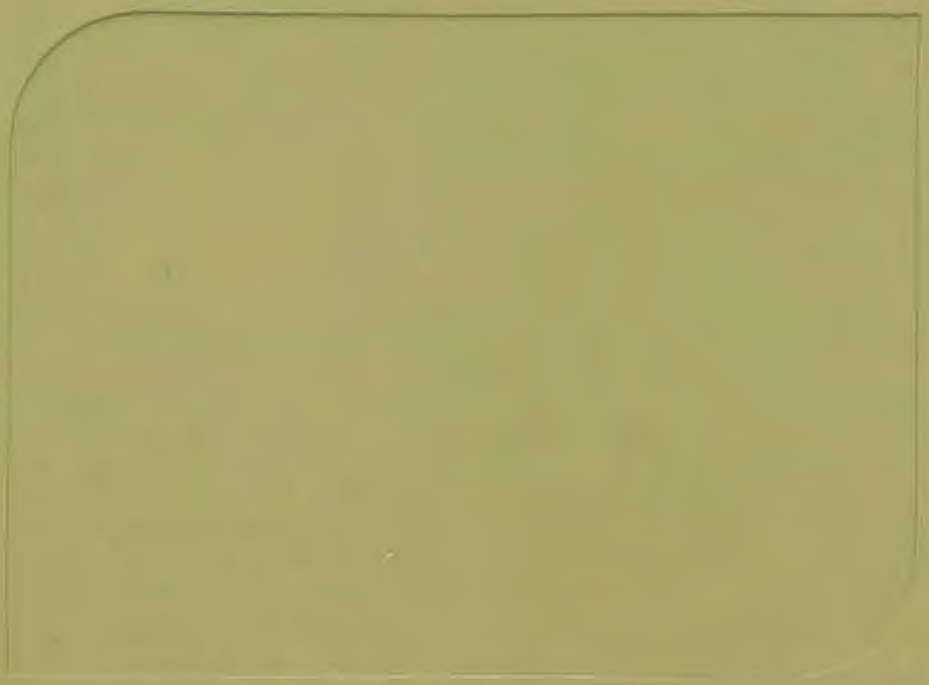
TCPW/CW/K.INT.



# BIM-PROLOG

Joint Project between
BIM
and
Department of Computer Science
Katholieke Universiteit LEUVEN

Sponsored by DPWB/SPPS
under grant nr KBAR/SOFT/1

# Compilation of WIC to Assembler

by

Herman CRAUWELS *

Internal Report
BIM-prolog IR11

April 1985

* BIM
Kwikstraat 4
B-3078 Everberg Belgium
tel. +32 2 759 59 25

** Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A
B-3030 Heverlee Belgium
tel. +32 16 20 06 56

DPWB = Diensten van de eerste minister : Programmatie van het Wetenschapsbeleid.
SPPS = Services du premier ministre : Programmation de la Politique Scientifique.

# COMPILATION OF WIC TO ASSEMBLER

Herman Crauwels

BIM

ABSTRACT

This paper describes how WIC can be converted to VAX assembly code. The resulting code can be directly executed without an interpreter. Several optimizations in the assembly code are discussed. Problems concerning the integration of the code for the builtin predicates are mentioned. In the last section some "improvements" for the interpreter are suggested.

1. Structure of the compiled program.

A prolog program is converted to a WIC file, which can be interpreted. This interpretation is now skipped by converting the WIC file into an executable program. Most modules of this program are more or less identical to modules in the interpreter. The main module is simplified and the interpreting module is changed to a routine called "query":

```
jmp_buf kern_env;       /* global variable */

main()
{
        initializations();
        if (setjmp(kern_env) == 0 )
        {
                query();
                exit(0);
        }
        else
                exit(1);
}

query()
{
        /* initializations */
        jump to STARTQUERY
        /* compiled static code */
        /* dynamic code is not allowed */
STARTQUERY:
        /* compiled query */
}
```

The query routine is constructed from the WIC code file by translating each WIC instruction into a set of assembler instructions.

The general structure of such a set is:

```
copy arguments to predefined places
        registers six and seven
jump to local subroutine
        corresponding to the WIC instruction
```

The subroutines for some WIC instructions are so small that inline substitution is more efficient both in space and in time (e.g. most PUT-instructions ). Other WIC instructions must be translated in line because a subroutine would give very complicated code (e.g. SWITCH_ON_TERM ).

## 2. Usage of hardware registers.

Before the code of the different WIC instructions is described, a summary is given of used hardware registers.

The value fields of the E, H and A WIC registers are stored in the general registers three, four and five:

```
value field of E register (_E+4)    = r3
value field of H register (_H+4)    = r4
value field of A register (_A+4)    = r5
```

Each time evalpred is called these three registers are saved on the machinestack by a "pushr" instruction.

The B WIC register can not be put efficiently in a general register because it must be available in the routine "falen". This can not be guaranteed with the save mechanism used for r3,r4 and r5:

```
query
{
        ...
        pushr   r3,r4,r5
        evalpred()
        popr    r3,r4,r5
        ...
}

evalpred
{
        ...
        falen()
}

falen()
{
        B must be accessible
        ...
        reset r3,r4,r5 from information kept in choicepoint
        ...
}
```

In the routine "falen" B cannot be popped from the machine stack because nobody knows in which frame on the machine stack B is saved.

In stead of saving the B WIC register on the machinestack, it can also be copied in a global variable before each call to "evalpred" and restored after the return but this seems to give too much overhead.

The P and CP WIC registers are not used very frequently in the generated assembler code; no hardware registers are reserved for them.

What to do with the remaining two registers (HB and TR) is not yet decided.

Register 6 and 7 (r6 and r7) are used to store the first and second argument (if any) before a local subroutine call is made.

UNIFY instructions have at most one argument. So register seven (r7) is used to temporarily store the value field of the S WIC register.

The startaddresses of the Aregister and Xregister areas are put in registers 8 and 9 (r8 and r9).

```
        moval   _Aregister+8,r8
        moval   _Xregister+8,r9
```

## 3. The generated code.

### 3.1. Inline substitutions.

#### 3.1.1. PUT-instructions.

The put_constant (int and real) is the simplest one. The type field of the Aregister gets the type. For integers the integer value itself is put in the Aregister. For reals end constants a pointer to the value is put in the Aregister. For example:

```
        movb    $106,(r8)
        moval   _ct+8,4(r8)
```

The put_yvar instruction initializes the Aregister in the same way, but it also "undefs" the variable in the environment:

```
        movb    $109,(r8)
        moval   16(r3),r0               # r3 = _E+4
        movl    r0,4(r8)
        movb    $103,(r0)
```

The put_xvar and put_void instructions are very analogous.

In the put_list instruction the value field of the Aregister is initialized with the heap pointer. Also the global variable "mode" must be set to WRITE (1).

```
        movb    $101,(r8)
        movl    r4,4(r8)
        cvtlw   $1,_mode
```

The put_structure instruction does the same and also sets the pointer to the structure on the heap.

```
movb    $102,(r8)
movl    r4,4(r8)
movl    $110,(r4)+
moval   _ft+80,(r4)+
cvtlw   $1,_mode
```

### 3.1.2. UNIFY-instructions.

After a put_list and a put_structure instruction the unify_mode is always WRITE and it does not change during a unify_list of unify_structure instruction. With this knowledge the code generation of some unify instructions can be somewhat optimized. These instructions are:

| | |
|---|---|
| unify_constant | unify_yvar |
| unify_int | unify_xvar |
| unify_real | unify_void |

In stead of inserting the complete code for the instruction and doing the "mode" test at runtime, only the code for the WRITE case is generated. This code is very similar to code described in the previous section about the PUT-instructions.

### 3.1.3. SWITCH_ON_TERM-instruction.

The switch_on_term instruction has four parameters: the addresses of WIC instructions where execution can start if the first Aregister has a specific type. These four addresses are translated into labels in the assembler code. Some of the addresses can be zero, but that does not matter; the label can still be generated. The testing of the type is done with a "caseb" instruction.

```
                #  switch_on_term  a1 a2 a3 a4

        moval   (r8),r6
        jbr     2f
1:
        movl    4(r6),r6
2:
        cmpb    (r6),$109
        jeql    1b
        caseb   (r6),$100,$7
1:
        .word   SxSa2-1b        # type = INT
        .word   SxSa3-1b        # type = LIST
        .word   SxSa4-1b        # type = STRUCT
        .word   SxSa1-1b        # type = UNDEF
        .word   SxSa1-1b        # type = KREF
        .word   SxSa1-1b        # type = SREF
        .word   SxSa2-1b        # type = CONST
        .word   SxSa2-1b        # type = REAL
SxS0:
        calls   $0,_falen
SxSa1:
        #  code if first A register has type UNDEF
SxSa2:
        #  code if first A register has type CONST, REAL or INT
SxSa3:
        #  code if first A register has type LIST
SxSa4:
        #  code if first A register has type STRUCT
```

Although cases KREF and SREF are mentioned in the test because of the structure of the "caseb" instruction, the first argument can never have those types.

### 3.1.4. Other instructions.

The code for some WIC instructions is small and can be directly inserted.

```
PROCEED :
        jmp     *_CP+4
ALLOCATE :
        movb    _E,(r5)
        movl    r3,4(r5)
        movl    _CP+4,12(r5)
        movl    r5,r3
EXECUTEE :
        pushr   $824
        pushl   $-3
        pushl   $number_of_builtin_predicate
        calls   $2,_evalpred
        popr    $824
        movb    $70,_E
        jmp     *_CP+4
EXECUTEC :
        movb    $70,_E
        jmp     $procedure
CALLE :
        pushr   $824
        pushl   $number_of_permanent_variables
        pushl   $number_of_builtin_predicate
        calls   $2,_evalpred
        popr    $824
        movb    $70,_E
INIT :
        movb    $103,address_of_permanent_variable
JUMP :
        jmp     $instruction_after_orlist
```

The "retry" instructions must only substitute the "alternative" field in the choicepoint:

```
        movl    _B+4,r1
        moval   $alternative,12(r1)
        movb    $78,_E
```

The reset of the A register is already done in the routine "falen".

The "try_me_else" and "try" instructions are translated into a procedure call to "s_createchoice". The update of the B, HB and A register is also done in that routine.

## 3.2. Subroutines.

### 3.2.1. GET, PUT and UNIFY -instructions.

All GET instructions are translated to a subroutine call. The UNIFY instructions are also translated to a subroutine call when it can not be determined at compile time that only the WRITE unify case is needed. This is after a get_list or get_structure

instruction. The put_yval, put_xval and put_unsafe instructions do a lot of testing; so a subroutine is needed because of space optimizations. Before the subroutine call is made register six and if needed register 7 are initialized to the arguments of the instructions.

If the instructions uses a Aregister, its address is put in register 7:

```
        moval   (r8),r7         # if Aregister[1]
```

The first argument or a reference to it is put in register 6 depending on the type of that argument:

```
type = INT       :  movl        $value,r6
type = REAL      :  moval       address_to_the_real,r6
type = CONST     :  moval       address_to_table_of_constants,r6
type = STRUCTURE :  moval       address_to_table_of_functors,r6
type = permanent :  moval       16(r3),r6
type = temporary :  moval       (r9),r6
```

In the subroutine itself the necessary tests and moves are performed using registers 6 and 7. A side-effect is that some subroutines implementing a WIC instructions are identical:

```
        PUTxval == PUTyval
        GETxvar == GETyvar
        GETxval == GETyval
```

### 3.2.2. Other instructions.

The "ortry" instruction is translated into a subroutine call because before the call to "s_createchoice" the update of the A register must be done and this takes a few instructions.

```
        movl    $length_of_environment,r6
        moval   alternative,r7
        jsb     ORtry
```

The "trust" instructions are converted to a subroutine call because there are no arguments. It thus takes just one line of code.

Also the lastcut instruction is converted to a subroutine call with no arguments.

The "callc" instruction has two arguments. The first argument, a reference to the procedure that is called, is put in _P+4. The second argument, the length of the environment is put in r6. Also the continuationpointer CP must be set. In stead of a normal subroutine call, here a jump is made to the subroutine. This is done because the return from Callc will be done using the continuationpointer.

```
        movl    $length_of_environment,r6
        moval   1f,_CP+4
        moval   $procedure,_P+4
        jmp     Callc
1:
```

Dealexc and dealexe are translated in the same way.

```
DEALEXC :
        moval   $procedure,_P+4
        jmp     DEalexc
DEALEXE :
        movl    $number_of_builtin_predicate,r6
        jmp     DEalexe
```

4. Generation of labels.

There are several places where labels must be inserted in the assembler code:

1 The code of each prolog procedure starts with a label which is the concatenation of an underscore (_), the name of the predicate and the arity of the predicate. These labels are used by the CALLC, DEALEXC and EXECUTEC instructions.
Remark. If the name of the predicate does not start with an alphabetical character, a unique label of the form "PRDnumber" is generated.

2 The addresses of the SWITCH_ON_TERM instruction are translated to labels of the form

'S'-'number_1'-'S'-'number_2'

'Number_1' is the instruction address of the SWITCH_ON_TERM. 'Number_2' is one of the four arguments of the SWITCH_ON_TERM. For example:

at address 453 : SWITCH_ON_TERM 454 455 463 0

```
----->  S453S454
        S453S455
        S453S463
        S453S0
```

3 The addresses used in the TRY_ME_ELSE, RETRY_ME_ELSE, TRUST_ME_ELSE sequence are translated into '9f' labels:

```
                # address_1 TRY_ME_ELSE address_2 n
                create choicepoint
                choicepoint.alternative = 9f
                code for the first alternative
        9:
                # address_2 RETRY_ME_ELSE address_3
                choicepoint.alternative = 9f
                code for the second alternative
        9:
                ....
        9:
                # address_n TRUST_ME_ELSE
                remove choicepoint
                code for the last alternative
```

4 The addresses used in TRY, RETRY, TRUST and ORTRY, ORRETRY and JUMP are converted to a label that has the same form as in the SWITCH_ON_TERM instruction, except the the 'S' is replaced by a 'J'.

## 5. Implementation of the backtrack operation.

In the startup code of the query routine, the current frame-, argument- and stackpointer are saved in a global variable "back_buf". Each time backtracking is initiated in the routine "falen", these three registers are restored and then a jump is made to the code in the query routine. To what code is jumped, is determined by the alternative field in the choicepoint.

## 6. Optimizations in the code.

In the code the C-compiler generates for the WIC instructions some parts can be optimized:

REF - SREF - KREF :

These three types have now all the same value, REF. This means that during dereferencing only one value must be tested instead of three.

the copy of a real:

The "prs_move" macro tests the type of the item being moved and uses different statements for the move of a real or that of another type. When these C-statements are compiled into assembler instructions, there is no real difference between the move of a REAL value and another value. In both cases four bytes must be moved from one place to another. So during the translation of WIC to assembler this test on the type is not made and the move is always done

with a "movl" instruction.

the tests to reset:

Addresses are put on the trail stack if one of the following two situations holds:

- the address points to an item on the heap stack older than the item to which HB points

  ( cp_morerecent(HBval,address)
  && cp_morerecent(address, cp_bodem-1) )

- the address points to an item on the local stack older then the current choicepoint

  ( s_morerecent(Bval,address)
  && s_morerecent(address, s_bodem) )

The C compiler generates for these four tests ( address is stored in a local variable -8(fp) ):

```
        cmpl    _B+4,-8(fp)
        jleq    1f
        cmpl    -8(fp),_s_bodem
        jgtr    2f
1:      cmpl    _HB+4,-8(fp)
        jleq    3f
        subl3   $8,_cp_bodem,r0
        cmpl    -8(fp),r0
        jleq    3f
2:      movl    _TR+4,r0
        movl    -8(fp),4(r0)
        subl2   $8,_TR+4
3:
        #       next instruction
```

With WIC to assembler this becomes (address is stored in r7):

```
        cmpl    _HB+4,r7
        jgtr    3f
        cmpl    r7,_s_bodem
        jlss    6f
        cmpl    _B+4,r7
        jlss    6f
3:      movl    _TR+4,r0
        movl    r7,4(r0)
        subl2   $8,_TR+4
6:
        #       next instruction
```

The test on "cp_bodem-1" disappears because it always succeeds if HBval is morerecent than "address".

save-restore argumentregisters in choicepoint

The C-compiler generates here very inefficient code. In the assembler version the autoincrement addressing mode is used. For example, in "s_createchoice" the argumentregisters are saved in the choicepoint:

```
        addl3   $56,r5,r1               # r5 == _A+4
        moval   _Aregister+8,r0
        movl    $1,r11
        jbr     1f
2:
        movq    (r0)+,(r1)+     # move quadword (8 bytes)
        incl    r11
1:
        cmpl    r11,4(ap)       # the number of registers
                                # that must be saved
        jleq    2b
```

As a side effect register 1 has at the end of the loop the new value for the local stackpointer (A). In the interpreter however A is calculated:

```
        A <-  B + length_choicepoint
```

For restoring, the roles of register zero and one are interchanged.

falen_unif:

Each call of "falen_unif" is changed to a direct call of "falen".

Initialization of S register:

In stead of calling the routines "sinit" or "linit", the contents of register six or seven (possibly incremented with 8) is moved to _S+4.

resetting variables:

In "falen" the procedure call "r_varreset" is replaced by the body of the routine.

adding elements to the heap stack:

The explicit increment of the heap pointer is changed to an implicit one by using the autoincrement addressing mode:

```
INTERPRETER                     COMPILER
movl    _H+4,r4
movb    $type,(r4)              movl    $type,(r4)+
movl    $waarde,4(r4)           movl    $waarde,(r4)+
addl2   $8,_H+4
```

calculating A:

In the "call" instruction the A register gets a new value, either "E+length_environment" or "B+length_choicepoint". The second one takes some assembler instructions to be calculated:

```
        movl    _B+4,r0
        cvtbl   (r0),r0
        ashl    $3,r0,r0
        addl2   _B+4,r0
        movl    r0,_A+4
```

This calculation can be done once during the creation of the choicepoint and stored some where (e.g. in the first four bytes of the B register). Each time the value "B+length_choicepoint" is needed it can be found in the B register.

In the first "CALLC" instruction after an "ALLOCATE", it is known at compile_time that the new value for A is "E+length_environment". The test to see what is more recent (E or B) can be omitted. The code for such a first CALLC can be substituted in_line:

```
        moval   1f,_CP+4
        movb    $70,_E
        addl3   $length_environment,r3,r5
        jmp     _name&arity_predicate
        1:      # next WIC instruction
```

the cutflag:

The cutflag is stored in the type field of the E register. Each ALLOCATE instruction copies the flag to the top of the local stack (the place to which the A register points). So the following sequence of instructions is frequently used:

```
30 try_me_else 54 2
        ....
        set cutflag on          movb    $78,_E
31 allocate
        copy cutflag            movb    _E,(r5)
```

or

```
38 callc 80 3
        ....
        set cutflag off         movb    $70,_E
80 allocate
        copy cutflag            movb    _E,(r5)
```

By storing the cutflag on top of the stack in stead of in the E register, the copy in the ALLOCATE instruction can be skipped.

```
30 try_me_else 54 2
        ....
        set cutflag on          movb    $78,(r5)
31 allocate
```

or

```
38 callc 80 3
        ....
        set cutflag off          movb    $70,(r5)
80 allocate
```

There are cases where the cutflag is set and directly thereafter reset. For example:

```
10 try_me_else 14 3
        ....
        set cutflag on           movb    $78,(r5)
13 executec 80
        set cutflag off          movb    $70,(r5)
```

This is when after the "try_me_else" and "retry_me_else" (also "try" and "retry") no ALLOCATE instruction follows. The "set cutflag on" statement can then be omitted.

The saved cutflag in a choicepoint is never used or restored. So there is no need to save it during the creation of the choicepoint.

The cutflag is not changed during the CALLE, EXECUTEE and DEALEXE instructions.

save-restore registers:

When two builtin predicates are called immediately after each other, the "save-" and "restore hardware register" statements between the the two calls can be dropped:

```
        movl    r4,_H+4
        pushr   $824
        pushl   $number_of_permanent_variables
        pushl   $number_of_builtin_predicate
        calls   $2,_evalpred
                #       popr    $824
                #       movl    _H+4,r4
                #       movl    r4,_H+4
                #       pushr   $824
        pushl   $number_of_permanent_variables
        pushl   $number_of_builtin_predicate
        calls   $2,_evalpred
        popr    $824
        movl    _H+4,r4
```

# 7. Optimizations in WIC.

## 7.1. The put_unsafe.

The sequence of tests to see if a permanent variable is unsafe is changed:

The interpreter:

```
if ( s_onadres(address)
        && s_morerecent(address, E_register)
        && address->type == UNDEF )
                address points to an unsafe variable;
else
        if ( address->type == UNDEF )
                A_register = REF , address;
        else
                A_register = *address;
```

Compiled:

```
if ( address->type == UNDEF )
        if ( s_morerecent(address, E_register) )
                address points to an unsafe variable;
        else
                A_register = *address;
else
        A_register = REF , address;
```

The test "s_morerecent" contains implicitly the test "s_onadres".

If "address" points to an unsafe variable, the variable is copied to the heap and the A_register gets a pointer to that heap location. The unsafe location itself is not changed, so its address is not put on the trail stack. If the variable is also needed in another A_register, it can not be done by a "put_yval" instruction. In stead the first A_register (containing the unsafe variable) is copied into the second A_register. Thus the sequence

```
put_unsafe      Y3,A1
put_yval        Y3,A2
```

is changed into

```
put_unsafe      Y3,A1
move_areg       A1,A2
```

For a unify_yval after a put_unsafe the argument for the subroutine performing the unify_yval is the address of the A_register, initialized in the put_unsafe instruction.

## 7.2. Generalization of move_areg.

If the same variable (permanent or temporary) is needed in two or more A_registers for a call, the second and following put instructions can be changed into move_areg's:

```
put_yvar Y3,A1
put_yval Y3,A3    --->   move_areg A1,A3
```

```
put_yval Y3,A1
put_yval Y3,A3     --->   move_areg A1,A3


put_xvar X3,A1
put_xval X3,A3     --->   move_areg A1,A3


put_xval X3,A1
put_xval X3,A3     --->   move_areg A1,A3
```

In the first call after the get_instructions some put instructions can be skipped because the A_register has still the good value:

```
get_yvar Y4,A1
get_yvar Y2,A2
get_yvar Y3,A3
put_yvar Y5,A1
        put_yval Y2,A2   can be skipped.
put_xvar X1,A3
callc ...
```

In other cases put instructions can be changed to move_areg's, from A_registers which are not yet overwritten:

```
get_yvar Y2,A1
get_yvar Y3,A2
get_yvar Y4,A3
put_yval Y3,A1   --->   move_areg A2,A1
put_yval Y4,A2   --->   move_areg A3,A2
callc ...
```

## 7.3. Deterministic calls.

If an argument in a call is a constant, and all predicates corresponding to that call have a constant in the same argument, then not all alternatives must be tried but only those whose "constant" argument is equal to the constant in the call. For example:

```
fact(a, b).
fact(a, c).
fact(a, d).
fact(a, e).

?- fact(_x, d), .. .
```

That call of "fact" can directly jump to the third alternative without creating a choicepoint.

If the "constant" argument of several predicates match then the call must be replaced by a "try - retry - trust" sequence.

## 7.4. Input-output mode declarations.

If the mode of an argument is known at compile-time then a specific input/output WIC instruction could be generated. The pseudo C-code for these specific instruction can be very simple because no testing on the mode must be done at run-time. Examples:

```
get_yvar_output  Y2,A1 :        Y2.type = REF
                                Y2.value = A1.value


get_yvar_input   Y2,A1  :       Y2.type = A1.type
                                Y2.value = A1.value
```

In these two instructions the dereferencing of A1 is postponed until later.

```
put_yval_output Y2,A1  :        if Y2.type == REF
                                        A1.value = Y2.value
                                else
                                        A1.value = address of Y2
                                A1.type = REF
```

Y2 is dereferenced because it possibly can disappear by trimming.

```
put_yval_input  Y2,A1  :        A1.type = Y2.type
                                A1.value = Y2.value


unify_xvar_input  X1   :        X1.type = S->type
                                X2.value = S->value
                                S++;


unify_xval_output X1   :        H->type = X1.type
                                H->value = X1.value
                                H++;
```

A little bit more difficult is the get_list_input because the argument can be the empty list in which case the instruction must fail:

```
get_list_input A1      :        a = deref(A1)
                                if a.type != LIST
                                        falen();
                                else
                                        S = a.value
                                        mode = READ
```

## 8. Builtin predicates.

With builtin predicates, there are two main problems. The first problem is that some builtin predicates can not be implemented. The second concerns the compilation of the "startup.o" file.

The code of a few builtin predicates can not be copied from the interpreter version to the compiler version.

1 The predicates "clause", "dump" and "listing" consult the code table. Because in the compiler version this table does not exist, these predicates are not available.

2 The predicates "assert", "retract", "retractall", "consult" and "reconsult" update the code table. Again this is not possible in the compiler version.

3 The predicates using the metacall ("call", "not" and "bagof") are partially integrated in the compiler version. However "bagof" uses "assert" and "retractall" and therefore it can not be called.

4 All functors in the functor_table have type "UNSPEC". So the "builtin" predicate would always return FALSE and therefore is not implemented.

The assembler generator converts one WIC code file to one assembler file. This assembler file contains one big routine, "query". It is not possible to convert two or more WIC code files to one assembler file. So it is not possible to use the builtin predicates that are defined in the "startup.pro" file.

### 8.1. The metacall.

The builtin "call" can be used in the compiler version. When the argument is a simple structure the implementation is straightforward. For "and"- and "or"-lists, something must be found to compile the ANALYSE_ANDLIST and ANALYSE_ORLIST cases of the interpreter.

In stead of generating the normal code for calling a builtin predicate, some specific code is produced:

```
set the CP register to the next WIC instruction
save r3,r4,r5 in the global variables E, H, A
(normally they are pushed on the stack)
call directly ev_call
restore r3,r4,r5
jump to the address found in the P register
```

In the "ev_call" routine the A register is updated and the address of the prolog procedure that is called, is looked up in

the symboltable of the loadmodule and put in the P register. For efficiency reasons, this address is also stored in the entry of the procedure in the functor table and the type of the entry is changed to "STAT". (So if the procedure is called again at a later time, its address must not be looked up in the symbol table but can be found in the functor table).

At initialization time (in the routine "main"), the file "BUILTIN_PRO" is read and the functortable is extended with the names of the builtin predicates. With this information it is possible to use the metacall with a builtin predicate as argument.

When the argument of the metacall is an "and-" or "or-list", some special WIC instructions are used:

```
ANALYSE_ANDLIST
ANALYSE_ORLIST
```

These two WIC instructions are converted to assembler routines and added to the set of expanded WIC routines. When the metacall of an and-list is executed, one sets the continuation pointer CP to the address of the ANALYSE_ANDLIST routine and calls recursively "ev_metacall" for the first argument:

```
call( (A1 , A2) )
--> routine ev_call
        ev_metacall( (A1 , A2) )
        --> routine ev_metacall
                it is an andlist
                CP <- &ANALYSE_ANDLIST
                Aval+VERVOLG <- A2
                ev_metacall( A1 )
                --> routine ev_metacall
                        P <- A1
                <--
        <--
<--
jmp     *P
```

With the "jmp" instruction the execution of the first argument starts. This execution ends with:

```
jmp     *CP     # CP points to the ANALYSE_ANDLIST routine.
ANALYSE_ANDLIST:
        pick up A2 from Aval+VERVOLG
        ev_metacall(A2)
        --> routine ev_metacall
                P <- A2
        <--
        jmp     *P
```

and the execution of the second arguments starts... .

For an orlist the sitation is very analogous. In this case the address of the ANALYSE_ORLIST routine is stored in the alternative field of the choicepoint.

## 8.2. The linker.

To be able to use the builtin predicates that are defined in the "startup.pro" file, a "linker" was written. This "linker" concatenates several WIC code files to one WIC file. (Starting from this one big WIC file assembler can be generated). During this concatenation several addresses must be changed:

- One "constant" table is built from the "constant" tables of each WIC file. This means that in the code each reference to the "constant" table must be changed.

- The same goes for the "functor" table.

- Local references to instructions are changed into global ones by adding the sum of the lengths of the code of the already processed files:

```
file1.o : length of static code = 76
file1.o : length of static code = 123

All references in file3.o are incremented with 199 .
```

- Some external references (CALLU, DEALEXU, EXECUTEU) can be resolved. For example: if file1.pro contains a call to a procedure that is defined in file2.pro

```
file1.pro:
        a :- b(_x), ... .

        put_xvar _x A1
        callu  b 1

file2.pro:
        b(_x) :- c(_x), ... .
```

then in the output file of the "linker", the CALLU instruction is changed to a CALLC instruction.

# 9. Results.

The compiled WIC code is tested on three programs:

rever:
a program that builds a list of 100 elements, reverses the list and counts the number of elements in the reversed list.

relat:
a program that looks up all possible relations (father, son, brother, nephew, ...) of a man with members of his family.

permu:
a program that generates all permutations of a list of eight (8) elements.

In the table below the execution times (user and system) of these programs are given both for interpretation and compilation. The user time of interpretation when all predicates are declared to be dynamic is given between parentheses. The column labeled with "subrout" gives the number of local subroutines that where called in the compiled execution. This number of subroutines gives an idea of how many WIC instructions are executed. The column labeled with "falen" gives the number of times the routine "falen" was called. This is the number of times backtracking occurred.

| | subrout | falen | compilation | | interpretation | | |
|---|---|---|---|---|---|---|---|
| rever | 47765 | 101 | 2.1u | 2.0s | 7.1u | 1.4s | ( 12.2u) |
| relat | 100838 | 73442 | 12.1u | 2.4s | 40.5u | 3.4s | ( 41.4u) |
| permu | 796772 | 46234 | 35.4u | 3.1s | 117.0u | 2.8s | (129.0u) |
| opt1 | 32515 | 101 | 1.4u | 1.5s | | | |
| opt2 | 33053 | 23078 | 4.4u | 1.7s | | | |
| opt3 | 265600 | 46234 | 23.3u | 2.6s | | | |

The entry "opt1" is the reverse program whereby optimized A_ and X_ register allocation is done. The entry "opt2" is the relations program with deterministic calls. The entry "opt3" is the permutation program whereby specific assembly code is generated for input and output arguments.

The "opt1" and "opt3" program give an improvement factor of 5 (!!!) between the compiled and interpreted version; "opt2" approximates an improvement factor of 10.

## 10. Suggestions for the interpreter.

### 10.1. The trail stack.

The type field of an entry in the trail stack is never used. The structure of the trail can thus be changed:

```
struct trail
{
        char    *tr_address;
};
```

There is also no need to locate the trailstack between the heap and the local stack. As a consequence the heap has then a constant upper bound ("st_midden") in stead of the variable TR register.

## 10.2. Types and unifymode.

Several switch statements in the C code have cases for INT, REAL, CONSTANT, LIST, STRUCTURE and UNDEF but not for SREF, KREF and REF. Therefore it seems natural to rearrange the numerical values assigned to types and give consecutive values to the types used in switch statements:

```
INT     100
REAL    101
CONST   102
LIST    103
STRUCT  104
UNDEF   105
FUNC    106
SREF    107
KREF    108
REF     109
```

As everybody knows, the types SREF, KREF and REF should be the same.

In stead of using a switch statement for testing "mode" in unify instructions an if-then-else can be used because it is guaranteed that "mode" is either READ or WRITE:

```
if (mode == READ)
        READ-actions
else
        WRITE-actions
```

## 10.3. Hardware registers.

Local variables in C procedure which will be frequently used can best be declared to be register variables. In routine "kern" this is the case for:

```
register WPRE   adres;
register WPRE   uadres;
register WPRE   varadr;
register WCP    cpadres;
register WS     ws;
```

## 10.4. Macro's.

The following procedures should be implemented as macro's:

- falen_unif: it simply calls falen.
- linit and sinit: initialization of the S register.
- r_varreset: one small loop to reset variables.

## 10.5. Builtin predicates.

In stead of the routine "evalpred" with the big switch an array with function addresses can be used:

```
evalpred(nr, nvar);
```

is changed into

```
(*evalpredarray[nr].ev_routine)(nvar);
```

Table of Contents