about the instantiation of the caller arguments, e.g. from abstract interpretation, the compiler can omit certain instructions and regroup the others, so that a simple peephole optimization yields optimal native code. For example, if the clause

  ```
  p(g(f(X), h(X))).
  ```

  is always called with an instantiated argument, but the arguments of **g/2** are known to be unbound, a mixture of **read** and **write** sequence can be generated, the compiler can replace some of the **read** instructions by **write** ones:

```
g(f(X),
  h(X))
```

	Default			Mixture	
	get_structure	g/2, A_i, LR		get_structure	g/2, A_i, L1
	write_down	X_1	L1:	read_down	X_1
W1:	write_structure	f/1		read_test	L_2
	write_variable	X_2		write_structure	f/1
	write_up	X_1		write_variable	X_2
	write_test	X_1, R1		read_up	X_1
W2:	write_structure	h/1		read_test	X_1, L_3
	write_value	X_2	L3:	write_structure	h/1
	branch	Ends		write_value	X_2
LR:	read_down	X_1			
	read_test	W1			
	read_structure	f/1			
	read_value	X_2			
	read_up	X_1			
R1:	read_test	W2			
	read_structure	h/1			
	read_value	X_2			
Ends:					

Note also that the depth-first approach is even more flexible - the compiler might decide to compile e.g. the last argument depth-first and the other ones breadth-first etc.

5 Discussion

Optimizing the unification of compound terms in Prolog is a challenging task. One of the problems of Prolog efficiency is exactly this - inefficient handling of structures. Often a compound term is created before calling a goal only to decompose it in the called procedure, which is a severe overhead. The *structure sharing* systems have an advantage here, because the structures do not have to be explicitly created. On the other hand, structure sharing has other disadvantages like poor locality of references and creating long reference chains, so that the only way currently being followed is to optimize the structure copying mechanism.

Despite the problems in the WAM handling of compound terms, there has been very little work published about alternative approaches[1]. Current emulated systems seem to use the WAM code since it is compact, but the emulator contains separate versions of the **unify** instructions for **read** and **write** mode. The mode flag is not used because each concerned instruction knows to which version of the next instruction it should continue.

Native code systems often use a scheme similar to [6], where explicit **read** and **write** sequences are generated, e.g. for **f(g(a), g(b))**

	get_structure	**f/2**, $\mathbf{A}_i$, **LR**
	write_variable	$\mathbf{X}_1$
	write_variable	$\mathbf{X}_2$
	branch	**S1**
LR:	**read_variable**	$\mathbf{X}_1$
	read_variable	$\mathbf{X}_2$
S1:	**get_structure**	**g/1**, $\mathbf{X}_1$
	...	

In this way the use of the mode flag is not necessary, however nested structures are still unified separately from the main term and so all the problems with the **write** mode remain. If the last argument is compound, it can be unified directly, without a **write_variable** instruction and so the **write** mode can be propagated to it. Propagating the **write** mode to other substructures is obviously difficult with *breadth-first* traversal, but it is a trivial task when the *depth-first* approach is used.

Turk [6] describes an approach where the **write** mode is propagated using a branch into the middle of the **get** instruction, to omit dereferencing and testing. The **write** mode can thus be directly propagated only to the first compound

[1] Our work presented in this paper is based on [2].

argument, its first compound subargument etc. The following arguments have to push a free variable on the heap and the **get** instruction has to dereference and test it like in the basic WAM. In our method the number of branches depends on the actual data, how often it is necessary to switch between the two sequences; the **write** mode is propagated everywhere.

Recently, Van Roy [5] described a lower-level abstract language for the unification which generates separate sequences for the **read** and **write** mode. The **write** mode sequences use a breadth-first approach similar to ours for body terms, but the whole skeleton is pushed at once, so that some of the **push** instructions need two offsets as arguments, which is less efficient for bytecode and for some machines. The **read** mode sequences use depth-first traversal, but the code is not shared between the two sequences, except for the last argument, and so the **write** mode instructions appear also inside the **read** mode sequence. In this way, Van Roy's **write** mode sequence does not contain any redundant operations, however for the expense that this code cannot be shared. For complex structures the size of the generated code grows exponentially. When we use the **push** instructions instead of the **write** ones, except for the last argument, and generate an additional **write** sequence for every label in **read_test** using the **push** instructions, we obtain a method which is very similar to the Van Roy's one.

Our methods have the following advantages over the WAM or the above derivatives:

- Since we are using the depth-first approach, our method for head unification needs less temporary variables than the WAM for wide or shallow structures.
- We generate explicit **read** and **write** sequences and for every source item (including parentheses) there is at most one **read** and one **write** instruction. The code size is proportional to the term size, no code is duplicated.
- The **write** mode is automatically propagated to all head substructures, no intermediate free variables are pushed on the heap in order to save and restore the mode information. If it is necessary to save the mode information, most often this is done in a register. Our method thus performs less memory accesses than the pure WAM.
- Our method uses the **H** and **S** register in a uniform way both in **read** and **write** instructions. It is therefore possible to jump from one sequence tothe other one, or generate directly mixed code.
- Our method can be used both for depth-first and breadth-first structure traversal. After performing the global analysis of the program, the compiler can generate the appropriate code and omit sequences or parts of them which are not necessary.

- The code for constructing compound terms in the body is optimal. For general head terms the length of our code is (after postprocessing) minimal.

The drawbacks of our method concern the following cases:

- For left-balanced head structures like **f(g(h(a), b), c)** where each structure has only one compound argument which is not the last one, our method needs more temporary variables than the WAM. Prolog programmers normally prefer right-balanced structures which can be processed using tail-recursive loops.
- When a structure is unified in **read** mode and all its arguments are unified in **write** mode, two branches are executed for each argument. In this case, either breadth-first approach can be used, or the compiler can generate a mixture of **read** and **write** sequences.

Acknowledgements

The author would like to thank to Dominique Henry de Villeneuve, Alexander Herold, Jacques Noye, Bruno Poterie and Joachim Schimpf for fruitful discussions about this topic and for commenting previous versions of this paper and to the referees of the submitted paper for interesting comments about up-to-date systems. Many of the ideas presented here are probably already part of Prolog folklore and so the author would also like to acknowledge all people that have contributed to their development.

Bibliography

[1] John Gabriel, Tim Lindholm, E. L. Lusk, and R. A. Overbeek. A tutorial on the warren abstract machine for computational logic. Technical Report ANL-84-84, Argonne National Laboratory, 1984.

[2] Micha Meier. The compilation of compound term unification in Prolog. Internal Report IR-LP/lpp1, ECRC, May 1987.

[3] Micha Meier. Analysis of Prolog procedures for indexing purposes. In ICOT, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 800–807, Tokyo, November 1988.

[4] Micha Meier, Abderrahmane Aggoun, David Chan, Pierre Dufresne, Reinhard Enders, Dominique Henry de Villeneuve, Alexander Herold, Philip Kay, Bruno Perez, Emmanuel van Rossum, and Joachim Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the 11th World Computer Congress IFIP'89*, pages 1127–1132, San Francisco, August 1989.

[5] Peter Van Roy. An intermediate language to support Prolog's unification. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the NACLP'89*, pages 1148–1164, Cleveland, October 1989.

[6] Andrew K. Turk. Compiler optimizations for the WAM. In *Third International Conference on Logic Programming*, pages 657–662, London, July 1986.

[7] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI, October 1983.

A Program to Compile Compound Terms

```
% Compile a head compound term
head(Term) :-
    functor(Term, F, A),
    compile_args(Term, 1, A, 1, Code,
         [branch(lab(_)), label(LR)|ReadCode]),
    read_seq(Code, ReadCode, []),
    pwrite([get_structure(F/A, reg(i), lab(LR))|Code]).

% Compile a compound subgoal argument
body(Term) :-
    functor(Term, F, A),
    A1 is A + 1,
    compile_body([Term|Cont], Cont, A1, [_|Code], []),
    pwrite([put_structure(F/A, reg(i))|Code]).

% Write sequence for the arguments of a compound term
compile_args(Term, A, A, Reg) -->
    {arg(A, Term, Arg)},
    compile_arg(Arg, Reg, last).
compile_args(Term, I, A, Reg) -->
    {I < A, arg(I, Term, Arg), I1 is I + 1},
    compile_arg(Arg, Reg, notlast),
    compile_args(Term, I1, A, Reg).

% Generate the write sequence for one argument
compile_arg(Struct, Reg, last) -->
    {compound(Struct), functor(Struct, F, A)},
    [label(_), write_structure(F/A)],
    compile_args(Struct, 1, A, Reg).
compile_arg(Struct, Reg, notlast) -->
    {compound(Struct), functor(Struct, F, A), Reg1 is Reg+1},
    [write_down(reg(Reg)), label(_), write_structure(F/A)],
    compile_args(Struct, 1, A, Reg1),
    [write_up(reg(Reg)), write_test(lab(_))].
compile_arg(Const, _, _) -->
    {atomic(Const)},
    [write_constant(Const)].

% Generate the read sequence and fill in the labels
read_seq([branch(lab(L))|_]) -->
```

```
    [label(L)].
read_seq([write_down(R)|T]) -->
    [read_down(R)],
    read_seq(T).
read_seq([label(L), write_structure(S)|T]) -->
    [read_test(lab(L)), read_structure(S)],
    read_seq(T).
read_seq([write_up(R), write_test(lab(L))|T]) -->
    [read_up(R), label(L)],
    read_seq(T).
read_seq([write_constant(C)|T]) -->
    [read_constant(C)],
    read_seq(T).

% Compile a queue of body structures
compile_body([], [], _) --> {true}.
compile_body([Struct|Rest], Cont, Off) -->
    {functor(Struct, F, A), Off1 is Off - 1},
    [push_constant(F/A)],
    compile_struct(Struct, 1, A, Off1, NewOff, Cont, NewCont),
    compile_body(Rest, NewCont, NewOff).

% Compile one body structure
compile_struct(Struct, A, A, Off, NewOff, Cont, NewCont) -->
    {arg(A, Struct, Arg)},
    compile_body_arg(Arg, Off, NewOff, Cont, NewCont).
compile_struct(Struct, I, A, Off, NewOff, Cont, NewCont) -->
    {I < A, arg(I, Struct, Arg), I1 is I + 1},
    compile_body_arg(Arg, Off, NO, Cont, NC),
    compile_struct(Struct, I1, A, NO, NewOff, NC, NewCont).

% Compile one argument of a body structure
compile_body_arg(Const, Off, NewOff, C, C) -->
    {atomic(Const), NewOff is Off - 1},
    [push_constant(Const)].
compile_body_arg(Struct, Off, NewOff, [Struct|C], C) -->
    {compound(Struct), functor(Struct, _, A),
        NewOff is Off + A},
    [push_structure(Off)].

% Print the generated code
pwrite([]).
pwrite([label(Lab)|Rest]) :-
    write(Lab),
    write(:),
    pwrite(Rest).
pwrite([Instr|Rest]) :-
```

```
    put(9),
    functor(Instr, F, A),
    write(F),
    name(F, LS),
    length(LS, Length),
    tab(20-Length),
    writeargs(Instr, 1, A),
    nl,
    pwrite(Rest).

writeargs(Instr, A, A) :-
    arg(A, Instr, Arg),
    writearg(Arg).
writeargs(Instr, I, A) :-
    I < A,
    arg(I, Instr, Arg),
    writearg(Arg),
    write(', '),
    I1 is I + 1,
    writeargs(Instr, I1, A).

writearg(lab(L)) :-
    write(L).
writearg(reg(R)) :-
    write('X'),
    write(R).
writearg(Arg) :-
    write(Arg).
```