Chapter 5

Compiling Kernel Prolog to BAM Code

1. Introduction

The previous chapters described the conversion of standard Prolog to kernel Prolog and the optimizing kernel transformations. This chapter shows how the optimized kernel Prolog is compiled to BAM code. The compilation to BAM is performed in two steps for each predicate. In the first step, the control instructions that make up the framework of the predicate are compiled by the predicate compiler. This includes compiling the deterministic case statements into conditional branches and the disjunctions with choice point instructions.

In the second step, the clauses that make up the body of the predicate are compiled by the clause compiler. The clause compiler uses two primitives, the goal compiler and the unification compiler, to compile goals and explicit unifications. The clause compiler also does register allocation, entry specialization (replacing built-in predicates by faster entry points), and performs the write-once transformation (for fast trailing), and the dereference chain transformation (to maintain consistency with the dataflow analysis). These transformations are explained in detail in the sections below.

2. The predicate compiler

In the kernel transformation stage (Chapter 4), determinism extraction attempts to convert each predicate into a series of nested case statements. This is not always successful; sometimes the case statements still retain disjunctions (OR choices) that could not be converted into deterministic code. The predicate compiler compiles both the case statements and the disjunctions into BAM code. The case statements are compiled into conditional branches. The disjunctions are compiled into choice point instructions. The predicate compiler uses two primitives, the determinism compiler and the disjunction compiler, to compile the predicate's case statements and disjunctions.

2.1. The determinism compiler

Compiling a kernel Prolog predicate into deterministic BAM code is done in two steps. First, the determinism transformation (a kernel Prolog transformation, Chapter 4) converts a kernel Prolog predicate into a series of nested case statements. Then the determinism compiler compiles the nested case statements into BAM code. A case statement may contain any test set, and each test set is mapped to a conditional branch. The test sets and their corresponding conditional branches are given in Table 4.11.

2.2. The disjunction compiler

A disjunction (an OR formula) is a list of clauses that encapsulates a choice. The first clause is executed the first time the disjunction is encountered. The remaining clauses are executed in order on backtracking—each time backtracking returns to the disjunction the next clause is tried. This is implemented by code which generates choice points. A choice point encapsulates the state of the abstract machine at the time it is created. Backtracking restores machine state from a choice point to let execution continue from the point at which the choice point was created.

Creating and restoring machine state in choice points is time-consuming. To minimize the size of the choice points (and hence the time required to create them), the choice point management instructions in the BAM are streamlined to perform the least amount of data movement. They save only those registers that are needed in the clauses of the disjunction after the first, and for each clause of the disjunction they restore only those registers that are needed in that clause. Argument registers are restored in the clause itself and not in the `fail` instruction. Therefore the size of the choice point does not have to be stored in the choice point and decoded in the `fail` instruction. A disadvantage is a slightly larger code size.† Consider the following kernel Prolog for a predicate P with n clauses:

Head :- (C_1 ; C_2 ; ... ; C_n ; fail).

A single choice point is created for each invocation of P. The set of registers saved in the choice point is the set of all head arguments that are used in clauses after the first, i.e. C_2 through C_n. Arguments that

† This is less of a problem in the VLSI-BAM since the instruction reorderer merges pairs of single-word loads into double-word loads.

occur only in clause C_1 do not have to be stored in the choice point. The set of registers that is restored for each clause is the set of arguments used in that clause.

Before creating the choice point, the compiler dereferences those arguments that it can deduce will be dereferenced later. This avoids dereferencing the same argument more than once. The set of arguments to be dereferenced is derived by checking the type formula corresponding to each goal in the body of the predicate's definition, and noting whether its arguments have to be dereferenced. For example, arithmetic operations and relational tests are goals that require their arguments to be dereferenced.

To illustrate the compilation scheme, consider the following predicate:

```
p(A,B,C,D) :- ( a(A)
              ; c(C)
              ; d(D)
              ; fail
              ).
```

It is compiled as:

```
procedure(p/4).
        choice(1/3,[2,3],l(p/4,2)).   ; Save registers r(2) and r(3).
        jump(a/1).
label(l(p/4,2)).
        choice(2/3,[2,no],l(p/4,3)).  ; Restore only register r(2).
        move(r(2),r(0)).
        jump(c/1).
label(l(p/4,3)).
        choice(3/3,[no,3],fail).      ; Restore only register r(3).
        move(r(3),r(0)).
        jump(d/1).
```

The choice instructions do all the choice point manipulation: `choice(1/3,...)` creates the choice point, `choice(2/3,...)` modifies the address to return to on backtracking, and `choice(3/3,...)` removes the choice point. Register `r(0)` is not saved in the choice point because it is not needed in clauses beyond the first. The second and third clauses restore only the registers they need. Register `r(1)` is not saved because it is not needed at all.

Each choice instruction contains a list of the registers that it used. The length of the list is the same for all choice instructions in a predicate. For choices after the first, the atom `no` is put in the positions of registers that do not have to be restored. For example, the list `[0,no,5]` means that registers `r(0)` and `r(5)` are restored from the first and third locations in the choice point, and the second location is not accessed.

In this example a further optimization can be done by merging the move instructions with the choice instructions, i.e.:

```
choice(3/3,[no,3],fail).
move(r(3),r(0)).
```

becomes:

```
choice(3/3,[no,0],fail).
```

This is possible because the value loaded in a register is determined by its position in the list, not by its number, and because register `r(3)` is only used to load `r(0)`.

Figure 5.1 – Structure of the clause compiler

3. The clause compiler

The clause compiler converts a clause from kernel Prolog form (with type annotations) to BAM code. The structure of the clause compiler is given in Figure 5.1. After compiling the goals in the body there are two intermediate results: (1) BAM code in which variables have not yet been allocated to registers (*skeleton* code) and (2) a variable occurrence list (the *varlist*), that contains all unallocated variables in the skeleton code. The final BAM code is obtained by passing the varlist to the register allocator.

Each goal in the clause body is compiled in four steps. First, three transformations are performed on the goal: entry specialization, the write-once transformation, and the dereference chain transformation. Then the goal is compiled into BAM code by one of two routines, the unification compiler or the goal compiler, depending on whether the goal is a unification or not.

These are the important blocks in the clause compiler:

(1) **The goal compiler**. Its main task is to handle argument passing. Because of the interaction between the different kinds of unbound variables, initialized and uninitialized, this results in a case analysis. In addition, the goal compiler compiles in-line some built-in predicates and the dummy predicates that were created in the transformation to kernel Prolog.

(2) **The unification compiler**. Its task is, given a type, to compile an explicit unification into the simplest possible code.

(3) **The register allocator**. Its task is to allocate variables to registers in such a way that the number of superfluous move instructions is minimized. It uses a data structure called the *varlist* which is generated by the clause body compiler.

(4) **Entry specialization**. This attempts to replace each goal in the clause by a faster entry point, depending on the types known at the call.

(5) **Write-once transformation**. This transformation is part of a technique for reducing the overhead of trailing.

(6) **Dereference chain transformation**. This transformation is necessary to keep the dataflow analysis and the clause compiler consistent.

The following sections give more details about these each of these blocks. First, an example of a clause compilation is given, with emphasis on the skeleton code, the varlist, and a specification of the register allocator. This is followed by discussions of the goal compiler, the unification compiler, entry specialization, the write-once transformation, and the dereference chain transformation.

3.1. Overview of clause compilation and register allocation

This section gives an example of how a clause is compiled. Consider the following clause with no types:

```
a(A,B) :- b(A,C), d(C,B).
```

Compilation of this clause proceeds in three steps: First the kernel Prolog is compiled to BAM code and a variable occurrence list, or *varlist*. In this example, most of the work in this step is done in the goal compiler. The resulting BAM code is referred to as *skeleton* code since variables have not yet been allocated to registers. The varlist is derived from the skeleton code and contains the list of variables and registers in it. Second, the register allocator uses the varlist to allocate variables to registers. Third, after all predicates and all clauses are compiled, the BAM optimization stage improves the code (Chapter 6). The skeleton code for this clause is:

```
allocate(X).           ; Create an environment (its size is still unknown).
move(r(0),A).          ; Load the head arguments into variables A and B.
move(r(1),B).
move(tvar^r(h),C).     ; Create an unbound variable and put it in C and D.
move(tvar^r(h),D).     ; C may exist beyond a call, D exists between calls.
pragma(push(variable)).
push(D,r(h),1).
move(A,r(0)).          ; Load the parameters of the first call.
move(D,r(1)).
call(b/2).
pragma(tag(C,tvar)).   ; C has an extra link, with a tvar tag.
move([C],r(0)).        ; Extra indirection to remove the extra link.
move(B,r(1)).
call(d/2).
deallocate(X).         ; No last call optimization in the skeleton code.
return.
```

The varlist for this clause is:

```
[pref,r(0),A,           ; Corresponds to move(r(0),A).
 pref,r(1),B,
 C,pref,C,D,D,          ; Corresponds to the unbound variable in C and D.
 pref,A,r(0),
 pref,D,r(1),
 fence,                 ; Corresponds to call(b/2).
 C,r(0),
 pref,B,r(1),
 fence]                 ; Corresponds to call(d/2).
```

3.1.1. Construction of the varlist

The varlist is constructed to satisfy these conditions:

(1) The only contents of the varlist are unbound variables, temporary registers, and the atoms `fence` and `pref`.

(2) The order of variable occurrences is the same in the skeleton code and the varlist.

(3) The atom `fence` is inserted as a marker at each point where temporary variables do not survive. This corresponds to each `call(..)` instruction in the skeleton code.

(4) Two variables that are preferably allocated to the same register are preceded by the atom `pref` and called a *pref pair*. A pref pair is created when allocating the variables to the same register allows an instruction to be removed. For example, the `move(A,r(0))` instruction can be removed if the variable A is allocated to register `r(0)`.

(5) A variable occurs exactly once in the varlist if and only if it occurs exactly once in the skeleton code. Such a variable is called a void variable. An instruction containing a void variable may be removed.

(6) A variable occurs more than once in the varlist if and only if it occurs more than once in the skeleton code.

3.1.2. The register allocator

The register allocator assigns a register to each variable in the varlist such that there are no conflicts, i.e. a single register never holds two values at the same time. The allocator also calculates the size of the environment (the number of permanent registers) for the `allocate` and `deallocate` instructions. The algorithm is defined in Figure 5.2. It assumes that variables are represented as logical variables, i.e.

procedure register_allocator(VL : varlist);
var V_{void}, V_{temp}, V_{pref}, V_{perm} : set of variable;
begin
    V_{void} := { variable Y | Y occurs exactly once in VL };
    $\forall X \in V_{void}$ **do** Allocate each X to `r(void)`;
    V_{perm} := { variable Y | The sequence [Y, ..., `fence`, ..., Y] occurs in VL };
    $\forall X \in V_{perm}$ **do** Allocate each X to a different `p(I)`;
    Environment size := number of elements in V_{perm};
    V_{temp} := { variable Y | Y occurs more than once in VL };
    V_{pref} := prefer(VL);
    while $V_{temp} \neq \emptyset$ **do begin**
        while $\exists X \in V_{pref}$: X is allocatable to `r(I)` without conflict **do begin**
            Allocate X to its preferred register `r(I)`;
            V_{pref} := $V_{pref} - \{X\}$;
            V_{temp} := $V_{temp} - \{X\}$;
            V_{pref} := prefer(VL)
        end;
        if $\exists X \in V_{temp}$ **then begin**
            Allocate X to the lowest `r(I)` possible without conflict;
            V_{pref} := $V_{pref} - \{X\}$;
            V_{temp} := $V_{temp} - \{X\}$;
            V_{pref} := prefer(VL)
        end
    end
end;

function prefer(VL : varlist) : set of variable;
begin
    return { variable Y | The sequence [`pref`, Y, _] or [`pref`, _, Y] occurs in VL }
end;

Figure 5.2 – The register allocator

that allocating a variable to a register binds that variable in all sets that contain it. It assumes that there are an infinite number of temporary and permanent registers. It uses the following correspondence between variable lifetimes and registers:

(1) A variable that occurs exactly once is allocated to `r(void)`.

(2) A variable occurring on both sides of a `fence` marker (it *crosses* a fence) is allocated to a permanent register `p(I)` (a location in the environment).

(3) A variable that does not cross a fence and that occurs more than once is allocated to a temporary register `r(I)`.

The algorithm is independent of the write-once transformation and the dereference chain transformation.

This is possible because the clause compiler is careful to feed the allocator a varlist that takes the two transformations into account.

In the example of the previous section, the allocator assigns the following values to the variables:

```
A = r(0)
B = p(0)
C = p(1)
D = r(1)
X = 2
```

Since both B and C cross a fence, they are allocated to permanent registers. Both A and D are allocated to their preferred registers. The number of permanent variables, X, is 2.

3.1.3. The final result

The final BAM code output by the compiler after all transformations and optimizations (including the BAM transformations of chapter 6) is:

```
allocate(2).            ; Allocate space for two permanent variables.
move(r(1),p(0)).
move(tvar^r(h),r(1)).   ; Create an unbound variable and put it in r(1) and p(1).
move(r(1),p(1)).
pragma(push(variable)).
push(r(1),r(h),1).
call(b/2).
pragma(tag(p(1),tvar)).
move([p(1)],r(0)).      ; Indirection due to dereference chain transformation.
move(p(0),r(1)).
deallocate(2).
jump(d/2).              ; Last call optimization converts 'call' to 'jump'.
```

3.2. The goal compiler

Given a goal and type information about the goal, this module sets up the arguments to call the goal, does the call, and sets up the return arguments. The main task of the goal compiler is to handle the complexities that arise when supporting combinations of uninitialized and initialized parameters. The following situations are also handled:

(1) **Duplicate variables**. An uninitialized variable that occurs twice in a goal must be initialized before calling the goal.

(2) **Uninitialized register variables**. Passing arguments as uninitialized register variables requires some care. These variables are not passed into a predicate, but are outputs returned in registers.

(3) **Dummy predicates**. Several compiler transformations create new predicates as part of the transformation. These predicates are only called once, so they are compiled in-line.

(4) **Built-in predicates**. Some built-in predicates are translated into in-line code (Table 5.5).

```
function compile_goal(G : goal; F : formula; V_sf : set) : return (Code : list; F_out : formula; V_sf,out : set);
var   V_uninit , V_init : set of variable;
      Initcode , Precode , Call , Postcode : list of instruction;
      A : term;
      g_i , r_i : {ini, mem, reg};
      i : integer;
begin
      /* Initialize all uninitialized variables that are duplicated */
      V_uninit := { X | F implies (uninit_mem(X) or uninit_reg(X)) };
      V_init := ((vars(G) – V_sf) ∪ V_uninit) ∩ dups(G);  /* Table 4.6 */
      Initcode := list of ( ∀ X ∈ V_init : Code to initialize the variable X);

      /* Pass arguments to the goal and clean up afterwards */
      Precode := [ ];
      Postcode := [ ];
      for i := 1 to arity(G) do begin
            A := (argument i of goal G);
            g_i := given_flag(A, F, V_sf);   /* Table 5.1 */
            r_i := require_flag(A, G);   /* Table 5.2 */
            Append precode[ g_i , r_i ] to Precode;   /* Table 5.3 */
            Append postcode[ g_i , r_i ] to Postcode    /* Table 5.4 */
      end;

      /* Call the goal */
      if (G can be expanded in-line) then
            Call := (in-line expansion of G)  /* Table 5.5 */
      else if (G is a dummy predicate) then
            Call := (in-line compilation of G's definition)
      else if (G does not alter temporary registers) then
            Call := (a simple_call instruction for G)  /* Table 3.7 */
      else
            Call := (a call instruction for G);
      Code := append(Initcode, Precode, Call, Postcode)
end;
```

Figure 5.3 – The goal compiler

The function compile_goal(G, F, V_{sf}) defines the goal compiler (Figure 5.3). Its inputs are the goal (G), a type formula (F), and the set of variables that have a value on input (V_{sf}). Its outputs are a list of BAM instructions (*Code*), the type formula true on output (F_{out}), and the set of variables that have a value on output ($V_{sf,out}$).

Each goal has three type formulas associated with it: a *Require* type, a *Before* type, and an *After* type. These types are optionally given by programmer input and are supplemented by dataflow analysis. The compiler maintains a table of these types for all predicates including built-ins and internals. The Require type gives the types that the arguments being passed to the goal must have, i.e. the goal compiler is required to make them true in all cases. The Before type gives the types that are true before the call. The After type gives the types that are true after the call returns. No special action is needed by the goal compiler to ensure the validity of the Before and After types.

Compiling a goal is made more complex because the kind of argument needed by the goal may not be the same as the one that is given to it. The goal's *Given* type (which is valid before the goal and given by F in Figure 5.3) must be reconciled with the goal's Require type. The most common Require and Given types are the three varieties of unbound variables: uninitialized memory and register variables and initialized variables. This requires a case analysis with 3×3 cases for each argument of the goal to properly match the Require and Given types.

Table 5.1 – Calculating the Given flag of an argument

Condition on argument A	g_i
nonvar(A)	`ini`
var(A) $\wedge$ (F implies uninit_mem(A))	`mem`
var(A) $\wedge$ (($A \notin V_{sf}$) $\vee$ (F implies uninit_reg(A)))	`reg`
var(A) $\wedge$ ($A \in V_{sf}$)	`ini`

Table 5.2 – Calculating the Require flag of an argument

Condition on argument A	r_i
require(G) implies uninit_mem(A)	`mem`
require(G) implies uninit_reg(A)	`reg`
otherwise	`ini`

Table 5.3 – Calculating the precode from the flags

g_i	r_i	precode[g_i , r_i]
reg	reg	[]
mem	reg	[]
ini	reg	[]
reg	mem	[move(tvar^r(h),B),adda(r(h),1,r(h))]
mem	mem	[]
ini	mem	[move(tvar^r(h),B),adda(r(h),1,r(h))]
reg	ini	[move(tvar^r(h),B),push(B,r(h),1)]
mem	ini	[move(A,[A]),move(A,B)]
ini	ini	[]

Table 5.4 – Calculating the postcode from the flags

g_i	r_i	postcode[g_i , r_i]
reg	reg	[]
mem	reg	[move(B,[A])]
ini	reg	unify(A,B)
reg	mem	[move(B,A)]
mem	mem	[]
ini	mem	unify(A,B)
reg	ini	[move(B,A)]
mem	ini	[]
ini	ini	[]

Require and Given flags r_i and g_i (with values in {ini, mem, reg}) are associated with each goal argument for the Require and Given types. Tables 5.1 and 5.2 define how the Require and Given flags are calculated. The function require(G) in Table 5.2 is a defined predicate in the compiler that returns the Require type for any goal. It knows all about built-in and internal predicates and the results of dataflow analysis.

Duplicate arguments (e.g. A in the call p(A, A)) are treated specially. An argument that is duplicate cannot be uninitialized—it occurs in more than one place, so it is not unaliased any more. The goal compiler initializes these arguments before doing the case analysis.

Table 5.3 gives the precode, i.e. the code that is generated before the call to set up, and Table 5.4 gives the postcode, i.e. the code that cleans up after the call. To enforce the Require type, in seven of the nine cases a different argument B is passed to the call instead of the goal's original argument A. For example, if the Given flag is mem and the Require type is reg, then the compiler must create a new variable B of type uninit_reg(B) to pass to the goal. After the goal returns, the original argument A and the returned argument B are unified together. The new variable B is created for all combinations of Given and

Require flags except (`reg`, `reg`) and (`mem`, `mem`). In these two cases no precode or postcode is needed.

To simplify the presentation, Figure 5.3 only does part of what the algorithm implemented in the compiler does. The definition of compile_goal in the figure only handles Require and Given types that are all uninitialized variables. The actual algorithm handles any types. The type formula F and the variable set V_{sf} are updated continuously during the execution of compile_goal. A variable occurrence list is calculated for the register allocator. The actual algorithm handles 12 cases for parameter passing instead of 9—as an optimization, two varieties of Given uninitialized register types are recognized.

Table 5.5 – BAM expansion of internal built-ins

Kernel Prolog	BAM instruction
`'$cut_load'(X)`	`move(r(b),X)`
`'$cut'(X)`	`cut(X)`
`'$name_arity'(X,'.',2)`	`test(ne,tlst,X,fail)`
`'$name_arity'(X,Na,Ar)`	`equal([X],tatm^(Na/Ar),fail)`
`'$name_arity'(X,Na,0)`	`equal(X,tatm^Na,fail)`
`'$test'(X,Types)`	(a sequence of `test` instructions)
`'$equal'(X,Y)`	`equal(X,Y,fail)`
`'$add'(A,B,C)`	`add(A,B,C)`
`'$sub'(A,B,C)`	`sub(A,B,C)`
`'$mod'(A,B,C)`	`mod(A,B,C)`
`'$mul'(A,B,C)`	`mul(A,B,C)`
`'$div'(A,B,C)`	`div(A,B,C)`
`'$and'(A,B,C)`	`and(A,B,C)`
`'$or'(A,B,C)`	`or(A,B,C)`
`'$xor'(A,B,C)`	`xor(A,B,C)`
`'$sll'(A,B,C)`	`sll(A,B,C)`
`'$sra'(A,B,C)`	`sra(A,B,C)`
`'$not'(A,C)`	`not(A,C)`

3.2.1. An example of goal compilation

This section gives a simple example of compilation to show how the goal compiler works in practice. Consider the following predicate in standard Prolog:

```
a(X, Y) :- Y is X+1.
```

This is converted to kernel Prolog:

```
a(X, Y) :- '$add'(X, 1, Y).
```

To compile the call to `'$add'/3` it is necessary to pass parameters in the right way. In particular, it is necessary to pass the output of the addition into variable Y. The built-in `'$add'(A,B,C)` has the following types associated with it:

Require = `(deref(A),deref(B),uninit_reg(C))`.
After = `(integer(A),integer(B),integer(C),rderef(A),rderef(B),rderef(C))`.

From the Require type, the first two arguments X and 1 of `'$add'/3` must be dereferenced and the third argument Y must be an uninitialized register. The Given types of X and Y depend on the type formula for `a(X,Y)`. Assume first that no type is given for `a(X,Y)`. From Tables 5.1 and 5.2, the Given flag for Y is `ini` and the Require flag for Y is `reg`. From Tables 5.3 and 5.4, the precode in this case is empty and the postcode is a call to unify(A, B) to generate unification code. The compiled BAM code is:

```
procedure(a/2).
    deref(r(0),r(0)).                ; Dereference X.
    add(r(0),1,r(0)).                ; Perform the addition.
    deref(r(1),r(1)).                ; Dereference Y.
    unify(r(0),r(1),nonvar,?,fail).  ; Unify Y with the result of the addition.
    return.
```

If `a(X,Y)` has a type then the code can often be simplified. For example, assume that its type is `(deref(X),uninit_mem(Y))`, i.e. X is dereferenced and Y is an uninitialized memory variable. Then the Given flag for Y is `mem`. The compiled BAM code is:

```
procedure(a/2).
    add(r(0),1,r(0)).                ; Perform the addition (X is dereferenced).
    pragma(tag(r(1),tvar)).
    move(r(0),[r(1)]).               ; Bind Y to the result of the addition.
    return.
```

3.3. The unification compiler

This section gives an overview of the compilation of unification, the optimizations that are done, and several examples.

3.3.1. The unification algorithm

Given a unification goal and type information about its arguments, this algorithm generates the simplest possible code to implement the unification. In the general case, the algorithm builds a tree of instructions. Each node of the tree has three branches—one each for read mode and write mode unification, and one for failure. The algorithm generates dereference instructions if necessary and trail instructions to undo variable bindings when backtracking. It does other optimizations including optimal write mode unification, type propagation, and depth limiting.

Write mode unification of a term generates a block of push instructions that builds the term on the heap. Read mode unification of a term is done sequentially for each of the term's arguments. First it checks the name and arity of the term. Then the arguments are unified. For arguments that are simple terms this consists of a single `move`, `equal`, or `unify` instruction. For arguments that are compound terms the unification algorithm is called recursively.

The function unify(X, Y, F, V_{sf}) defines the unification algorithm (Figure 5.4 and 5.5). Its inputs are the two terms to be unified (X and Y), the type formula true on input (F), and the set of variables that have a value on input (V_{sf}). Its outputs are a list of BAM instructions (*Code*), the type formula true on output (F_{out}), and the set of variables that have a value on output ($V_{sf,out}$).

The algorithm does several tasks that are not shown in the figure since they would unnecessarily complicate the presentation. The instruction list, the type formula, and the variable set are updated continuously during the compilation. Before using the value of a variable, it is dereferenced if necessary. Before binding a value to a variable, it is trailed if necessary. A variable occurrence list (varlist) is calculated for the register allocator (Figure 5.2).

3.3.2. Optimizations

The actual implementation does four optimizations not shown in Figure 5.4 and 5.5. It does optimal write mode unification. It keeps track of terms that are ground and recursively dereferenced to avoid compiling superfluous write mode unifications and dereferences. To reduce code size, it performs the last argument optimization and the depth limiting transformation.

3.3.2.1. Optimal write mode unification

The algorithm is modified to build a compound term in write mode with the least number of move instructions. First the code for building the main functor with empty slots for its arguments is generated. This is followed by the code for building the arguments and filling in the slots with the correct heap offsets.

```
function unify(X, Y: term; F : formula; V_sf : set) return (Code : list; F_out : formula; V_sf,out : set);
begin
    Code := [ ];
    if (var(X) and var(Y)) then begin
        if (F implies (unbound(X) or unbound(Y))) then
            Compile a store instruction
        else
            Compile a call to a general unification subroutine;
        return
    end else if (nonvar(X) and nonvar(Y)) then begin
        Compile a check that X and Y have the same functor and arity a;
        for i := 1 to a do begin
            Append unify(X_i, Y_i, F, V_sf) to Code
        end;
        return
    end if (nonvar(X) and var(Y)) then Swap X and Y
    else if (var(X) and nonvar(Y)) then Do nothing;

    if (X ∉ V_sf) then return unify_write(X, Y, F, V_sf);
    else begin /* At this point X ∈ V_sf */
        if (F implies nonvar(X)) then return unify_read(X, Y, F, V_sf)
        else if (F implies var(X)) then return unify_write(X, Y, F, V_sf)
        else begin
            Compile a three-way conditional branch comparing the tags of X and Y;
            Call unify_read and unify_write to compile the read and write mode branches
        end
    end
end;
```

Figure 5.4 – The unification compiler: the main routine

This technique was proposed as an optimization over the WAM by André Mariën [44]. The examples of unification given later use this technique. The justification of the BAM instructions needed for unification was done with this technique (Chapter 3).

3.3.2.2. Last argument optimization

This is an important optimization that significantly reduces the code size. It can be performed whenever a compound term has a compound term in its last argument. Without this optimization, the tree generated by the algorithm has the same depth as the term that is compiled. For each level in the tree a new block of write mode code is generated. For lists of n elements this results in $O(n^2)$ move instructions. The optimization reduces the code size to $O(n)$ by creating only a single write mode block, and letting all depths of the tree jump into it. This optimization was proposed by Mats Carlsson [14]. The code for write

```
function unify_write(X, Y: term; F : formula; V_sf : set) return (Code : list; F_out : formula; V_sf,out : set);
begin
    /* At this point X is an unbound variable */
    Generate a block of instructions to create the term Y on the heap;
    Bind X to this block (i.e. generate code to dereference X if necessary,
    store a pointer to this block in X, and trail X if necessary)
end;

function unify_read(X, Y: term; F : formula; V_sf : set) return (Code : list; F_out : formula; V_sf,out : set);
begin
    /* At this point Y is a nonvariable and F implies nonvar(X) */
    Code := [ ];
    Compile a check that X contains a structure of same functor and arity as Y;
    for i := 1 to arity(Y) do begin
        Append unify(X_i, Y_i, F, V_sf) to Code
    end
end;
```

Figure 5.5 – The unification compiler: read and write mode unification

mode unification of a nested term is replaced by a single jump instruction to the write mode code block of the outermost term. An example of unification given below uses this optimization.

3.3.2.3. Type propagation

There are two ways in which propagating type information during the compilation of unification improves the code. First, during the unification, the algorithm keeps track of the variables that are ground, uninitialized, and recursively dereferenced. This information is propagated into the arguments of compound terms. The propagation of ground and recursively dereferenced types was added after measurements of the dataflow analyzer showed that these types are numerous.

Second, when a new variable is encountered in a term, then the unification compiler has the choice whether to create it as an initialized variable or as an uninitialized variable. It is not always best to create new variables as uninitialized, since this often makes it impossible to apply last call optimization. To solve this problem it is necessary to look ahead in the clause. The variable is created as uninitialized only if there is a goal later in the clause with this variable in an argument position that must be uninitialized.

3.3.2.4. Depth limiting

Because the unification compiler generates a separate read and write mode branch for each functor in the term that is unified, deeply nested terms result in a code size explosion. The last argument optimization (see above) reduces the code size when the nesting occurs in the last argument. For other cases, a different technique is necessary. The unification compiler replaces a deeply nested subterm by a variable, creates the subterm with write mode unification and does a general unification with the variable. The depth limit is set by the compiler option `depth_limit(N)`, and the default depth is `N=2`. For example, consider the following unification where the complicated term `z(...)` is nested deeply:

```
X=s(t(u(...z(...)...)))
```

It is replaced by a sequence of three unifications:

```
X=s(t(u(...A...))), B=z(...), A=B
```

The variable B does not yet have a value, so the unification `B=z(...)` is executed in write mode. A general unification is performed for `A=B`. Since the size of a write mode unification is linear in the size of the compound term, this considerably shortens the code for deeply nested terms. Measurements were done to determine the effect of this transformation on execution time. In most cases it is insignificant, e.g. for the nand benchmark (Chapter 7), a program that contains deeply nested structures, the difference in execution time between depth limits of two and three is insignificant (i.e. only a few cycles out of several hundred thousand).

3.3.3. Examples of unification

Consider the following sample clause:

```
a(A, s(A,[X|X])).
```

The WAM code for this clause is (assuming the two arguments of the clause are in registers `r(0)` and `r(1)`):

```
procedure  a/2                      ;; the clause has two arguments.
    get_structure  s/2,r(1)         ;; unify r(1) with s(A,[X|X]).
    unify_value  r(0)               ;; unify the first argument with r(0).
    unify_variable  r(3)            ;; load the second argument into r(3).
    get_list  r(3)                  ;; unify r(3) with [X|X].
    unify_variable  r(2)            ;; load the first argument into r(2).
    unify_value  r(2)               ;; unify the second argument with r(2).
    proceed                         ;; return to caller.
```

Temporary values are stored in registers `r(2)` and `r(3)`. The execution time of this code averaged over read and write mode is 63 cycles on the Xenologic X-1 processor [85], an implementation of the PLM architecture [28]. The BAM code generated for the same clause is (the pragmas have been left out for clarity):

```
procedure(a/2).
    deref(r(1),r(1)).                           ;; dereference r(1).
    switch(tstr,r(1),l(a/2,3),l(a/2,4),fail). ;; three-way branch.
label(l(a/2,3)).                                ;; write mode for s(A,[X|X]).
    trail(r(1)).                  ; conditionally push r(1) on trail stack.
    move(tstr^h,[r(1)]).          ; bind s(A,[X|X]) to second argument.
    push(tatm^(s/2),h,1).         ; create the term s(A,[X|X]).
    push(r(0),h,1).
    push(tlst^(h+2),h,1).
    pad(1).
label(l(a/2,1)).                  ; common code for last arg. opt.
    move(tvar^h,r(2)).            ; create the two arguments of [X|X].
    push(r(2),h,1).
    push(r(2),h,1).
    return.
label(l(a/2,4)).                                ;; read mode for s(A,[X|X]).
    equal([r(1)],tatm^(s/2),fail).; check functor & arity of s/2.
    move([r(1)+1],r(3)).          ; load first argument into r(3).
    deref(r(3),r(3)).
    deref(r(0),r(0)).
    unify(r(3),r(0),?,?,fail).    ; unify first argument with r(0).
    move([r(1)+2],r(0)).          ; load second argument into r(0).
    deref(r(0),r(0)).
    switch(tlst,r(0),l(a/2,6),l(a/2,7),fail). ;; three-way branch.
label(l(a/2,6)).                                ;; write mode for [X|X].
    trail(r(0)).
    move(tlst^h,[r(0)]).
    jump(l(a/2,1)).               ; jump to common code (last arg. opt.).
label(l(a/2,7)).                                ;; read mode for [X|X].
    move([r(0)],r(2)).
    move([r(0)+1],r(0)).
    deref(r(0),r(0)).
    deref(r(2),r(2)).
    unify(r(0),r(2),?,?,fail).    ; unify arguments of [X|X].
    return.
```

Again, the two arguments of the clause are in registers `r(0)` and `r(1)` and temporary values are stored in registers `r(2)` and `r(3)`. To reduce the code size, the write mode code for `[X|X]` jumps into the middle of the code for `s(A,[X|X])`. With this optimization the code is 29 BAM instructions long (after translation and instruction reordering, this is 264 bytes on the VLSI-BAM). The WAM code is only 7 instructions long (17 bytes on the PLM) because each instruction encapsulates a choice. WAM instructions for unification assume the existence of a read/write mode bit in the implementation, which collapses the execution tree onto itself.

The code size ratio VLSI-BAM/PLM is large for this example. It was hoped during development that (1) code expansion would be less for other kinds of Prolog code (e.g. calls, parameter passing, backtracking), and (2) dataflow analysis would reduce the complexity of unifications. These intuitions have been borne out (Chapter 7): the static code size in VLSI-BAM bytes measured for large programs is only three times that of the PLM, a microcoded WAM with a byte-coded instruction set.

The execution time of the above code on the VLSI-BAM is 25 cycles (measured with a simulator taking pipeline delays into account and averaged over read and write mode). This is about 40% of the cycles needed for the X-1. This time can be estimated by taking the average execution times of BAM instructions when translated to the VLSI-BAM architecture: `unify` takes 5 cycles, `equal` takes 3 cycles, `switch`, `deref`, `trail`, and `move` from memory take 2 cycles each, `push`, `adda`, and all other `move` instructions take 1 cycle each, and `pad` instructions take 0 cycles because they are collapsed into the pushes. These estimates are only approximately correct because of instruction reordering optimizations performed on VLSI-BAM code.

Through programmer annotation or dataflow analysis it is sometimes possible to know the type of an argument at compile-time. For example, sometimes it is known whether an argument is unbound or bound. Consider the same sample clause again:

```
a(A, s(A,[X|X])).
```

Assume it is known that the second argument is an uninitialized memory variable. This is expressed with the following type declaration:

```
:- mode((a(A,B):- uninit_mem(B))).
```

With this type the clause's code is only 9 BAM instructions long (36 bytes on the VLSI-BAM):

```
procedure(a/2)
    move(tstrˆh,[r(1)]).     ;; bind s(A,[X|X]) to second argument.
    push(tatmˆ(s/2),h,1).    ;; create the term s(A,[X|X]).
    push(r(0),h,1).
    push(tlstˆ(h+2),h,1).
    pad(1).
    move(tvarˆh,r(0)).       ;; create the two arguments of [X|X].
    push(r(0),h,1).
    push(r(0),h,1).
    return.                  ;; return to caller.
```

The execution time of this example is 11 cycles.

3.4. Entry specialization

For each goal in the clause, the clause compiler attempts to replace it with a faster entry point, depending on the types existing at that point. For example, if it is known that the arguments N and A of the predicate `functor(X,N,A)` are atomic then a faster version can be compiled.

Entry specialization is done in both the clause compiler and the dataflow analysis. Doing it in both places is complementary since the analysis only keeps track of a limited set of types: ground, nonvariable, uninitialized, and recursively dereferenced. During clause compilation more information is known, for example, if the goal `X<Y` occurs in a clause, then afterwards it is known that `X<Y` is true. Analysis does not have a representation for this information, but it could be useful for entry specialization.

Figure 5.6: Example of a modal entry tree for entry specialization

Entry specialization can be done for any predicate whose definition is not in the program. The system has implemented this for the built-in predicates, but it can be used by the programmer for any library predicate. For each predicate that has faster entry points, a `modal_entry` declaration is given, along with type declarations for the fast entry points. These declaration are used in the dataflow analysis and the clause compiler to replace any call to the predicate with a faster entry point. For example, here is the modal entry declaration for the `name(A,B)` built-in predicate:

```
:- modal_entry(name(A,B),
   mode(atomic(A),
        mode(uninit(B),
             entry(’$ name > 1 *2’(A,B)),
             entry(’$ name > 1’(A,B))
            ),
   mode(uninit(A),
        entry(’$ name < *1’(A,B)),
        mode(uninit(B),
             entry(’$ name > *2’(A,B)),
             entry(name(A,B))
            )
       ))).
```

This declaration defines a binary tree, depicted in Figure 5.6. The nodes of the tree are decision points containing a type. If the type is valid then the left subtree is chosen, otherwise the right subtree is chosen. The leaves of the tree are the entry points. If none of the types are valid then the leftmost leaf is chosen, which usually is the same predicate as the original one. Each of the four fast entry points also has a type declaration:

```
:- mode('$ name > 1'(A,B),    (deref(A),deref(B)),  atomic(A),
                              (list(B),ground(B)), n).
:- mode('$ name < *1'(A,B),   (uninit(A),deref(B)), true,
                              (atomic(A),deref(A),list(B),ground(B)), n).
:- mode('$ name > *2'(A,B),   (deref(A),uninit(B)), true,
                              (atomic(A),list(B),ground(B),rderef(B)), n).
:- mode('$ name > 1 *2'(A,B), (deref(A),uninit(B)), atomic(A),
                              (list(B),ground(B),rderef(B)), n).
```

These declarations are written in a five-argument form that is more general than a standard type declaration (Appendix A): it gives the entry types (both Require and Before) and the exit (After) types for the predicate.

3.5. The write-once transformation

In the BAM all unbound variables are kept on the heap. This makes trail checking significantly faster. However, when combined with the ability to destructively modify the value of permanent variables (e.g. to dereference them and save the dereferenced value in the permanent) it leads to several problems. These problems are all neatly resolved by the write-once transformation.

Putting all unbound variables on the heap means that there are no pointers to the environment/choice point stack; all pointers point to the heap. This reduces trail checking to a single comparison with the heap backtrack pointer `r(hb)` and a conditional push to the trail stack. It is not necessary to do another comparison to decide whether the variable is on the heap or in an environment. In addition, since all unbound variables are created on the heap there are no ''unsafe variables'' as in the WAM. An unsafe variable is an unbound variable that is created on the environment and that must be moved to the heap (''globalized'') before last call optimization deallocates its memory.

Modifying the value of a permanent variable (e.g. by dereferencing or binding it) cannot be done without a trail operation. Indeed, consider the case where a permanent dereferences to a nonvariable term. If the dereferenced value overwrites the original value, then both the original value and its address have to be trailed since backtracking has to restore the original value. This is expensive, since it has to be done every time a permanent is bound or dereferenced.

One solution to this problem is never to store a dereferenced permanent back in the environment. This solves the problem but it is inefficient since a permanent may have to be dereferenced several times in a clause.

A better solution is to allocate a new permanent on the environment whenever the value of an old one needs to be changed. The new permanent gets the new value and the old permanent is unchanged. As a result, all permanent variables are only given values once, so they are called ‘‘write-once’’ permanents. Because it is not changed, the old permanent does not have to be trailed. At the cost of a slightly bigger environment, this completely eliminates the need to trail permanent variables. This allocation scheme is implemented in the clause compiler.

To summarize:

(1) All unbound variables are created on the heap, and unbound permanent variables in an environment always point to the heap.

(2) The trail check is a single comparison with `r(hb)` and a conditional push to the trail stack (2 cycles on the VLSI-BAM).

(3) Permanent variables are only given a single value in a clause. Whenever a permanent would be changed, a new one is allocated and given the modified value.

(4) Register allocation must allocate a different permanent register for each permanent variable in the clause. It is not allowed to use the same register for two variables whose lifetimes do not overlap.

This solution is implemented in the clause compiler by mapping a permanent variable onto a new variable whenever its value would change. The register allocator treats the new variables just like any other, and allocates them to temporary or permanent registers.

The main disadvantage of this technique is that environments are larger. For example, consider a clause of the form:

```
e(A,E) :- a(A,B), b(B,C), c(C,D), d(D,E).
```

where variables are chained from one predicate to the next. In the WAM, it is allowed to allocate permanent variables such that variables whose lifetimes do not overlap are allocated to the same permanent register. For the above example, this requires just two permanent registers, so the total environment size is four words (it also includes registers `r(e)` and `r(cp)`). Only two permanents are needed no matter how long the chain of body goals is. This method requires trailing of the permanent's values, because backtracking must see the original values. This scheme is consistent with the original implementation of the WAM, i.e. binding permanent variables on the environment and globalizing unsafe variables to ensure correctness.

In contrast, the number of permanent variables needed by the write-once technique increases linearly with the length of the chain. For the above example, this requires four permanent variables, so the total environment size is six words. The total memory usage is increased by less than this amount because no trailing of permanents is needed.

This is an example of a trade-off between memory space and execution time. The extra memory space needed is comparable to the increased size of the trail stack if there is no trail check for permanent variables. Since this is small, I have opted to decrease execution time at the expense of larger environments. By keeping all unbound variables on the heap and by implementing permanent variables as write-once variables, permanent variables can be dereferenced and bound without trailing, and the cost of trailing heap variables is reduced to a single comparison and conditional push.

3.6. The dereference chain transformation

This transformation is needed to maintain consistency between the dataflow analysis and the clause compiler. A new unbound variable (of either initialized type or uninitialized memory type) is created as a pointer to a memory location. Binding the variable stores the new value in the location. However, the register(s) that originally contained the unbound variable still have pointers to the location. One level of indirection is needed to access the value.

Figure 5.7 – The need for the dereference chain transformation

To see why this is necessary and what it implies, consider the execution of the clause `main` (Figure 5.7):

```
main :- (i) a(A), (ii) write(A).

a(A) :- A=s(t(a),u(b),v(c)).
```

The relevant situation can be seen in the transition from *(i)* (just before the call to `a(A)`) to *(ii)* (just after the call to `a(A)`). At *(i)* a new unbound variable A is created on the heap. At *(ii)* the variable A has been bound to a value. The important point is that A still has a `tvar` tag, and that one indirection is needed to access the `tstr` pointer. The extra link exists because the creation of A and its binding are done in separate steps. This is true for both initialized unbound variables and uninitialized memory variables.

This situation is not a problem unless dataflow analysis determines that A is returned as a dereferenced value. In that case there is a conflict between what the analysis deduces and what the clause compiler thinks is true. There are two ways to solve this problem: either weaken the analysis so that it will not deduce a dereference type in this case, or modify the clause compiler to ensure that the variable is dereferenced by doing an extra indirection whenever the variable is accessed after it is bound. The compiler implements the second solution since dereferencing is a time-consuming operation and it is important to derive as many dereference types as possible. The trade-off between doing an extra indirection for a value that may not be accessed later and doing an extra dereference loop seemed to be a fair one.

The compiler inserts code to do this indirection whenever the variable is accessed after it is bound. In addition to maintaining consistency with the analysis, this speeds up later dereferencing. There is a minor interaction with the register allocator—for correctness, variables that get an extra indirection are not allowed to be pref pairs.