Prolog Execution Model

Applied Logic Systems, Inc.
Company Confidential

Date Printed: 3/20/98

Contents

¶1. Memory Areas
¶2. Data Structures / Tagging
  §2.1.Data Structures
  §2.2.Tags
¶3. Machine Registers
¶4. Stack Organizations
  §4.1.Argument/Environment Stack
  §4.2.Backtrack (Choice Point) Frames
¶5. Parameter Passing / Predicate Calling
  §5.1.Predicate Calls
  §5.2.Parameter Passing
¶6. Code Area Organization
¶7. Symbol Table, Name Table, & Relatives
¶8. Abstract Machine Instructions
  §8.1.Control Instructions
  §8.2.get Instructions
  §8.3.unify Instructions
  §8.4.put Instructions
  §8.5.Cut Instructions
¶9. Compiler Organization
  §9.1.Unit Clauses
  §9.2.Clauses with one goal
  §9.3.Clauses with multiple goals
  §9.4.Environments
¶10. Compiler Details
¶11. Interrupts
¶12. Decompilation
¶13. Debugger
¶14. Foreign Interface & Packaging

1. Memory Areas

1.1. Code Area

Contains prolog code. This area consists of blocks of one of three types:

- procedure entries
- index blocks
- clauses.

1.2. Argument Environment Stack

Arguments and environments are put on this stack. The stack discipline incorporates tail recursion optimization. It grows from high to low and is located below the heap.

1.3. Heap

Place where Prolog data structures live. This area grows from low to high (towards the choice point stack).

1.4. Choice Point / Trail Stack

Choice points and variables to reset upon failure are pushed on this stack. It grows from high to low and is located highest in memory (of the four areas).

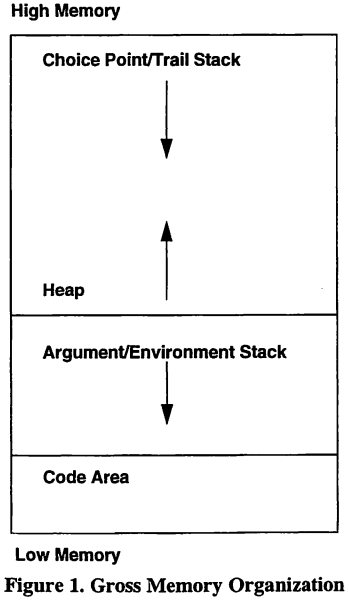


Figure 1. Gross Memory Organization

2. Data Structures / Tagging

2.1. Data Structures

Files:

80386 -mtypes.h, mtypes.m4

- 68020 -mtypes.h, winter.h
- 88000 -
- MacThreaded -
- VAX -
- RS/6000 -

We wish to accommodate the following types of primitive objects in this design:

- Constants
- Lists
- Structures
- Variables
- UIAs & other funny objects

All objects are composed of one or more (32 bit) words. Each object contains a tag identifying the type of the object.

- Constants include:

  integers, symbols, uninterned atoms, and floating point numbers.

  Integers:

  have only 24 bits, which is due to the byte-sized tag.

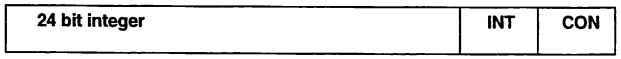


Figure 2. Tagged Integer Layout

  Symbols:

  entered into the symbol table.

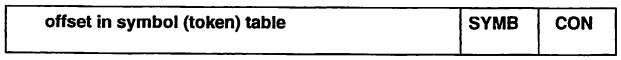


Figure 3. Tagged Symbol Layout

  Uninterned atoms:

  also symbolic program constants, but are not entered into the symbol table.

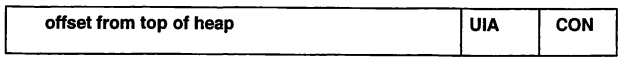


Figure 4. Uninterned Atom Layout.

A symbol and an uninterned atom with the same print name will unify. In fact, it is very difficult for the application programmer to distinguish between an uninterned atom and a symbol. The only difference between them is that uninterned atoms are stored on the heap as strings and take longer to access. In particular, they take much longer to unify. Comparing UIAs with each other or with a symbol amounts to comparing their print name strings byte-by-byte. They are commonly used for prompts and for representing strings stored in external databases. Making every such string into a symbol is a bad idea because the symbol table is fixed in size and very difficult to garbage collect.

Floating point numbers:

On 80386/68020: A special term: $double/4.

Files: arith.c, foreign.c, wdisp.c, lexan.c

On 88000:

64 bits in size. Due to the fact that a fence is needed at both ends of the numeric data for the garbage collector, however, four 32 bit words are needed for storing one of these doubles. Unfortunately, floating point operations will probably not be very fast (at least not as fast as in more conventional languages in which the data types are not tagged). It is possible, using the fence mechanism to create a vector of floats in which each individual float would not have to be dereferenced. A library of vector operations could be created for these objects thus lending speed to numeric intensive computations in Prolog.

- List cell:

  two consecutive 32 bit cells, a head and a tail. This object is called a cons cell in LISP. The head (or car) will occupy the cell with the smaller address. The tail (or cdr) occupies the next cell higher in memory.

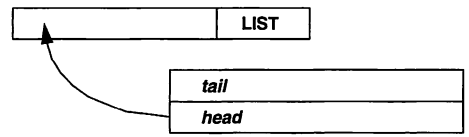


Figure 5. List Cell Layout.

- Structure (or structured term):

  N+1 cells where N is the arity of the structure. The first cell (the cell with the lowest address) is used to hold the functor and arity. The other N cells hold the arguments of the structure

where addresses increase from bottom to top.

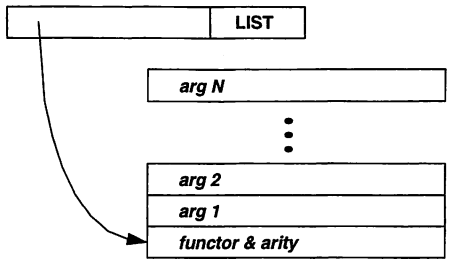


Figure 6. Structure Layout.

- Variable:

  single 32 bit cell. The tagging is arranged so that a variable may appear as part of a structure or list cell. Note: A bound variable is simply a reference to itself.

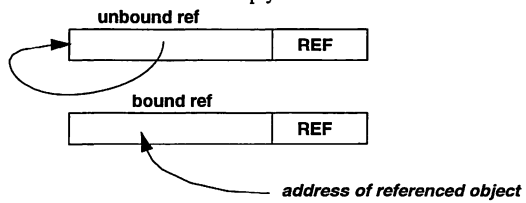


Figure 7. Layouts of Variables.

2.2. Tags

Files: mtypes.h

- 80386/68020 Tags:

  The four (4) least significant bits of the word containing the tag are used.

- 88000 Tags:

  The six (6) most significant bits of the word containing the tag are used. [We assume that our Prolog data objects will live somewhere in the region addressed by 0xfc000000 through 0xfffffffc.]

In addition, pointer objects have an additional bit available for use during garbage compaction. On the 60020 and 80386, this bit is the most significant bit. On the 88000, it is the least significant bit.

The scheme here is to use "horizontal" tagging for the data types which are most speed critical and

to vertically tag the others.

Bit Number	31.....	3	2	1	0
Reference				0	0
Structure				0	1
List	*offset from top of heap*			1	0
Integer		0	0	1	1
Symbol		0	1	1	1
Fence		1	0	1	1
UIA		1	1	1	1

Table 1: 80386/68020 Tagging Scheme

Symbolic Name	BREF	BSYM	BNUM	BUIA	BLIST	BSTRC
Bit Number	31	30	29	28	27	26
Reference	1	1	1	1	1	1
Fence	0	1	1	1	1	1
UIA	0	1	0	1	0	1
Double	0	0	1	1	0	1
Symbol	0	1	0	0	0	1
Integer	0	0	1	0	0	1
List	0	0	0	0	1	1
Structure	0	0	0	0	0	1

Table 2: 88000 Tagging Scheme

- 68020 Dereference Loop

  Files: icode1.c; slight diff if starting with known mem. ref.move.1Src, d0

```
top:        move.1   d0, a0
            and.w    #3, d0
            bne      readmode
            move.1   (a0), d0
            cmp.1    top
            bne      top
            <write mode code>
readmode:
            <read mode code>
```

- 80386 Dereference Loop (MASM syntax)

  Files: wamops.m4, imisc.c

```
drfloop:    mov      dst, src
            and      srcb,3
            jne      short grnd   (= <readmode>)
            mov      src,[dst]
            cmpl     dst,src
            jne      short drfloop
            <write mode code>
```

- 88000: Dereference A1.

```
top:        bb0      BREF,    A1,      readmode
            ld       Tmp1,    A1,      BIAS
            cmp      A1,      Tmp1,    A1
            bb0.n    eq,      A1,      top
            add      A1,      Tmp1,    0
            <write mode code>
readmode:
            <read mode code>
```

Notice that after an object has been dereferenced, a single instruction test will suffice to test for either a list or a structure. Two instructions are necessary to test for a specific type of constant. For example, to test for an integer, it is we need to make sure that the BNUM bit is set, but that the BUIA bit is not.

2.2.1. Discussion of the 88000 tagging scheme:

For pointer objects, it is assumed that the least significant bit is zero. During garbage compaction this bit is used to indicate pointer reversal.

Bit 24 is the garbage compaction reversal bit and is used only during garbage compaction. Only pointer objects need to have the reverse bit set, so integers and symbols could use this bit as part of the value field.

Bit 25 indicates whether or not the value should be treated as a reference. When it is zero, the value is a reference. When it is one, other bits in the tag must be examined to see what the object is. This bit is used to test the termination condition in the dereference loop. The dereference loop looks like this:


```
drf:        bbl      25, r2,  out; Branch if bit 25 is one
            br.n     drf      ; branch to drf after executing next instr
            ld       r2, r31, r2; follow reference chain
```

The value field of a pointer object is an offset into the Prolog data region with base in r31. If an extended addressing range is required, the scaled mode could be used on the ld instruction (giving an effective address space of 26 bits).

Bit 28 set indicates that the object is an unbound variable. Bits 0 through 23 are set so that the object references itself (with respect to the base) when the tag field is cleared.

Bit 29 set indicates that the object is a list. Bits 0 through 23 are the offset to the list.

Bit 30 set indicates a structure. Again, the value part of this object is an offset into the Prolog data area. Bit 30 also indicates a fence which is used to delimit the beginnings and ends of uninterned atoms, floats, and other sorts of objects. We can tell the difference between a fence and a structure pointer by looking at either of bits 26 or 27. These will be zero in a structure pointer and one in a fence. A fence will only be pointed at by a UIA so a single bit test will suffice after dereferencing to know whether we have a structure or not.

Bit 31 set indicates a constant of some sort. The type of constant is determined by examining bits 26 and 27. When bit 27 is clear, the value field contains a 24 bit integer (25 if we choose to use the reverse bit as part of the value). When bit 27 is set, we have a symbolic constant ; either a symbol or a UIA. If bit 26 is clear we have a symbol; otherwise we have a UIA. The value part of a symbol is a pointer into the symbol table — from which the print name may be obtained. The value part of a UIA is an offset into the Prolog data space to a fence which delimits the constant object. Fences have a length field to indicate how long the constant object is and a type field to indicate what the type of the object is. Between the two fences, we may store any bit string we choose. With this mechanism, strings, floats, and doubles may be stored as well as a lot of other useful objects. The reason for delimiting both sides of the bit string is so that the garbage compactor knows where the object begins and ends (The garbage compactor makes two sweeps of the heap, one in each direction).

3. Machine Registers

3.1. Overall View

Assignments of APM registers to real processor registers are set out in the following table.

88000 Biasing All registers that point into areas other than the code area are biased by 32K. The reason for this is that negative offsets may not be used with the ld instruction. The 32K bias gives us a way of accessing data prior to where the pointer really points.

APM			Processor Registers					
Reg	68020	VAX	386	SPARC	88K	PA	RS6K	Description
ZERO					r0			Contains 0; cannot be overwritten (88K).
RET					r1			Set up by the bsr and jsr instructions.(88K)
A1					r2			First argument - shadows stack frame.
A2					r3			Second argument - shadows stack frame.
A3					r4			Third argument - shadows stack frame.
T1					r5			Temp 1
T2					r6			Temp 2
T3					r7			Temp 3
T4					r8			Temp 4
T5					r9			Temp 5
UArg1					r10			Aux dereference register. Also holds unifier args. (Most args deref'd from S])
UArg2					r11			Second argument to the unifier
Tmp1					r12			Temp (Used for any purpose within an abstract instruction)
Tmp2					r13			Temp (Used for any purpose within an abstract instruction)
OldE					r14			E pointer for the previous environment. Used by unit clauses to return to the caller.

Table 3: APM Hardware Register Assignments

APM			Processor Registers					
Reg	68020	VAX	386	SPARC	88K	PA	RS6K	Description
CP					r15			Continuation pointer. Place to continue execution when APM equivalent of a proceed instruction is executed.
TR	A4		ECX		r16			Trail Pointer. Points at the top trail entry, if any.
H	A5		EDX		r17			Top of heap pointer. Points to the "top" of heap which grows from low to high. Actually points to the next available heap location.
Fail	A2				r18			Place to branch to on failure. Might be able to do some shallow backtracking optimizations by adding constants to this value.
SPB	D6		EDI		r19			Stack pointer backtrack point. Place to reset E and SP to upon failure.
HB	D5		ESI		r20			Heap backtrack point.
B	A3				r21			Top Backtrack frame pointer. Points at the top backtrack frame.
Safety	D7				r22			When TR-H < Safety, exception handling code is called. - includes gc, decompiler, debug hooks, interrupt hooks, and delay handling hooks.
S	A1		EBX *		r23			Read Mode Structure Pointer. Also the register from which we dereference arguments which are not in registers.
Heap Base					r24			Address of where the heap begins. Occasionally needed for testing whether a variable is on the local stack or not. May be more useful to have this as the offset from Base to where the heap starts. Comparisons could be done more directly then.
Stack Base					r25			Address used in checking for stack overflow. At present a tbnd instruction is used for checking stack overflow. (88K)

Table 3: APM Hardware Register Assignments

APM			Processor Registers					
Reg	68020	VAX	386	SPARC	88K	PA	RS6K	Description
Reserved					r26			Programming card says LINKER so we won't touch these registers. (88K)
					r27			
					r28			
					r39			
E	A6		EBP		r30			Environment pointer. Points at the stack frame from which we obtain source arguments.
SP	A7		ESP		r31			Stack pointer. Points at the "top" of the argument environment stack which grows from high to low.

Table 3: APM Hardware Register Assignments

* This register is also used by lots of other code; each patch "steals" the register....

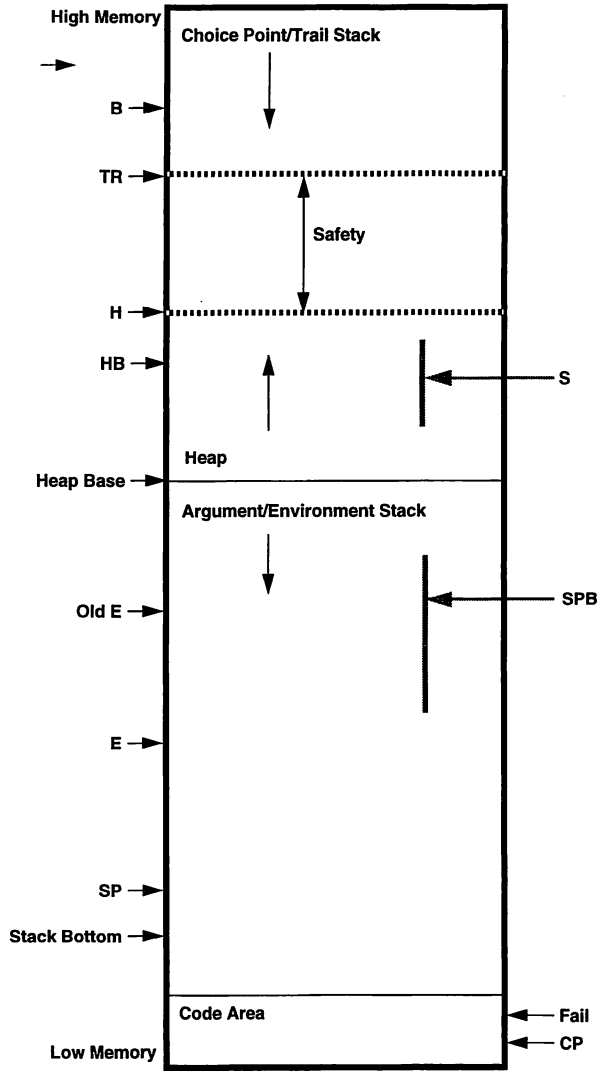


Figure 8. Memory Organization Detail, Including APM Registers.

3.2. Rationale for mappings: APM registers to processor registers

3.2.1. General considerations:

These considerations apply to the native code compiler models, and are somewhat conditioned by the fact that C is being used together with assembler as the substrate implementation language.

a. In general, one wants to put heavily used APM registers in real machine registers. The consensus on the ranking (most to least) of AMP register use is:

b. The Prolog stack pointer SP should be assigned to the same register that the processor uses for its stack pointer.

c. The Prolog environment pointer E should be assigned to the processor register which is used as the frame or environment pointer, either due to hardware considerations, or to general conventions for languages on this processor, or at least used by the C compiler adopted.

d. The register which the C compiler uses for its function return values should not be assigned to any Prolog machine register, but should be used as a scratch register.

3.2.2. 80386 Processor:

a. ECX is used by certain instructions (e.g., string move instructions) to pass a count value; So it should, if possible, not be used for an APM register. In fact, at present (4/89) it is used for the top of trail pointer TR, and EBX is used as a scratch register. This means that all instructions utilizing this approach (a count in ECX) were avoided. A better choice would be: ECX as a scratch register and EBX as TR.

b. If one decides to use multiply instructions (which were avoided because they are so slow), one would have wanted to have EAX/ECX to work together as a pair since they then could be used to receive 64 bit multiply results.

3.2.3. 68020 Processor:

A useful picture of the assignment of processor registers to APM registers:

Data regs	APM reg	Address regs	APM reg
d7	Safety	a7	SP
d6	SPB	a6	E
d5	HB	a5	H
d4	Temp (Xn-reg)	a4	TR
d3	Temp (Xn-reg)	a3	B
d2	Temp (Xn-reg)	a2	Fail
d1	Temp (Xn-reg)	a1	S & scratch
d0	scratch & deref	a0	scratch & deref

Table 4: 68020 APM Register Assignments.

Considerations leading to this:

a. a7 is always used as system stack pointer, hence assigned to SP.

b. a6 is conventionally used as the frame pointer, hence assigned to E.

c. SP,E,H,TR,B, and Fail values need at times to be used as real machine pointers (real addresses); hence they must be assigned to address registers. {If they were assigned to data registers, they would have to be moved in and out of address registers when necessary, costing in performance.}

d. SPB and HB don't need to be used as machine addresses, only as integers, so they don't need to be address registers, only data registers.

e. Safety could get away as a memory location. However, it is used for the interrupt test on every procedure entry, so it is valuable to have this in a register. {However, if good data caching is present, this might not cost too much to be in a memory location, since it is hit often enough to always stay in the cache -- this happens on the 386}.

f. The ordering of the APM registers as assigned to the 68020 registers, given the foregoing considerations, is determined by the structure of choice points and some particular 68020 instructions. The structure of a choice point is:

Previous B (B)
Next Clause (Fail)
SPB
HB

The one APM restriction on the ordering within the choice point is that B must be the uppermost entry (most significant {=highest} address). Instruction usage should make moving these values into a choice point (& out later) as fast as possible. The 68020 has a movem {move multiple} instruction which is good for moving things into the CP (tho, it turns out not to be good for moving them out). The instruction takes a 16-bit mask which specifies the registers to be moved (data first, address second), with the data registers moving to the least (lower) significant target address. So this dictated SPB and HB being placed together as a group (in the data registers), and Fail and B being placed together as a group in the address registers.

These appear to be all the 68020 considerations; the rest of the assignment is random, relative to these constraints.

3.2.4. VAX Processor:

3.2.5. 88000 Processor:

- r0 is always 0 (by the hardware) -- because moves are really adds with 0;
- r1 holds the return address for the bsr and jsr instructions;

Thus, we don't touch r0 or r1.

- r31 is the system stack pointer (hence holds SP), and r30 is the system frame pointer (hence holds E).
- Conventionally, registers are used for argument passing, r10-r16 for temporaries, and r17-r25

should be automatically saved by the caller. However this latter doesn't seem to be carried out by C or the BCS.

- The hardware and system manuals say to stay away from r26-r29.

These appear to be the only 88000 constraints.

3.2.6. SPARC Processor:

3.2.7. Precision Architecture (PA) Processor:

3.2.8. RS/6000 Processor:

4. Stack Organizations

4.1. Argument/Environment Stack

Files:

80386 - wamops.m4, winter.h, winter.c

68020 - winter.h, winter.c; assumed by icode

Stack frames on the argument/environment stack are arranged as follows

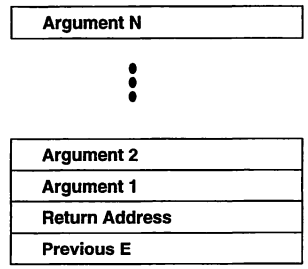


Figure 9. Argument/Environment Stack Frame Organization.

By the time any given clause which requires an environment is entered all of the slots in the frame are filled in. This differs from a previous model where only part of the frame was filled in. This also differs from the 68020 implementation in that the previous environment is filled in prior to entering a clause. In addition to the stack frame being filled in as described above at clause entry, the first through third arguments are available in A1, A2, and A3. If the clause is a unit clause, CP will contain the address to continue at and OldE will have the previous environment. E will point at the stack frame. SP will also point at the stack frame.

4.2. Backtrack (Choice Point) Frames

Files:

80386 - chpt.h, chpt.m4

68020 - chpt.h, .c

A backtrack frame consists of exactly four cells arranged as follows for the 80386 and 88000:

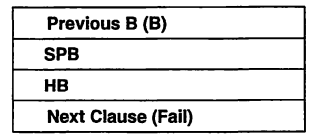


Figure 10. Backtrack Frame Organization on 386 & 88000.

On the 68020 (Sun), it looks like this:

Previous B (B)
Next Clause (Fail)
SPB
HB

Figure 11. Backtrack Frame Organization on 68020.

{The reason for the 68020 form is the interaction between the assignment of APM regs to 68020 processor regs (A/D), and the movml instruction. It seems possible that the 80386 and 88000 can be changed [easily?] so that they agree with the 68020 version.}

4.3. Try/Retry/Trust Sequences.

The choice point creation/modification/removal sequences are referred to as try/retry/trust sequences after instructions of a similar name in the APM. A **try sequence** is used in an indexing block.

Files:

80386 - choice.m4, index.c
68020 - icode2.c, rts.c

4.3.1. On the 80386:

```
__;
__; The following routines are used by clause indexing
__;
__; An indexing block of code includes patches of the form
__;
__;         call wm_try
__;         address of try clause
__;         call retry
__;         address of retry clause
__;         ..... other retrys like above
__;         call wm_trust
__;         address of trust clause
__;
__; The next_clause address in the choice point created by these routines
__; should be the return address from the call to the wm_try, wm_retry,
__; wm_trust, routines after going over the address of the clause to go to.
__;

PROC(wm_try)

            POPL    EAX                             _; pop the return address
            MOVL    OPRS(SPB,SP)                    _; move SP to SPB
            MOVL    OPRS(HB,H)                      _; move H to HB
            SUBL    OPRS(TR,IMM(chpt_SIZE))         _; Make room for chpt

            CMPL    OPRS(ECX,EDX)
            JBE     DISP(PROCADDR(bad))

            MOVL    OPRS(EBX,GVAR(wm_b_reg))        _; Prev B
            MOVL    OPRS(MADDR(chpt_B,ECX),EBX)
```



```
            MOVL    OPRS(MADDR(chpt_SPB,ECX),EDI)   _; SPB
            MOVL    OPRS(MADDR(chpt_HB,ECX),ESI)    _; HB
            LEA     OPRS(EBX,MADDR(5,EAX))          _; Next clause
            MOVL    OPRS(MADDR(ECX),EBX)
            MOVL    OPRS(GVAR(wm_b_reg),ECX)        _; set B to its new value

            JMP     REGDISP(MADDR(1,EAX))           _; execute the clause code

ENDPROC(wm_try)
```

4.3.2. On the 68020:

```
|
| wm_try
|

            .globl  _wm_try
            .proc
_wm_try:
            movl    sp@+, a0
            movl    a7, d6              | move SP to SPB
            movl    a5, d5              | move H to HB
            moveml  #0x630, a4@-        | lay down CutPt, B, E, SPB, and HB
            movl    a0@+, a6            | get address to jump to
            movl    a0, a4@-            | save next clause address
            movl    a4, a3              | set B to its new value
            jmp     a6@                 | execute the clause
```

4.3.3. On the 88000:

```
            bsr.n   _wm_tryN
            subu    TR,     TR,   16
            br.n    clause_start
            addu    Fail,   RET,  0
```

A different type of try sequence will be found in the procedure entry code for non-indexed procedures with more than one clause. This sequence looks like the following.

```
            bsr.n   _wm_tryN
            subu    TR,     TR,     16
            or      Fail,   ZERO,   next_clause_lo16
            br.n    clause_start
            oru     Fail,   Fail,   next_clause_hi16
```

Note that in both of the above sequences a subroutine call is made to _wm_tryN. This label actually refers to one of _wm_try0, wm_try1, wm_try2, or wm_try3. If for example, code for a one argument procedure is being emitted, then the _wm_try1 label would be used. If the procedure has three or more arguments, the _wm_try3 label is used. If the 88k implementation is changed to pass either more or fewer arguments in registers then the appropriate subroutines will either have to be

added or deleted. The code for the _wm_tryN subroutines:

```
_wm_try3:
                st      A3,     E,      BIAS+16

_wm_try2:
                st      A2,     E       BIAS+12

_wm_try1:
                st      A1,     E       BIAS+8

_wm_try0:
                st      CP,     E       BIAS+4
                st      OldE,   E,      BIAS
                st      B,      TR,     12
                st      SPB,    TR,     8
                st      HB,     TR,     4
                st      Fail,   TR,     0
                addu    B,      TR,     0
                addu    SPB,    E,      0
                jmp.n   RET
                addu    HB,     H,      0
```

In the above code, BIAS refers to the 32K bias needed by pointers to data objects.

Note that on the 88000, the top choice point really resides in registers. This differs from the 80386 and 68020 versions in that HB, and SPB were merely shadows of what was stored in the top choice point.

The 80386 and 68020 versions also have an extra word in the choice point frame used for storing the environment value upon backtracking. This information is now part of the argument stack frame. {IS THIS STILL TRUE???}

The sequence of instructions which unwind the trail and modify the choice point for all clauses but the first and last is called either a retry or a retry_me sequence. A retry sequence is found sandwiched amidst other choice point creation/modification/destruction sequences in an index block. A retry_me sequence is found just prior to the start of the first clause. Under certain circumstances, it will be necessary for proper execution to have the CP and OldE registers restored before entering the clause. When this is the case a different kind of retry or retry_me sequence is run.

4.4. Retry sequence formats:

4.4.1. 80386:

```
PROC(wm_retry)

            MOVL    OPRS(EAX,GVAR(wm_b_reg))        _; Get where to take
                                                    _; the stack back to
            JMP     SDISP(retry2)
                                                    _; Untrailing loop
retry1:
            MOVL    OPRS(EBX,MADDR(ECX))            _; Get var
            MOVL    OPRS(MADDR(EBX),EBX)            _; And reset it
            ADDL    OPRS(ECX,IMM(4))                _; Point to next entry
```



```
retry2:
            CMPL    OPRS(ECX,EAX)
            JNE     SDISP(retry1)

            MOVL    OPRS(EDX,ESI)                   _; reset H from HB
            POPL    EAX                             _; Get calling address
            MOVL    OPRS(ESP,EDI)                   _; Reset SP from SPB
            MOVL    OPRS(EBP,EDI)                   _; reset E from SPB
            LEA     OPRS(EBX,MADDR(5,EAX))          _; Get next clause address
            MOVL    OPRS(MADDR(ECX),EBX)            _; Put in Choice point

            JMP     REGDISP(MADDR(1,EAX))           _; execute clause code

ENDPROC(wm_retry)
```

4.4.2. 68020:

```
|
| wm_retry
|

      .globl_wm_retry
      .proc
1:    movl a4@+, a0
      movl a0, a0@
_wm_retry:
      cmpl a3, a4
      bcss 1b
      movl sp@, a0
      movl d5, a5
      movl d6, a7
      movl a3@(chpt_E), a2
      movl a0@+, a6l get address to jump to
      movl a0, a3@                          l save next clause address in chpt
      movl sp@(4), a1l put first arg in a1
      jmp  a6@                              l execute the clause
```

4.4.3. 88000:

```
            bsr.n       _wm_retryX
            addu        E,      SPB,  0
            bsr.n       clause_start
            addu        Fail,   RET,  0
```

A retry_me sequence which begins a clause doesn't need the second bsr and so looks like this:

```
            bsr.n       _wm_retryX
            addu        E,      SPB,  0
            or          Fail,   ZERO, next_clause_lo16
            oru         Fail,   Fail, next_clause_hi16
```

The wm_retryX code is either one of _wm_retry0, _wm_retry1, _wm_retry2, _wm_retry3, _wm_retry_u1, _wm_retry_u2, or _wm_retry_u3. The _uN is used with either unit clauses or clauses with only one goal. This code has the following form and may also be found in the souce file chpt.88k:

```
_wm_retry_u3:
          ld   A3,    e,     BIAS+0x10   ; reload A3

_wm_retry_u2:
          ld   A2,    e,     BIAS+0xC    ; reload A2

_wm_retry_u1:
          ld   A1,    e,     BIAS+0x8    ; reload A1

_wm_retry_u0:
          ld   cp,    e,     BIAS+4
          ld   OldE,  e,     BIAS
          subu tmp1,  b,     tr
          bcnd eq0,   tmp1,  untr_done_r
          br.n untr_r
          subu tmp1,  tmp1,  4


_wm_retry3:
          ld   A3,    e,     BIAS+0x10   ; reload A3

_wm_retry2:
          ld   A2,    e,     BIAS+0xC    ; reload A2

_wm_retry1:
          ld   A1,    e,     BIAS+0x8    ; reload A1

_wm_retry0:
          subu tmp1,  b,     tr
          bcnd eq0,   tmp1,  untr_done_r
          subu tmp1,  tmp1,  4
untr_r:
          ld   tmp2,  tr,    tmp1
          subu tmp1,  tmp1,  4
          st   tmp2,  tmp2,  BIAS
          bcnd ge0,   tmp1,  untr_r
          or   tr,    b,     0
untr_done_r:
          addu sp,    spb,   0
          jmp.nrl
          addu h,     hb,    0
```

4.5. Trust code:

4.5.1. 80386:


```
PROC(wm_trust)

          MOVL   OPRS(EAX,GVAR(wm_b_reg))  _; Get where to take
                                           _; the stack back to
          JMP    SDISP(trust2)
                                           _; Untrailing loop
trust1:
          MOVL   OPRS(EBX,MADDR(ECX))      _; Get var
          MOVL   OPRS(MADDR(EBX),EBX)      _; And reset it
          ADDL   OPRS(ECX,IMM(4))          _; Point to next entry
trust2:
          CMPL   OPRS(ECX,EAX)
          JNE    SDISP(trust1)

          MOVL   OPRS(EDX,ESI)             _; Reset H
          POPL   EBX                       _; And get return address
          MOVL   OPRS(ESP,EDI)                   _; Set SP from SPB
          MOVL   OPRS(EBP,EDI)                   _; Set E from SPB
          MOVL   OPRS(EAX,MADDR(chpt_B,ECX))_; Reset B
          MOVL   OPRS(GVAR(wm_b_reg),EAX)
          ADDL   OPRS(TR,IMM(chpt_SIZE))         _; Remove choice point
          MOVL   OPRS(ESI,MADDR(chpt_HB,EAX))_; Reset HB
          MOVL   OPRS(EDI,MADDR(chpt_SPB,EAX))_; set SPB
          ANDL   OPRS(EDI,IMM(HEX(fffffffe)))_; nuke compaction bit

          JMP    REGDISP(MADDR(1,EBX))           _; execute clause code

ENDPROC(wm_trust)
```

4.5.2. 68020:

```
|
| wm_trust
|

          .globl _wm_trust
          .proc
1:        movl   a4@+, a0
          movl   a0, a0@
_wm_trust:
          cmpl   a3, a4
          bcss   1b
          movl   sp@+, a0
          movl   d5, a5
          movl   d6, a7
          lea    a3@(chpt_E), a4
          movl   a4@+, a2
          movl   a4@+, a3
          movl   a3@(chpt_HB), d5          | set HB
          movl   a3@(chpt_SPB), d6         | set SPB
          andw   #0xfffc, d6               | nuke compaction bit
          movl   sp@(4), a1                | put first arg in a1
          movl   a0@, a0                   | get address to jump to
          jmp    a0@                       | jump there
```

4.5.3. 88000:

branch.

5.1.3. Proceed sequence

The following sequence are used to return from a procedure:

80386:

68020:

88000:

```
        br.n    CP
        add     E, OldE, 0
```

5.1.4. Procedure entry points

Files:

80386 - iindex.c {also module.c, call.m4, int.m4}
68020 - icode2.c, wintcode.c

A procedure entry consists of some data describing the procedure (such as the module, procedure id, and arity). It also has two entry points. The entry points into the 88000 code below are call_entry, and exec_entry. call_entry is the target of a bsr or jsr for running a goal which is not the last in a clause. exec_entry is the target of the last goal in a clause. {Is this the structure for the 80386 and 68020 too???}

80386:

68020:

88000:

```
        ov:         bsr     wm_overflowN
                    br      code
        call_entry:
                    add     CP, RET, gc_info_size
                    add     E, ZERO, SP
        exec_entry:
                    sub     T1, TR, H
                    cmp     T1, T1, Safety
                    bb1     #lo, T1, ov
        code:
                    More procedure entry code.
```

The instructions which follow code1 will depend on what the number and types of clauses which constitute the procedure. If the procedure consists of more than one clause, or has more than one clause but needs the environment information filled in, the stack frame will be completed. That is

to say that CP, OldE, and A1 thru A3 will be written onto the stack. Instructions which cause choice point creation may also be present if the procedure has more than one clause.

The label, wm_overflowN, is actually wm_overflow0, wm_overflow1, wm_overflow2, or wm_overflow3 depending upon the number of arguments to the procedure. This code is responsible for handling the exceptional conditions including garbage compaction, decompilation, debugging, delay handling, and interrupts.

As mentioned before, gc_info_size is the number of bytes (a multiple of four) following the call sequence needed for garbage collection information. At present this value is 8. The information immediately following the call sequence is two addu instructions with r0 as the destination register. The unsigned sixteen bit constant in the first instruction is the argument usage mask. The unsigned sixteen bit constant in the second instruction is the number of words back to the beginning of the clause from the instruction itself. This information is used by the code space garbage collector. Finally, the number of arguments (10 bits worth) is split between the source registers of the two instructions. The source register of the first addu represents the low 5 bits of the number of arguments while the second represents the upper five bits. Since these data are represented as valid instructions, the disassembler will have no problem with them. Also, since r0 is the destination, these instructions are essentially nops and nothing awful will happen should they be accidentally executed.

5.2. Parameter Passing

There are two ways to pass parameters; on the stack or in registers. Both methods have their advantages and disadvantages.

Passing arguments in registers makes determinate programs run fast. Setting the arguments up and matching them require no memory accesses to get at the top level of the argument. There are several disadvantages, however. The first disadvantage is that there are a limited number of registers which may be used for passing parameters; some place else is needed to put the arguments that cannot be put into registers. The second disadvantage is that the registers must be saved when a choice point is created and restored when failure occurs. If failure occurs quite early in the head of each clause, a lot of unnecessary loads will take place. It is possible to arrange the code so that arguments will be restored out of memory only when necessary but at the cost of either compiling the head twice or placing tests in the head matching code to see whether or not an argument needs to be loaded from the choice point. Neither option is attractive; the first consumes too much space, the second too much time.

When arguments are passed on a stack, determinate programs slow down somewhat. It is necessary to do one memory write (above the register case) to set up an argument. Similarly, an extra memory read will be required for each argument in the head matching phase. When creating a choice point, however, the arguments need not be saved; only a pointer to the argument vector needs to be saved in the choice point. When failure occurs the argument vector pointer is restored. Memory accesses occur only when it is necessary to fetch an argument. So if failure occurs early in the head, the stack-based approach can offer superior performance over the register-based approach. Consequently, the stack-based approach will make highly non-determinate programs run faster.

The stack-based approach for passing parameters offers another advantage over the register-based approach. Consider the following clause:

```
p(X,Y,Z) :- q(X,Y,Z), r(X,Y,Z).
```

The register-based approach will create an environment and save the parameters X, Y, and Z in this environment. It will need these parameters to set up the last goal. The stack-based approach allows the parameter vector to be part of the environment block. Thus, the arguments don't need to be copied since they are already part of the environment. They do, however need to be moved to set up the call for q. So the costs appear to be the same. But what about nondeterminate programs? The register-based approach will copy the registers into the choice point and then copy the same registers into the environment. The stack-based approach won't do either, but it will copy the arguments in setting up the arguments for q. Passing the parameters in registers means that we do an extra set of memory writes.

It would be nice to get the best of both methods for passing parameters. Before proceeding, let us state a hypothesis that we believe to be true, but for which we have very little empirical proof:

> When failure occurs in the head, it usually does so in matching one of the first several arguments.

The reasons that we believe this statement to be true are:

a. Prolog predicates can be made to run forwards, backwards, and sideways. But there is usually a "direction" that the programmer had in mind when he wrote the predicate. Put more succinctly: The programmer writes predicates with a certain I/O behavior for the arguments in mind.

b. Given a), programmers usually arrange the first several arguments to be input arguments and the later ones to be output arguments. This is especially true of the first argument on which indexing is often generated.

The compromise solution passes the first k (for k small) arguments in registers. The rest are passed on the stack. Moreover, the k positions in the stack where the arguments would have been stored in the stack model are left open. These are used for storing the arguments when a choice point is created. This means that a clause which was entered through some choice point code will not have to save top level variables in the environment. What should we make k? The author feels that an integer between two and four (inclusive) will offer good performance. The compiler will be parameterized so that an optimal value for k may be found for a large collection of typical Prolog programs. Let us assume that k=3 for the rest of this discussion. A typical stack frame is depicted in figure 2. Arguments one through three are passed in the registers A1, A2, and A3. Similarly, the continuation pointer CP, and environment pointer E are passed in the registers rather than on the stack. These stack positions will be filled in when either a choice point is created or when a clause requiring and environment is run in a determinate manner.

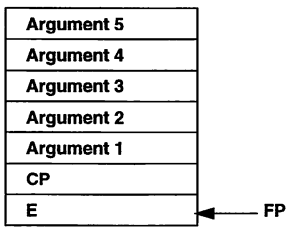


Figure 12. A stack frame with five arguments.

Only arguments four and five are actually stored on the stack.

On the 80386 and 68020, k = 0.

6. Code Area Organization

Files:nametbl.c, wintcode.h, wintcode.c

80386 - iindex.c

68020 - icode2.c

7. Symbol Table, Name Table, & Relatives

Symbol table:

Files: parser.h, parser.c, tokens.h, tokini

80386 - also, symbols.m4 lexinit + the awk stuff

Name table: as in ¶6; also int.m4, module.c, call.m4

8. Abstract Machine Instructions

Special Emphasis on 88K - to be changed

§8.1.Control Instructions

addto **SP, num**

SP <- SP + num

call **P/A**

OldE <- E
PC <- call_entry(P/A)

Notes: The call_entry is taken in be in the current module as determined by the compiler. It keeps track of what it believes to be the current module in a (compiler) global variable. In the present 88K system (and the 386 & Sun ??), the call_entry is actually an entry in the name table, effectively a machine pointer to the code for P/A (it may really be an offset relative to the current value of the PC).

execute **P/A**

E <- SP
PC <- exec_entry(P/A)

Notes: The exec_entry is taken in be in the current module as determined by the compiler. It keeps track of what it believes to be the current module in a (compiler) global variable. In the present 88K system (and the 386 & Sun ??), exec_entry gives the address of the exec entry point for P/A in the procedure entry table.

Procedure Entry Points -- 88K version:

```
overflow:bsroverflow_check
              br    code
call_entry:addCP, RET, gc_info_size
              add   E, SP, 0    % E <- SP
exec_entry:subT1, TR, .....
              cmp   T1, T1, Safety
              bbl   HB, T1, overflow
code: (???)
```

This is organized as it is on the 88K due to usage of the delay slot (.n) optimization. The full abstract spec should be set up without any such accounting for delay options, etc., but annotations should make it clear how the optimized form is obtained.

allocate **size**

SP <- SP - size

allocate1 **size**

SP <- SP - size

Notes: This allocate instruction is for use with one-goal clauses, when the clause has become determinate, or the last goal, when we don't need the newly allocated space, because we are reusing old space. Need details of what the compiler is doing here.

endallocate1

Sets up the determinate clause entry point.

deallocate1

set dp3 to 0; On the 88K, this is a noop at run-time, but as indicated, the compiler sets the dp3 patch to 0. [See below, after the deallocat4 instruction].

deallocate2 **size1**

if (SPB > E)
    PC <- dp2
else
    SP <- SPB - size1

deallocate3

PC <- dp3
define_label dp2 [this is a condition the compiler fulfills]

deallocate4 **size2**

SP <- E - size2
define_label dp3 if used [this is a condition the compiler fulfills]

The deallocate1 - deallocate4 instructions break up the old single deallocate instruction. In effect, for a full deallocate, we would get four patches of code, corresponding to these four instructions. On the 88K, this is what it looks like:

```
1:
2:  cmp   TMP1, SPB, E% are we determinate?
    bbl   hi, TMP1, p2
    subu  SP, SPB, size1% non-determinate
          Determinate case:  code moving old args to new args (those
          which stay the same) goes here
3:  br    p3
          p2:
          Maybe other code here
4:  subu  SP, E, size2
```

size1 = # of args required by the next call (which is an execute)

size2 = indicates amount by how much to modify the current stack to leave correct space for the new stack frame (which is overwriting the old one).

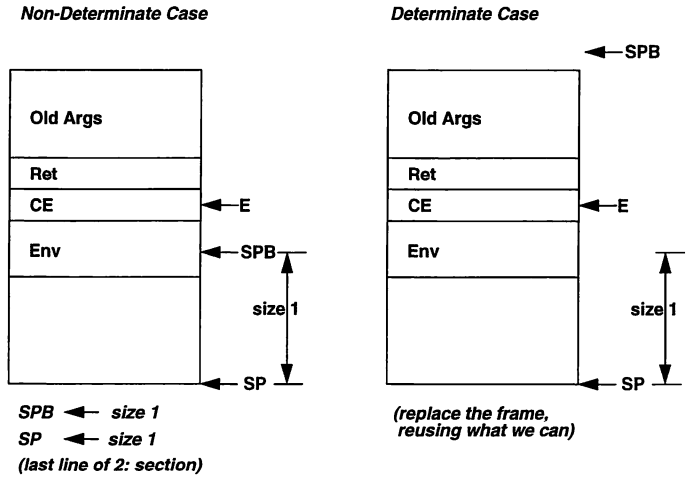


Figure 13. The 88000 Stack Frames.

---

trim **size1, size2**

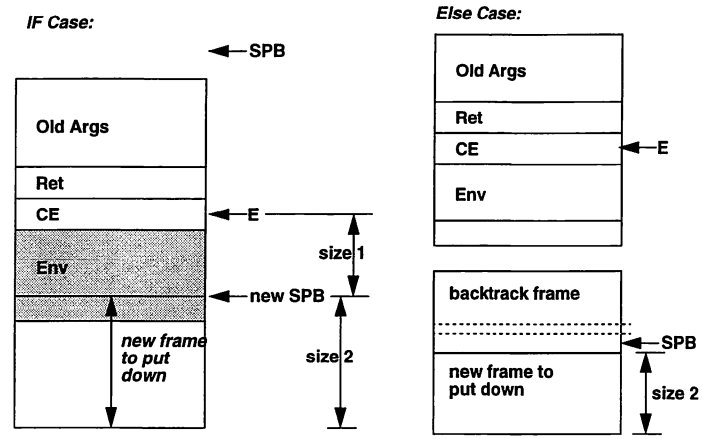
trim1 (determinate for sure case):
SP <- E - (size1 + size2)

trim2 (otherwise):
TMP1 <- SPB
if (SPB >= E)
TMP<- E - size1
SP <- TMP1 - size2

more abstract (both cases):
if (SPB => E)
SP <- E - (size1 + size2)
else
SP <- E - size2

Notes: Which trim instruction the compiler emits is determined by a boolean flag which the compiler maintains.

size1 = size of the next frame to be put down
size2 = size the (part of) the present environment to leave in place



The 88K code:

```
cmp   S, SPB, E
bb1.n lo, S, p
addu  TMP1, SPB, 0
subu  TMP1, E, size1
p:subuSP, TMP1, size2
```

---

proceed

E <- OldE
PC <- CP

---

deallocate_proceed

CP <- env_CP(E)
E <- env_E(E)
PC <- CP

Notes: Usually used at the result of a cut as the last goal.

8.1. get Instructions

move **Src, Dst**

```
Dst   <-  Src
```

Notes: g_var and put_value are implemented directly by move.

---

get_value **Arg1, Arg2**

Unify Arg1 and Arg2

Notes: Arg1 and Arg2 are not explicitly dereferenced before being passed to the unifier -- dereferencing is left as the unifier's problem.

---

get_list **Src**

```
writemode
S     <-  deref(Src)
if nonvar(S)
      PC <-  glist_readmode; setting PC causes jump -- so no else
val(S) <-  makelist(H)
trail_if_nec(S)

readmode
glist_readmode:
if list(S)
      fail
determinate entry code here
S     <-  striptag(S)    ; make S point at real car of list
```

---

get_structure **f/a, Src**

```
writemode
S     <-  deref(Src)
if nonvar(S)
      PC <-  gstruct_readmode
val(S) <-  make_struct(H)
trail_if_nec(S)
val(H) <-  makefunctor(f,a)
H     <-  next_loc(H)
readmode
gstruct_readmode:
if struct(S)
      fail
if (functor(S) =\= makefunctor(f,a))
      fail
S     <-  firstarg(S)    ; make S point at real first arg
```

---

get_int **Int, Src**


```
TMP   <-  deref(Src)
if nonvar(TMP)
      if int(TMP) & val(TMP) = makeint(Int)
            continue
      else
            fail
else
      trail_if_nec(TMP)
      val(TMP)<-makeint(Int)
```

---

get_sym **Sym, Src**

```
TMP   <-  deref(Src)
if nonvar(TMP)
      if sym(TMP) & val(TMP) = makesym(Sym)
            continue
      else if uia(TMP) & symname(Sym) = uianame(val(TMP))
            continue
      else
            fail
else
      trail_if_nec(TMP)
      val(TMP)<-makesym(Sym)
```

---

get_uia **UIAstr, Src**

```
TMP   <-  deref(Src)
if nonvar(TMP)
      if sym(TMP) & symname(val(TMP)) = UIAstr
            continue
      else if uia(TMP) & uianame(val(TMP)) = UIAstr
            continue
      else
            fail
else
      trail_if_nec(TMP)
      val(TMP)<-makeuia(UIAstr)
```

8.2. unify Instructions

unify_void

unify_value Src

```
readmode:
Unify val(S) with Src
S      <-  next_loc(S)
writemode:
val(H) <-  Src
H      <-  next_loc(H)
```

Notes: Src is usually a register (Ai or Ti), but can be a memory reference (which ought to be to the heap, but no checks are made).

unify_variable Dst

```
readmode:
Dst    <-  val(S)
S      <-  next_loc(S)
writemode:
if (Dst is an env var & occurs in body)
        trail_if_nec(Dst)
Dst    <-  val(H)<-makevar(H)
H      <-  next_loc(H)
```

Notes: Trailing above is not a priori necessary, but is performed to make GC work right.

unify_int Int

```
readmode:
TMP  <-  deref(S)
if nonvar(TMP)
        if int(TMP) & val(TMP) = makeint(INT)
            continue
        else
            fail
else
        trail_if_nec(TMP)
        val(TMP)<-makeint(Int)
S      <-  next_loc(S)
writemode:
val(H) <-  makeint(Int)
H      <-  next_loc(H)
```

Notes: Int is a real machine integer

unify_sym Sym

```
readmode:
TMP  <-  deref(S)
if nonvar(TMP)
        if sym(TMP) & val(TMP) = makesym(Sym)
            continue
        else if uia(TMP) & symname(val(Sym)) = uianame(val(TMP))
            continue
        else
            fail
else
        trail_if_nec(TMP)
        val(TMP)<-makesym(Sym)
        S    <-  next_loc(S)
writemode:
val(H) <-  makesym(Sym)
H      <-  next_loc(H)
```

Notes: Sym is the internal abstract representation of the symbol (i.e., in the present version, an index into the symbol (name) table, or, in the new version, the address of the symbol).

unify_local_value Src

```
readmode:
unify val(S) with Src
S      <-  next_loc(S)
writemode:
TMP  <-  deref(Src)
if var(TMP) & SPB < TMP < HeapBase  % target is an env. var prot.
        trail(TMP)                  % by a choice pt.
        val(H)<-val(TMP)<-makevar(H)
else
        val(H)<-TMP
H      <-  next_loc(H)
```

8.3. put Instructions

put_unsafe Src, Dst

```
TMP  <-  deref(Src)
if var(TMP) & TMP < SPB% not prot. by ch. pt.
        trail(TMP)
        val(TMP)<-val(H)makevar(H)
        Dst <-  TMP
        H    <- next_loc(H)
else
```

Dst <- TMP

---

put_int **Int, Dst**

Dst <- makeint(Int)

---

put_sym **Sym, Dst**

Dst <- makesym(Sym)

---

put_uia **UIAstr, Dst**

Dst <- makeuia(UIAstr)

---

put_yvar **ESrc, Dst**

ESrc <- makevar(ESrc)*% installs unbound var*
Dst <- Src

---

init_yvar1 **EDst**

TMP <- EDst<- makevar(EDst)
TMP <- next_loc(TMP)

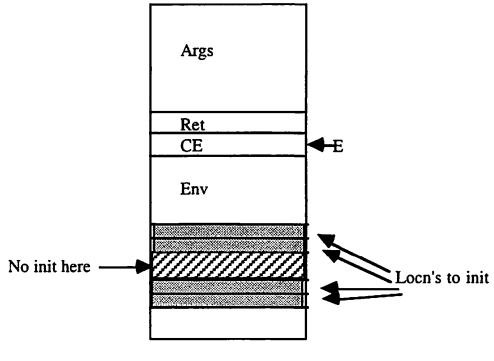
Notes: On the 88K, grab S & use it for TMP; this instr is always followed by at least one init_yvar2, hence its funny organization -- they fit together.

---

init_yvar2 **Incr**

TMP <- incr_loc(TMP, Incr)
val(TMP)<-makevar(TMP)

Notes: next_loc(X) = incr_loc(X, 1)



These vars need to be initialized for GC purposes.
Post-increment addressing (Sun) version:

```
lea-28(E),S
move.1S, (S)+
move.1S, (S)+
addq#4, S
move.1S, (S)+
move.1S, (S)+
```

---

put_xvar **Dst**

Dst <- val(H)<- makevar(H)
H <- next_loc(H)

---

put_list **Dst**

Dst <- makelist(H)

---

put_struct **f/a, Dst**

Dst <- makestruct(H)
functor(H)<-makefunctor(f,a)
H <- incr_loc(H, functorsize(f,a))

Notes: In the present version(s), functorsize(f,a) = 1, but it may become 2.

---

end_structure

Terminates structure building; may be a no-op; On the 88K, advances H (<- next_loc(H)) Basically, should form brackets with structure building initiators such as put_struct or put_list; General format:

put_structput_list
unify....unify....
unify....unify....
unify....endstruct
unify....
endstruct

8.4. Cut Instructions

---

do_cut **CutPt**

performcut(CutPt)

---

cut_proceed **CutPt**

E <- OldE
performcut(CutPt)
PC <- CP

Notes: The first two instructions here are really order independent. The indicated order, however,

makes the 88K optimization make sense, namely that instead of a jsr to performcut, one does a jump to performcut. The latter guarantees a branch to CP when done. Here is the 88K code: (using delay slot):

```
addu UArg1, E, 0
addu E, OldE, 0
br.n wm_docut
addu RET, CP, 0
```

---

deallocate_cut_proceedCutPt

```
performcut(CutPt)
CP    <-  env_CP(E)
E     <-  env_E(E)
PC    <-  CP
```

---

performcut -- *Specification*

```
docut:if Safety < 0
            goto cutexception

docut2:if SPB > UArg1 (= CutPt)
            goto cutdone
SPB         <-      cp_SPB(B)
cut_top:if SPB > UArg1
            goto cut_restore
B           <-      cp_B(B)
SPB         <-      cp_SPB(B)
goto cut_top
cut_restore:HB<-    cp_HB(B)
```

MORE TO COME HERE

Here is the 88K code:

```
_wm_docut:

bcnd  lt0,    Safety,@cutexception; exception?

;
; Code for cut operation (without exception test)
; --called with bsr  (i.e. return is in r1)
;
; The environment to cut to is passed through UArg1
;

global _wm_docut2
_wm_docut2:

cmp   tmp1, spb,  UArg1       ; determinate?
bb1   hi,   tmp1, @cut_done; nothing to do

;
```



```
; Must be less than, unwind choice points.
;
ld    spb,  b,    0x8         ; Choice Point spb

@cut_top:

clr   spb,  spb,  1<0>        ; clear compaction bit
cmp   tmp1, spb,  UArg1 ;
bb1   hi,   tmp1, @cut_restore;

ld    b,    b,    0xC         ; Prev Choice Point
ld    spb,  b,    0x8         ; Choice Point spb

br    @cut_top

;
; Here is where the top choice point is loaded into the registers.
;    --  b points to last choice point to cut
;
@cut_restore:

ld    hb,   b,    0x4         ; Choice Point hb
ld    Fail, b,    0x0         ; NextClause
addu  tmp1, tr,   0           ;
addu  tr,   b,    0x10        ;
subu  tmp2, b,    4           ;
ld    b,    b,    0xC         ; Prev Ch Pt

;
; Run down the trail and retrail those items that require it.
; --tmp1 is OldTr, tmp2 is trail runner, tmp3 gets entries
;
@ctop:
cmp   tmp4, tmp2, tmp1        ; runner < OldTr?
bb1   lo,   tmp4, @cut_done;

ld    tmp3, tmp2, 0           ;

cmp   tmp4, tmp3, tmp2        ;
bb0.n lo,   tmp4, @arnd ;
subu  tmp2, tmp2, 4           ;

cmp   tmp4, tmp3, spb         ;
bb1   lo,   tmp4, @ctop       ;
cmp   tmp4, tmp3, hb          ;
bb1   hi,   tmp4, @ctop       ;

subu  tr,   tr,   4           ; keep the entry
br.n  @ctop                   ;
st    tmp3, tr,   0           ;

@arnd:
br.n  @ctop                   ;
subu  tmp2, tmp2, 0xC         ; skip over ch pt

@cut_done:
jmp   r1                      ; return

#endif
```

9. Compiler Organization

Files:

compile.h, compile.c, varproc.h, varproc.c -- the front end behind the parser;

icode*.* (68020) or i*.* (80386) -- compiler backend

indexing in index.c

9.1. Clause Compilation

9.1.1. Unit Clauses

A unit clause is a clause with no goals. The code needed to implement a unit clause consists of head matching code followed by an instruction which jumps to the current continuation pointer. This may be implemented in the M78000 by the single instruction:

jmpCP

Notice that the frame pointer is not reset when leaving a unit clause. This action will be performed by the code to which the unit clause returns control.

9.1.2. Clauses with one goal

The code for a clause with one goal starts with FP pointing at the source arguments. It must set up FPD (the destination frame pointer), match the head code (setting up arguments as convenient), and set up the arguments for the first goal. FP is used as the base register for doing the head matching. FPD is used as the base pointer for setting up the arguments in the goal. When execution is determinate, the destination frame and the source frame will overlap so care must be taken not to write an argument into the destination frame until the argument has been consumed by the head matching code. When execution is non-determinate, a new frame is allocated next to the current frame. Stack-based arguments which live in the same location in the determinate case (and hence do not need to be copied) need to be copied in the non-determinate case. After the arguments have been set up for the goal a branch instruction is used to transfer control to the appropriate entry point.

In order to set up FPD appropriately, the clause may be entered through one of two points. The first one is entered when the procedure is non-determinate. The second one is entered when the clause is determinate. Let m be the number of head arguments and n the number of goal arguments in the following schematic of the clause code.

```
ndentry:br.ncommon
addFPD, FP, n
detentry:addFPD, FP, n-m
common:
; Head matching and argument set up code goes
; here.
br.nentry point for procedure in first goal
addFP, FPD
```

Notice that the frame pointer is set to the destination frame pointer prior to branching to the first goal.

9.1.3. Clauses with multiple goals

An environment must be maintained for clauses with multiple goals. The continuation pointer, and

the environment pointer can be changed in the process of running the goals in the clause. These values must be saved and restored prior to executing the last goal. The last goal is treated in the same fashion as clauses with only one goal; it is branched to directly. The intermediate goals are run by performing a subroutine call. Prior to executing the first goal, however, the clause must make sure that the continuation pointer and environment pointer are saved in the stack. The environment pointer is then updated to point at the current environment. This situation is shown in figure 3. Note that the environment pointer does not point at the source frame pointer, but at the end of the environment. It would perhaps be more convenient for the environment pointer to point at the same place that FP does, but it is not possible

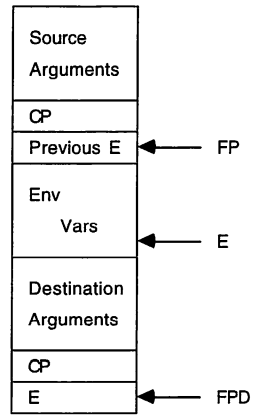


Figure 3: Argument / Environment Stack prior to calling the first goal in a mult-goal clause. CP, E, and the first three arguments are not filled in on the stack yet.

Prior to executing a goal in a clause, the stack point must be positioned so that the arguments may be set up for that goal. In the case of a first goal in a mult-goal clause, the stack pointer is rather easy to position. This is the arrangement depicted in figure 4 above. But it is necessary to do a comparison to position the stack pointer in other situations. In a clause with only one goal, it is necessary to see whether the procedure is determinate or not. This condition may be checked by comparing SP with SPB. If SP and SPB are equal, the procedure is non-determinate (a choice point was created for it) and the argument block may not be reused. Otherwise, the arguments for the current procedure may be over-written. This is also the case for the last goal in a clause with more than one goal. If the head and the last goal have the same number of arguments, it is not necessary to move the return address. It is also unnecessary to move arguments which will live in the same memory location on the stack in the determinate situation. In fact, prior to argument setup for the nondeterminate case, it is useful to copy the arguments which live in the same memory locations (in the determinate case) so that argument setup may share as much code as possible between the non-determinate and determinate cases.

When executing one of the middle goals in a clause with more than 2 goals, it is possible that we can trim some of the environment away and reuse it for the argument block in the next call. We can do this trimming if we compare E and SPB and find that E is less than or equal to SPB. This

means that the clause may be non-determinate, but not because one of the previous goals in the clause was non-determinate.

9.1.4. Environments

As discussed in the previous section, an environment is created in a clause with multiple goals. The entire environment for such a clause may be considered to be the parameter block, the return address, the previous environment pointer, and the locals.

As its name suggests, the parameter block contains the parameters. These are always tagged objects and may be constants, pointers to a local variable in a previous environment, or heap pointers:either to structured objects (lists and structured terms) or variables. A variable may not live in an argument position. This possibility was considered in the design of the model, but involves too much hair for serious consideration.

The return address is the location to continue execution at when a unit clause is finally executed. The environment pointer is the value of E prior to creating the current environment. E will be restored to its previous state by code corresponding to the deallocate instruction in the APM just prior to setting up the arguments for the last goal.

The locals are used for two purposes. The first is to keep a handle on variables whose first (compiled) occurrence is in structure. These variables will be living on the heap and are never unsafe. The second purpose of the locals is to serve as genuine variables. When used in this fashion, the first compiled occurrence will initialize the variable to unbound. If the variable later appears in structure, it will need to be dereferenced. If it dereferences to an environment variable, a new variable needs to be created on the heap and the environment variable bound to this heap location. Trailing may have to be done. This trail check will take only one comparison instead of the usual two. In the last goal in which one of the locals appears all occurrences are considered to be unsafe. This is due to the fact that the variable is trimmed away if the procedure is determinate.

The classic append program in Prolog is:

```
append([],L,L).
append([H|T],L,[H|TL]) :- append(T,L,TL).
```

These clauses are used in the standard nrev benchmark used to compute LIPS (Logical Inferences Per Second). Of the two clauses, the second is run most frequently for long lists and plays the biggest role in the nrev bencmark.

The following code is M78000 code for the procedure entry point and the second clause for append/3. Although this is hand written code, we feel that our compiler will be able to generate it. The register names used are the symbolic register names from our M78000 Prolog design document. Their are two entry points. append_c is entered when append is called (bsr). append_e is entered when append is executed (br).

```
1   append_o:   jsr     Overflow                ; handle gc and decompilation
2               br      append_e
3   append_c:   add     CP, R1, #cpinfo         ; set continuation pointer
4   append_e:   sub     Tmp, TR, H              ; check closeness of TR & H
5               cmp     Tmp, Tmp, Safety        ; compare with safety
6               bb1     #le, Tmp, append_o ; branch on exception
7   drf:        bb1     25, A1, switch          ; branch if A1 is ground
8               br.n    drf                     ; follow reference chain
9               ld      A1, Base, A1            ;
10  switch:     bb1.n   29, A1, gotlist1        ; branch if object is a list
11              extu    S, A1, 24<0>            ; put list pointer in S
```



```
12              bb1     31, A1, gotconst        ; branch if object constant
```

The above constitutes most of the entry code. Missing is a branch instruction to some choice point creation code (try_me_else) and a jump to failure when neither a list, constant or variable is seen. The label gotconst does not appear anywhere in this code. It would be in the first clause after the initial dereference loop. The label gotlist appears in the code for the second clause below.

```
13  drf1:bb1    25, A1, out1    ; branch if A1 is ground
14   br.n                       ; follow reference chain
15   ld         A1, Base, A1
16  out1:bb1.n  29, A1, gotlist1 ; branch if object is a list
17   extu       S, A1, 24<0>    ; put pointer into S
18   bb1.n      28, A1, gotvar1 ; branch if object is var
19   or.u       A1, H, 0x2200   ; put list pointer to H in A1
20   jmp        Fail            ; Jump to failure point
21  gotvar1:st  A1, Base, S     ; bind variable (S) to list
22   cmp        Tmp, S, HB      ; see if S ≥ HB
23   bb1        #ge, Tmp, notr1 ; branch if so (do not trail)
24   cmp        Tmp, S, FPB     ; see if S < FPB
25   bb1        #le, Tmp, notr1 ; branch if so
26   sub        TR, TR, 4       ; decrement trail pointer
27   st         S, TR, Base     ; trail the variable
28  notr1:or.u  Tmp, H, 0x1200; put unbound ptr to H in Tmp
29   st         Tmp, H, Base    ; store this temp ptr
30   add        T1, H, 0        ; put H in T1
31   add        H, H, 4         ; advance heap pointer
32   or.u       Tmp, H, 0x1200; put unbound ptr to H in Tmp
33   st         Tmp, H, Base    ; store Tmp at top of heap
34   add        A1, H, 0        ; move H to A1
35   br.n       drf2            ; process second argument
36   add        H, H, 4         ; but first advance H
37  gotlist1:ld T1, Base, S     ; put head of list in T1
38   add        S, S, 4         ; advance write mode ptr
39   ld         A1, Base, S     ; put tail of list in A1
40  drf2:bb1    25, A3, out2    ; branch if at end of chain
41   br.n       drf2            ; follow chain by loading A3
42   ld         A3, Base, A3    ;  into A3 and looping
43  out2:bb1.n  29, A3, gotlist2 ; branch if A3 is list ptr
44   extu       S, A1, 24<0>    ; put offset part of ptr in S
45   bb1.n      28, A3, gotvar2 ; branch if A3 is a variable
46   or.u       A3, H, 0x2200   ; put list pointer in A3
47   jmp        Fail            ; fail if A3 not list or var
48  gotvar2:st  A3, Base, S     ; bind the variable
49   cmp        Tmp, S, HB      ; see if variable needs to
50   bb1        #ge, Tmp, notr2 ;  be trailed - branch if not
51   cmp        Tmp, S, FPB     ; compare some more  - branch
52   bb1        #le, Tmp, notr2 ;  if trailing is not needed
53   sub        TR, TR, 4       ; push a new
54   st         S, TR, Base     ;     trail entry
55 notr2:stT1, H, Base ; put down head of first list
56   add        H, H, 4         ; advance heap pointer
57   or.u       Tmp, H, 0x12    ; put var ptr to H in Tmp
58   st         Tmp,H,Base      ; put down the var ptr
59   add        A3, H, 0        ; put ref to var in A3
60   br.n       append/3        ; branch to append again
61   add        H, H, 4         ;   but first advance H
62  gotlist2:ld U1, Base, S     ; set up unifier arguments
```

```
63    add          U2, T1, 0      ;
64    jsr.n        Unify          ; unify heads of 1st and 2nd
65    add          S, S, 4        ; update read mode ptr
66    ld           A3, Base, S    ; put tail in A3
67    br.nappend/3 ; run append again
```

9.1.5. Speed Estimates

Our speed estimate for LIPS is based on the code for the second clause above. The second clause is run most often in running naive reverse. The first argument is a list, the second argument is also a list (though this doesn't matter), and the third argument is a variable. Assuming each reference chain is zero (not an unreasonable assumption), the following lines are run in the order that they appear per iteration. Each line (instruction) takes one clock cycle unless followed by a number in parentheses.

4, 5, 6, 7(2), 10, 11, 37(L), 38, 39(L), 40, 41, 42(L), 40(2), 43, 44, 45, 46, 48, 49, 50(2), 55, 56, 57, 58, 59, 60, 61

The number of clock cycles per iteration is 27+3*L where L is the length of time taken by a ld instruction. Assuming a 20 MHz clock, the formula for LIPS is given by:

$$\text{LIPS} = \frac{20{,}000{,}000\ \frac{\text{cycles}}{\text{second}}}{(27+3L)\frac{\text{cycles}}{\text{inference}}} = \frac{20{,}000{,}000}{27+3L}\ \frac{\text{inferences}}{\text{second}}$$

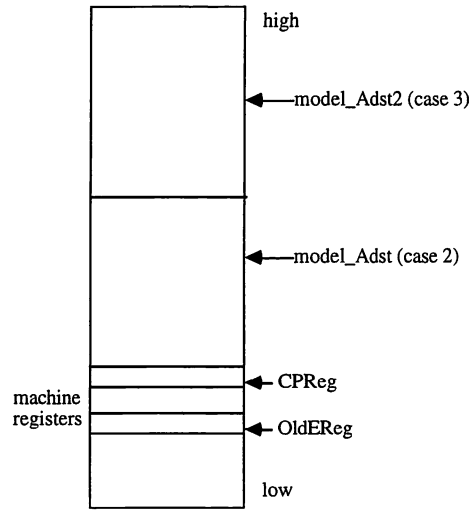
The following table gives LIPS for small values of L

L	LIPS
1	666667
2	606061
3	555556
4	512821
5	476190
6	444444
7	416666
8	392157
9	370370

We would expect L to be 4 or 5 at worst giving a speed of around 500 KLIPS.

10. Compiler Details

The compiler maintains an internal model of the "executing APM stacks and registers" roughly laid out as follows:



10.1. compile.h

This file describes to the compiler how Prolog maps onto a particular machine architecture. In principle, this file will be different for each particular machine architecture. However, the uniformity of the APM model is combined with #ifdefines to produce a single file for all the ALS Prolog systems.

```
#define TODOSIZE  16384
#define MODELSIZE 8192

/*
 * Register Base Numbers
 *  SPREG   -- stack pointer
 *  EREG    -- environment pointer
 *  HREG    -- heap pointer
 */

#ifdef m88k

#define REGS0
#define SPREG31%% index off this register for destination args
#define EREG30 %% index off this register for source args
#define HREG17

#endif
```

```
#ifdef m68k

#define REGS0
#define SPREG7
#define EREG6
#define HREG5

#endif

#ifdef i386
       %% these are mapped to real 386 registers using BASEREGS (at the %%
end of this file)
#define REGS0
#define HREG0
#define SPREG1
#define EREG2

#endif

/* CPREG and OLDEREG.
 *
 * CPREG is the machine register used for the continuation pointer.
 * OLDEREG is the machine register used for the old environment pointer.
 *
 * If no registers are available for either of these, there should be no
 * #define for them.  Otherwise, compiler will produce incorrect code.
 */

#ifdef m88k

#define CPREG15/* Continuation pointer */
#define OLDEREG14/* Old Env pointer*/

#endif

/*
 * CPREG and OLDEREG are not defined for the m68k and i386 versions.
 */


/*
 * Where things live in the low end of the model
 *
 * ASTART and AEND mark where the argument registers that actually live in
 * machine registers will be placed. ASTART marks the first register and AEND
 * marks the last one.  It is assumed that the registers are in consecutive
 * order.
 *
 * When changing these, to eliminate arguments being put into registers,
 * I recommend setting AEND to -1 and ASTART to 0.  This will make the NAREGS
 * macro work out right below.
 *
 * TSTART and TEND mark which machine registers can be used for temporary
 * variable storage. TSTART marks the first register and TEND marks the
 * last one.  It is assumed that the registers are in consecutive order.
 *
 * When changing these to eliminate temporary machine registers, I recommend
 * setting TEND to -1 and TSTART to 0.  This makes the NTREGS macro work out
```



```
 * right below.
 *
 *
 * LASTREG is the last machine register on the processor (which the compiler
 * models).
 */


#ifdef m88k

#define ASTART2/* first arg reg*/
#define AEND4/* last arg reg*/
#define TSTART5/* first temp reg*/
#define TEND9/* last temp reg*/

#define LASTREG31/* The last register in the model */

#endif

#ifdef m68k


/*
 * Low end of the model for the 020:
 *
 *  No A registers.
 *  Four temporaries in D1-D4.
 */

#define ASTART0
#define AEND-1
#define TSTART1
#define TEND4
#define LASTREG7

#endif

#ifdef i386

#define ASTART0
#define AEND-1
#define TSTART0
#define TEND-1
#define LASTREG7

#endif



/*
 * STACKADJUST is the value used to adjust the stack for the initial ALLOCATE
 * instruction in a multi-goal clause, or any stack adjustments before a CALL
 * instruction.
 */

#ifdef m88k

#define STACKADJUST 2
```

```
#endif

#ifdef m68k


/*
 * STACKADJUST is 0 for the 020 due to the fact that link and call
 * push their values onto the stack.
 */

#define STACKADJUST 0

#endif

#ifdef i386
/*
 * STACKADJUST is 0 for the 386 due to the fact that link and call
 * push their values onto the stack.
 */

#define STACKADJUST 0

#endif


/*
 * Calculate the number of argument registers (the K of the model) we have
 */

#define NAREGS ((AEND-ASTART)+1)

/*
 * The number of temporary registers which we have
 */

#define NTREGS ((TEND-TSTART)+1)



#ifdef i386

/* Provide the mapping from compiler register definitions to Prolog machine
 * registers.  All of the above definitions are offsets into this array.
 * In the other implementations, the offsets are the actual register numbers.
 */

#define BASEREGS{H_REG, SP_REG, E_REG }

#endif
```

compile.c

```
/*
 * compile_clause takes a rule r generated by parser and generates
 * code for the clause.
 *
```



```
 * The compiled code can be found in the icode buffer (see instrs.c).
 *
 */

compile_clause(r,fromparser)
   pword r;
   int fromparser;/* flag indicating if we came from the parser */
{
   pword rh;    /* rule head */

   rh = RULE_HEAD(r);
   if (TYPEOF(rh) == TP_TERM)
      rh = TERM_FUNCTOR(rh);    /* Get the rule head */

   if (TYPEOF(rh) != TP_SYM) {/* See if the rule head of appropriate type */
      if (fromparser)
     parser_error("Head of clause of inappropriate type.");
      return 0;/* return failure code if not*/
   }

   icode(IC_INIT,0,0,0,0);

   npv = classify_vars(r);      /* Do analysis of the variables and cuts */

   ngoals = RULE_NGOALS(r);     /* Tell everyone how many goals we have  */
   nvars  = RULE_NVARS(r);      /* Similarly for the number of variables */

   compute_call_env_sizes(ngoals,nvars);
                                /* Figure out how big of an environment we
                                   need after each call...results are left
                  in the call_env_sizes array         */

   if (!comp_rule(r)) {/* Generate the code for the clause     */
      if (fromparser)
     parser_error("Goal in clause of inapproprate type.");
      return 0;
   }

   /*
    * and now do something with it
    */

   icode(IC_ENDCLAUSE, FUNCTOR_TOKID(rh), FUNCTOR_ARITY(rh));
   return 1;
}



comp_rule(rule)
   pword rule;
{
   int gtp;             /* goal type */
   int macrofix;/* math fixup flag */
   pword goal;
   int functor_id;/* functor id of the goal */

   if (TYPEOF(rule) != TP_RULE) {
      fprintf(stderr, "Internal error: comp_rule called with inappropriate ar-
gument\r\n");
```

```
      als_exit(1);
   }



   ngoals = (int) RULE_NGOALS(rule);

   /*
    * Find the first goal that is not a cut.
    */

   for (firstgoalnumber=1;
        firstgoalnumber <= ngoals &&
           TYPEOF(RULE_GOALN(rule,firstgoalnumber)) == TP_SYM &&
           FUNCTOR_TOKID(RULE_GOALN(rule,firstgoalnumber)) == TK_CUT;
        firstgoalnumber++);

   /*
    * Initialize the model and emit instructions necessary for setting up
    * the environment and stack pointer for the first goal.
    */

   goalnumber = 0;

   init_model(RULE_HEAD(rule),(firstgoalnumber > ngoals)
              ? NIL_VAL
              : RULE_GOALN(rule,firstgoalnumber));

   /*
    * Compile the head of the rule.
    */

   comp_head(RULE_HEAD(rule));

   if (firstgoalnumber > 1) {
      /*
       * There is at least one cut prior to the first real goal. We want to
       * emit the instruction which performs the cut and then if the first goal
       * number is equal to the number of goals, initialize the target fields in
       * the vtbl array.  This was not done in init_model in order to
       * prevent the source argument positions from being clobbered.  It is
       * ok to clobber them after the cut since things are determinate.
       */

     if (ngoals < firstgoalnumber) {
    icode(I_CUT_PROCEED,EREG,0 ,0,0);
    return 1;
     }
     else {
    icode(I_DOCUT,EREG,0, 0,0);
    if (firstgoalnumber == ngoals)
       init_only_targets(RULE_GOALN(rule,firstgoalnumber));
     }
   }



   /*
    * firstgoalnumber will always be positive (since we start it out at one
```



```
    * in the above loop which sets it).  When ngoals < firstgoalnumber, we
    * have a unit clause and all we need to do is emit some proceed code.
    */

   if (ngoals < firstgoalnumber) {
      icode(I_PROCEED,index_of(vtbl[RETIDX].home),
            disp_of(vtbl[RETIDX].home),0,0);
      return 1;
   }

   /*
    * The garbage collector demands that each environment variable
    * has a safe value prior to execution of the first goal. So if we
    * are just prior to the first goal, we initialize all of the
    * uninitialized environment variables.
    */

   initialize_environment_variables();

   /*
    * Initialize goalnumber, goal, gtp, and macrofix.
    * The macrofix variable is a boolean which is set if do_macro was
    * able to successfully macro expand goal into some in-line code.
    * is/2, =:=/2, and other arithmetic predicates are macro expanded for
    * integer arguments (at present).
    */

   goalnumber = firstgoalnumber;
   goal  = RULE_GOALN(rule,goalnumber);
   gtp = TYPEOF(goal);
   macrofix = do_macro(goal,gtp);

   for (;;) {


     /*
      * Emit the argument set up code for the goal.
      */

     comp_goal(goal);

     /*
      * Get the functor id of the goal.  For example, the functor id in
      * p(_,a,b) would be the symbol table index representing p.
      *
      * If a valid functor id can not be obtained, then the goal is something
      * like a number which is invalid, so we return with an unsuccessful
      * completion code from this function.
      */

     if (!(functor_id = functor_id_of_term(goal)))
    return 0;

     /*
      * Check to see if the goal is a cutmacro.  If it is, then put
      * something resembling the current environment pointer into the n+1st
      * argument.  The cut point is actually the address of the last source
      * argument.  The reason for using the last one is that this cut point
      * may be passed determinately to other goals with successively smaller
```

```
            isdeterminateforsure = 1;
         model_Eend = model_E - call_env_sizes[goalnumber-1];
      }
      else
         isdeterminateforsure = 0;

      /*
       * Generate and save inline code in a secret place if the next goal
       * can be macro expanded.  The reason we generate it here is so that
       * we can skip over the trim or deallocate code if at all possible.
       */

      macrofix = do_macro(goal,gtp);

      /*
       * Put out the trim or deallocate code.  Also set up model_SP,
       * model_Adst, model_Adst2, and the targets for the next goal.
       */

      if (goalnumber < ngoals) {

         /*
          * The second argument in the following call gives the size of
          * the current environment.
          *
          * The third argument gives the value to add on to the second
          * argument for setting the stack pointer.  The reason for the
          * maximum function is to prevent us from setting the stack pointer
          * in the middle of the environment variables to be trimmed away.
          * (They still may be used to set up arguments in the next goal.)
          */

         icode(I_TRIM,
           call_env_sizes[goalnumber],
                max(call_env_sizes[goalnumber-1] - call_env_sizes[goalnumber],
                gsize)+STACKADJUST,
                isdeterminateforsure,0);
         model_Adst = model_E - call_env_sizes[goalnumber] - gsize;

         /*
          * Make stack pointer agree with the trim call above.
          */

         if (model_Adst < model_Eend)
            model_SP = model_Adst-STACKADJUST;
         else
            model_SP = model_Eend-STACKADJUST;

         /*
          * Make model_Adst2 zero since we will not be overwriting the
          * original arguments.
          */

         model_Adst2 = 0;

         init_targets(goal);
      }
      else
         deallocate_environment(goal,gtp,gsize,isdeterminateforsure);
```



```
         /*
          * model_SPstart is the place where the stack pointer starts out
          * at prior to the compilation of the goal.  Its only purpose is
          * so that we know where to start reallocating temporaries off of
          * the stack.
          */

         model_SPstart = model_SP;
          }
       }
   }


   /*
    * init_model is responsible for initializing the model of the machine
    *  prior to the start of the compilation.
    */

   init_model(head,firstgoal)
      pword head;
      pword firstgoal;
   {
      register int i;
      int firstgoalsize = goalsize(firstgoal);
      pword s;

      model_E = MODELSIZE-2;/* room for return address and prev env */
      nargs = 0;
      nargsmatched = 0;/* haven't matched any arguments yet. */

      /*
       * Install the head arguments into the appropriate part of the model.
       */

      if (TYPEOF(head) == TP_TERM) {
         register int ti,v;

         nargs = TERM_ARITY(head);
         model_E -= nargs;

         /*
          * Take care of the register part of the model first.
          */

         for (i=1; i<=NAREGS && i<=nargs; i++) {
            s = TERM_ARGN(head,i);
        ti = ASTART+i-1;
        model[model_E+i+1] = NIL_VAL;
            if (IS_VO(s) && vtbl[v=VO_VAL(s)].pvnum == 0 &&
                            vtbl[v].home == 0)  {
               vtbl[VO_VAL(s)].home = ti;
           model[ti] = s;
           model[model_E+i+1] = s;
           if (vtbl[v].lastocc > firstgoalnumber)
              vtbl[v].target = model_E+i+1;/* this may be reassigned */
           incr_usecnt(VO_VAL(s));
            }
         else {
```

```
            model[ti] = s;
         model[model_E+i+1] = NIL_VAL;
      }
       }


       /*
        * Now take care of the overflow case.
        */

       for (i=NAREGS+1; i<=nargs; i++) {
          s = TERM_ARGN(head,i);
      ti = model_E+i+1;

          if (IS_VO(s) && vtbl[VO_VAL(s)].pvnum == 0 &&
                          vtbl[VO_VAL(s)].home == 0)  {
             vtbl[VO_VAL(s)].home = ti;
         model[ti] = s;
         incr_usecnt(VO_VAL(s));
          }
      else
            model[ti] = s;
       }
   }

   /*
    * Initialize those registers not covered by the arguments (if any).
    */

   for (i=nargs+1; i<=NAREGS; i++) {
      model[ASTART+i-1] = NIL_VAL;
   }


   model_Eend = model_E;/* don't want to clobber return address */
   model_SP = model_Eend;

   for (i=0; i < nvars ; i++) {
      if (vtbl[i].pvnum) {
     vtbl[i].home = model_E-vtbl[i].pvnum;
     model[vtbl[i].home] = MK_VO(i);
      }
   }

   /*
    * Set the previous environment value up as a variable so that it can
    * be moved.
    */

#ifdef OLDEREG
   vtbl[ENVIDX].home= OLDEREG;
   model[model_E]= NIL_VAL;
   model[OLDEREG]= MK_VO(ENVIDX);
#else
   vtbl[ENVIDX].home= model_E;
   model[model_E] = MK_VO(ENVIDX);
#endif
   vtbl[ENVIDX].firstocc= 0;
   vtbl[ENVIDX].istoplevinhead= 0;
```



```
   vtbl[ENVIDX].lastocc= ngoals;
   vtbl[ENVIDX].noccurences= 2;
   vtbl[ENVIDX].pvnum= 0;
   vtbl[ENVIDX].usecnt= 1;
   vtbl[ENVIDX].target= 0;
   vtbl[ENVIDX].unsafe= 0;

   /*
    * Set the return pointer up as a variable so that it can be moved
    */

#ifdef CPREG
   vtbl[RETIDX].home= CPREG;
   model[model_E+1]= NIL_VAL;
   model[CPREG]= MK_VO(RETIDX);
#else
   vtbl[RETIDX].home= model_E+1;
   model[model_E+1]= MK_VO(RETIDX);
#endif
   vtbl[RETIDX].firstocc= 0;
   vtbl[RETIDX].istoplevinhead= 0;
   vtbl[RETIDX].lastocc= ngoals;
   vtbl[RETIDX].noccurences= 2;
   vtbl[RETIDX].pvnum= 0;
   vtbl[RETIDX].usecnt= 1;
   vtbl[RETIDX].target= 0;/* no target yet */
   vtbl[RETIDX].unsafe= 0;/* bad idea to emit a put_unsafe */

   icode(IC_BEGINALLOC,0,0,0,0);/* indicate to code generator that
                   * we are allocating environment/
                   * stack args for first goal
                   */
   if (ngoals - firstgoalnumber > 0) {
      icode(I_ALLOCATE,
            call_env_sizes[firstgoalnumber]+firstgoalsize+STACKADJUST,
            0,0,0);
      model_Eend = model_E - call_env_sizes[firstgoalnumber];
      model_SP = model_Eend - firstgoalsize - STACKADJUST;
      model_Adst = model_Eend - firstgoalsize;
      model_Adst2 = 0;
      init_targets(firstgoal);
   }
   else if (ngoals - firstgoalnumber == 0) {
      model_Eend = model_E;
      model_Adst = model_E - firstgoalsize;
      model_Adst2 = MODELSIZE - firstgoalsize;
      model[model_Adst-1] = NIL_VAL;/* this is where we'll put the
                   * return address
                   */

      if (firstgoalnumber == 1) {
     init_targets(firstgoal);
     icode(I_ALLOCATE1,nargs+2,0,0,0);
     model_SP -= (nargs+2);

     /*
      * The stuff in the following if statement will move variables
      * living in the same corresponding locations for the non-determinate
      * case.  The reason for the comparison of the firstgoalsize with
```

```
      * the arity is to avoid running this code when the first goal is
      * a cut macro (which has a goal size one greater than the arity)
      */

     if (TYPEOF(head) == TYPEOF(firstgoal)) {
        if (TYPEOF(head) == TP_TERM &&
        firstgoalsize == TERM_ARITY(firstgoal)) {
           int i,j;
           pword ah,ag;
           for (i=TERM_ARITY(head),j=TERM_ARITY(firstgoal) ;
            i > NAREGS && j > NAREGS;
            i--,j--) {
          ah = TERM_ARGN(head,i);
          ag = TERM_ARGN(firstgoal,j);
          if (TYPEOF(ah) == TP_VO && TYPEOF(ag) == TP_VO &&
              VO_VAL(ah) == VO_VAL(ag)) {
             move(model_E+i+1,model_SP+i+1);
             model[model_SP+i+1] = model[model_E+i+1];
          }
           }
           if (i==j) {
#ifndef CPREG
           move(model_E+1,model_SP+1);/* move return address */
           model[model_SP+1] = model[model_E+1];
           vtbl[RETIDX].home = model_SP+1;
#endif

#ifndef OLDEREG
           move(model_E,model_SP);/* move env pointer */
           model[model_SP] = model[model_E];
           vtbl[ENVIDX].home = model_SP;
#endif
           }
        }
        else if (firstgoalsize == 0) {
#ifndef CPREG
           move(model_E+1,model_SP+1);/* move return address */
           model[model_SP+1] = model[model_E+1];
           vtbl[RETIDX].home = model_SP+1;
#endif

#ifndef OLDEREG
           move(model_E,model_SP);/* move env pointer */
           model[model_SP] = model[model_E];
           vtbl[ENVIDX].home = model_SP;
#endif
        }
     }

     icode(I_ENDALLOC1,0,0,0,0);
     icode(I_ADDTOSP,-max(0,firstgoalsize-nargs),0,0,0);
     vtbl[RETIDX].target = model_Adst-1;
     vtbl[ENVIDX].target = model_Adst-2;
     model_SP -= max(0,firstgoalsize-nargs);
     for (i=model_SP; i < model_Adst-2; i++)
        model[i] = NIL_VAL;
      }
      else {
     /*
```



```
      * only goal will be determinate due to a cut.  In this case, we
      * may avoid doing the allocate1 code since it is pointless to
      * set things up for the non-determinate case.  But we don't want
      * the compiler to overwrite things in case it isn't so we don't
      * let any of the source locations be targeted.
      */

         model_Eend = model_E;
         model_Adst = MODELSIZE - firstgoalsize;
         model_Adst2 = model_Adst;      /* this will get us the variable
                  * movement optimizations
                  */
     init_targets(NIL_VAL);/* still need to do some of the
                  * initializations
                  */
     icode(I_ADDTOSP,-max(0,firstgoalsize-nargs),0,0,0);
     vtbl[RETIDX].target = model_Adst-1;
     model_SP -= max(0,firstgoalsize-nargs);
     for (i=model_SP; i < model_Eend-1; i++)
        model[i] = NIL_VAL;
      }


   }
   else {   /* no goals */
      model_Adst = model_Eend;
      model_Adst2 = 0;
      init_targets(firstgoal);
   }
   icode(IC_ENDALLOC,0,0,0,0);
   model_SPstart = model_SP;
}


/*
 * init_targets scans the given goal and looks for top-level temporary
 *  variables.  It initializes the target field in the vtbl array
 *  for the variable to be the location that this variable should be
 *  moved to.  When we attempt to move this variable later on, or we
 *  want to find a home for it, a good attempt will be made to make
 *  the variable live in the target location.
 *
 * init_targets is usually called just prior to compilation of a goal.  Thus
 *  it performs other initializations which are important.  The portion
 *  of the model corresponding to the machine registers is NIL'd out thus
 *  freeing them up for allocation.  This is not dangerous due to our
 *  assumption that these machine registers do not survive between goals.
 *  (They do however survive between the head and the first goal).  Those
 *  portions of the target locations for the goal which do not overlap
 *  part of the existing environment are also NIL'd out.  These locations
 *  correspond to portions of the stack yet to be allocated and so are
 *  assumed to be free.
 */

init_targets(t)
   pword t;
{
   register int i;
```

```
   /*
    * Initialize that portion of the model corresponding to the machine
    * temporary registers
    */

   for (i=TSTART; i<=TEND; i++)
      model[i] = NIL_VAL;

   if (goalnumber > firstgoalnumber) {
      for (i=ASTART; i<=AEND; i++)
     model[i] = NIL_VAL;
   }

   /*
    * Initialize the destination argument positions that do not overlap
    * the environment.
    */

   for (i = model_Adst-2; i < model_Eend; i++)
      model[i] = NIL_VAL;

   /*
    * Initialize the target fields in the vtbl array.
    */

   init_only_targets(t);

   if (goalnumber == ngoals) {
#ifdef CPREG
      vtbl[RETIDX].target = CPREG;
#else
      vtbl[RETIDX].target = model_Adst-1;
#endif

#ifdef OLDEREG
      vtbl[ENVIDX].target = OLDEREG;
#else
      vtbl[ENVIDX].target = model_Adst-2;
#endif
   }
}


/*
 * init_only_targets is called by init_targets to initialize the target
 *  fields in the vtbl array.  No other initializations are performed.
 *  This function is also called just prior to compilation of G in
 *  something of the form P :- !, G.  In this case it is desirable to
 *  initialize the target fields, but not perform the other initializations
 *  found in init_targets.
 */

init_only_targets(t)
   pword t;
{
   pword s;
   register int i,n;
```



```
   if (goalnumber >= firstgoalnumber)
      for (n=nvars,i=0; i<n; i++)
     vtbl[i].target = 0;/* clear the targets */

   if (TYPEOF(t) == TP_TERM) {
      n = TERM_ARITY(t);
      for (i=1; i<=n; i++) {
         s = TERM_ARGN(t,i);
     if (IS_VO(s)) {
        vtbl[VO_VAL(s)].target = ((i <= NAREGS) ? ASTART : model_Adst)+i-1;
     }
      }
   }
}


comp_head(head)
   pword head;
{
   int i;
   pword arg;
   int tp;
   int ti;
   if (TYPEOF(head) != TP_TERM) {
      record_first_argument(TP_VO,0);
      return;
   }

   for (i=1; i <= TERM_ARITY(head); i++) {
      nargsmatched = i;
      if (i<=NAREGS) {
     ti = ASTART+i-1;
      }
      else {
     ti = model_E+i+1;
      }
      arg = TERM_ARGN(head,i);
      tp = TYPEOF(arg);
      switch (tp) {
     case TP_SYM :
        icode(I_G_SYM,FUNCTOR_TOKID(arg),index_of(ti),disp_of(ti),0,0);
        model[ti] = NIL_VAL;
        break;
     case TP_INT :
        icode(I_G_INT,INT_VAL(arg),index_of(ti),disp_of(ti),0,0);
        model[ti] = NIL_VAL;
        break;
     case TP_VO :
        {
           int v = VO_VAL(arg);
           if (TYPEOF(model[ti]) == TP_VO &&
           VO_VAL(model[ti]) == v) {
          if (vtbl[v].usecnt == 0) {
             move(ti,vtbl[v].home);
             vtbl[v].usecnt++;
          }
          else if (vtbl[v].home != ti) {
             icode(I_G_VALUE,index_of(ti),disp_of(ti),
               index_of(vtbl[v].home),disp_of(vtbl[v].home));
```

```
            incr_usecnt(v);
          }
          if (vtbl[v].usecnt == vtbl[v].noccurences)
             model[ti] = NIL_VAL;/* only nil it out when
                          we are through with it */
           }
        }
        break;
     case TP_UIA :
     case TP_TERM :
     case TP_LIST :
        comp_head_structure(ti);
        break;
     default :
        printf("Something funny in comp_head!\n");
        break;
      }
      if (i == 1) record_first_argument(tp,arg);
   }
}

/*
 * record_first_argument is used to aid first argument indexing.
 */
move(n,m)
   int n,m;
{
   if (n == m && TYPEOF(model[m]) == TP_VO &&
       vtbl[VO_VAL(model[m])].pvnum == 0) return;
    /* don't move it if it's already in place */
   icode(I_MOVE,index_of(n),disp_of(n),index_of(m),disp_of(m));
}

/*
 * find_home is given an index of a variable in vtbl.  It attempts to find
 *  a home for the variable if it doesn't have one already.
 */

find_home(v)
{
   int home;
   if (vtbl[v].home) return vtbl[v].home;
   else if (vtbl[v].target && free_target(vtbl[v].target))
      home = vtbl[v].target;
   else
      home = find_temp();
   model[home] = MK_VO(v);
   vtbl[v].home = home;
   return home;
}

/*
 * find_temp looks first for a machine temporary to use and then on the
 *  stack between SP and Adst for a location to use.  If it can't find
 *  one this way, it decrements SP and uses the top of stack.  find_temp
 *  does not initialize the location in the model.  The caller is
 *  expected to perform this function in order to complete the allocation.
 */
```



```
int find_temp()
{
   int i;

   for (i=TSTART; i<=TEND; i++)
      if (model[i] == NIL_VAL)
     return(i);

   for (i=model_SP; i<model_SPstart; i++)
      if (model[i] == NIL_VAL)
     return(i);

   /*
    * No temporaries available.  Allocate a new stack location.
    */

   icode(I_ADDTOSP,-1,0,0,0);
   model_SP = model_SP-1;
   model[model_SP] = NIL_VAL;/* initialize new location */

   return(model_SP);
}

/*
 * find_temp_in_reg attempts to find a machine temporary to use.  It will
 *  return zero if it can not find one.
 */

int find_temp_in_reg()
{
#if NTREGS > 0
   int i;

   for (i=TSTART; i<=TEND; i++)
      if (model[i] == NIL_VAL)
     return i;
#endif

   return 0;
}


/*
 * free_target(loc)
 *
 *  This function is given a location in the model (loc) to free.
 *  It will do its level best to free up the location if it is not already
 *  free.  If the location is already free or can be freed, it returns 1.
 *  Otherwise it returns 0.
 */


free_target(loc)
   int loc;
{
   int start, tdbase, loc2;
   int limit, v, dst;

   TDT_SET(tdbase,0);
```

```
start = loc;

for (;;) {

   /*
    * Set loc2 and limit depending on the situation
    */

  if (ASTART <= loc && loc <= AEND) {
 limit = ASTART + nargsmatched;
 loc2 = loc;
  }
  else if (loc <= LASTREG) {
 limit = LASTREG;
 loc2 = loc;
  }
  else if (loc >= model_E) {
 limit = model_E + 2 + nargsmatched;
 loc2 = loc;
  }
  else if (model_Adst2 && loc >= model_Adst-2) {
 limit = model_E + 2 + nargsmatched;
 loc2 = (loc - model_Adst) + model_Adst2;
  }
  else {
 limit = MODELSIZE;
 loc2 = loc;
  }

  if (loc != loc2) {
 if (IS_VO(model[loc])) {
    v = VO_VAL(model[loc]);
    if (vtbl[v].pvnum)
       move_perms(loc);
    else if (vtbl[v].home == loc)
       move_to_temp(loc);
 }
  }

  if (model[loc2] == NIL_VAL)
 break;
 else if (IS_VO(model[loc2]) && vtbl[v=VO_VAL(model[loc2])].home == loc2) {
 if (vtbl[v].pvnum) {
    move_perms(loc2);
    break;
 }
 else if (vtbl[v].home == vtbl[v].target)
    break;
 else if (vtbl[v].target && vtbl[v].target != start) {
    to_do[to_do_top++] = loc;
    to_do[to_do_top++] = loc2;
    loc = vtbl[v].target;
 }
     else {
    move_to_temp(loc2);
    break;
 }
  }
  else if (loc2 < limit)
```



```
     break;
      else if (loc == start)
     return 0;  /* There is nothing on the stack */
      else {
     loc2 = to_do[--to_do_top];
     loc  = to_do[--to_do_top];
     move_to_temp(loc2);
     break;
      }
   }

   while (to_do_top > tdbase) {
      dst = loc;
      loc2 = to_do[--to_do_top];
      loc  = to_do[--to_do_top];
      v = VO_VAL(model[loc]);
      if (loc == loc2 && IS_VO(model[dst]) && v == VO_VAL(model[dst]))
     ;      /* variable is already in place */
      else {
     move(loc2, dst);
     model[dst] = model[loc2];
     model[loc2] = NIL_VAL;
      }
      vtbl[VO_VAL(model[dst])].home = dst;
   }
   return 1;
}


move_perms(start)
   int start;
{
   int i;
   for (i=start; i>=model_Eend; i--)
      if (IS_VO(model[i])) {
     int v = VO_VAL(model[i]);
     if (vtbl[v].pvnum && vtbl[v].home == i) {
        int t;
        if ((t = vtbl[v].target)&&/* if there's a target */
        model[t] == NIL_VAL&&
                 ( model_Adst2 == 0 ||
          model[model_Adst2+t-model_Adst] == NIL_VAL)) {
          ; /* do nothing....t is already set by the if */
        }
        else
           t = find_temp();
        if (vtbl[v].unsafe && vtbl[v].firstocc != 0) {
           icode(I_P_UNSAFE,index_of(i),disp_of(i),
               index_of(t),disp_of(t));
           vtbl[v].unsafe = 0;
        }
        else
           move(i,t);
        model[t] = model[i];
        vtbl[v].home = t;
        vtbl[v].pvnum = 0;/* no longer permanent */
        if (i != t)
           model[i] = NIL_VAL;
     }
```

```
      }
}


move_to_temp(loc)
   int loc;
{
   int v = VO_VAL(model[loc]);
   int t = find_temp();
   model[t] = model[loc];
   vtbl[v].home = t;
   move(loc,t);
   model[loc] = NIL_VAL;
}


/*
 * comp_goal is responsible for setting up arguments for a goal.  The
 *  strategy used here is to set up the non-variable parts of the
 *  first in the hope that by the time we get done setting up the
 *  structure, the variables will be set up in the appropriate locations.
 */

comp_goal(goal)
   pword goal;
{
   int i,n,v,loc;
   pword arg;
   if (TYPEOF(goal) != TP_TERM)
      return;

   n = TERM_ARITY(goal);

   /*
    * compile non-variable parts
    */

   for (i=1; i<=n; i++) {
      if (i<=NAREGS) {
     loc = ASTART+i-1;
      }
      else {
     loc = model_Adst+i-1;
      }
      arg = TERM_ARGN(goal,i);
      switch (TYPEOF(arg)) {
     case TP_INT :
            free_target(loc);
        icode(I_P_INT,INT_VAL(arg),index_of(loc),disp_of(loc),0);
        model[loc] = arg;
        break;
     case TP_SYM :
        free_target(loc);
        icode(I_P_SYM,FUNCTOR_TOKID(arg),index_of(loc),disp_of(loc),0);
        model[loc] = arg;
        break;
     case TP_UIA  :
     case TP_LIST :
     case TP_TERM :
```



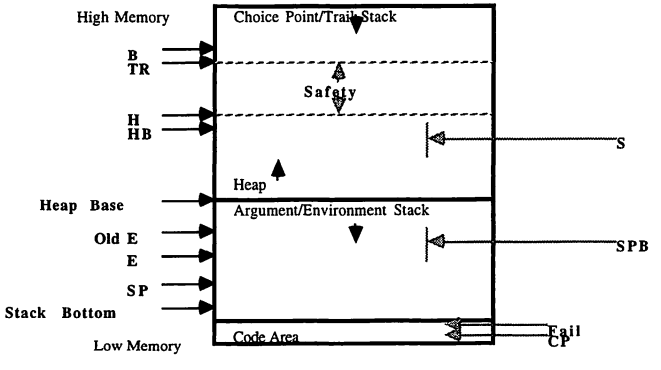
```
        comp_goal_structure(arg,loc);
        break;
     default :
        break;
      }

}


/*
 * set up variables
 */

for (i=1; i<=n; i++) {
   arg = TERM_ARGN(goal,i);
   if (i<=NAREGS) {
  loc = ASTART+i-1;
   }
   else {
  loc = model_Adst+i-1;
   }
   if (IS_VO(arg)) {
  v = VO_VAL(arg);
  if (!(IS_VO(model[loc]) && VO_VAL(model[loc]) == v) || vtbl[v].pvnum) {

     /* There is something to do if the variable is not already
      * at home or if it is at home, but is still a permanent variable.
      */
     free_target(loc);

     if (vtbl[v].usecnt == 0) {
        if (vtbl[v].pvnum) {
       icode(I_P_YVAR, index_of(vtbl[v].home), disp_of(vtbl[v].home),
              index_of(loc), disp_of(loc));
        }
        else {
       icode(I_P_XVAR, index_of(loc), disp_of(loc),0,0);
       vtbl[v].home = loc;
       vtbl[v].unsafe = 0;
        }
        vtbl[v].usecnt++;
     }
     else {
        int src = vtbl[v].home;
        if (vtbl[v].pvnum && vtbl[v].lastocc == goalnumber &&
                vtbl[v].firstocc != 0 && vtbl[v].unsafe) {
       icode(I_P_UNSAFE, index_of(src),
                disp_of(src),
                index_of(loc),
                disp_of(loc));
       vtbl[v].pvnum = 0;
       vtbl[v].home = loc;
       model[loc] = MK_VO(v);
       vtbl[v].usecnt++;
       if (loc != src) model[src] = NIL_VAL;
        }
        else {
       move(src,loc);
       model[loc] = model[src];
```

```
          vtbl[v].usecnt++;
          if (vtbl[v].lastocc == goalnumber) {
#ifdef notdef
             if (ASTART <= loc && loc <= AEND)
#endif
            vtbl[v].home = loc;
             vtbl[v].pvnum = 0; /* convert it to a temporary */
          }
           }
        }
     }
     else
        vtbl[v].usecnt++;/* somebody else did the move for us, but
                * they didn't update the use count
                */
      }
   }
}
```


11. Interrupts

Files: int.c, int.m4

ALS Prolog compares the distance between the backtracking stack and the heap to see if a garbage collection is necessary. If this distance ever becomes less than a pre-defined value, called the heap safety value, the program is interrupted, and the garbage collector code is run. {TR -H < wm_normal ==> gc}



The test is done at each entrance to any procedure. When a procedure is called by a call or execute instruction, control is transferred to a location in the name table entry for the procedure which is being called. The first thing done in this patch of code is the overflow check (i.e., whether TR - H < wm_safety, which is normally set at 4K = wm_normal). If no overflow has been detected, control continues on. Either control is passed to the first clause, or the indexing code is run, or the resolve reference code is run.

This overflow check is useful as a general interrupt mechanism in Prolog. Since it is always performed on procedure entry, and since all calls must go through the procedure table entry, any procedure call can be interrupted. {Note that this raises problems for in-line code expansion, but there is a solution at least as far as cut-macros, etc.} Notice that increasing the value of wm_safety increases the liklihood that TR-H < wm_safety. So if the heap safety value wm_safety is set larger than the current distance between the heap and backtrack stack, the next call will be stopped.

On the Sun, this idea is currently (11/88) used for garbage collection and for driving the debugger. However, the overflow code must examine all of the possibilities of why the interrupt happened and what to do about it. As the overflow code is written in assembler, it is difficult to add in new interrupts into the system. The approach taken in the 80386 implementation is to let the overflow handler, or what we will call now the *interrupt handler*, be written in Prolog. This was originally done on the Mac for garbage collection and the resolve refrence code, but it was not quite as general a mechanism as is being used now on the 386 and in the IM.

Either the system, or the user, can trigger an interrupt by setting the heap safety (wm_safety) to a value which guarantees that the next call will be interrupted. When the interrupt occurs, the interrupted call is packaged up in a term and passed as an argument to the interrupt handler. The continuation pointer from the handler will point into the interrupted clause. Consequently, when the

interrupt handler returns (with success), the computation will continue where it would have continued if the call had never been interrupted. An example will help make this clearer. Suppose the goal

```
:- b, c, f(s,d), d, f.
```

is running, and the call for c/0 has returned and f/2 is about to be called. Something happens to trigger an interrupt: wm_safety is set > TR-H, and then f/2 is called. The overflow check code will run, and the trap will occur. Since $int(f(s,d)) will be called, the goal will now operationally look as though it were

```
:- b, c, $int(f(s,d)), d, f.
```

Rather than f/2 running as originally "planned", $int/1 will run. When $int/2 returns (with success), d/0 will run, which is what would have happened if f/2 had run and returned with success. If $int/1 decides to run f/2, all it has to do is call f(s,d). $int/1 can leave choice points points and be cut since it is exactly like any other procedure call. Any cuts inside $int/1 will have no effects outside of the call. In other words, it is a fairly safe operation. Once the interrupt handler is called, the interrupt trigger should be reset, or the interrupt handler will interrupt itself, going into an infinite loop.

One use for this mechanism is write the debugger in Prolog (cf. Section 13) and another is to write a clause decompiler in Prolog itself. The basic scheme is to continually interrupt every call, perform some action, including saving some global information, and then release the interrupt by going onto the next call which would have run. But this call gets interrupted, and the cycle is repeated, etc. For the debugger, the scheme for simple creeping is roughly this:

```
interrupt Goal;
get global information showing depth and call numbers;
show Goal to user & get user action (= creep);
save new depth and call numbers globally;
make sure interrupts are reset;
"release" interrupt by calling Goal;
```

Thus, a trace mechanism would require interrupt handler ($int) clauses roughly of the form

```
$int(Goal) :-
        get previously globally stored depth & call number info,
        show user Goal,
        store new depth & call number infor globally,
        set an interrupt,
        call Goal.
```

For decompilation, the basic idea is to use the fact that the functioning of the interrupt mechanism produces a term which is a "decompilation" of the current goal; so all we have to do is continually interrupt every goal, and collect the terms representing the interrupted goals in list which is stored in some global location; when the end of the clause to be decompiled is reached, the globally stored list is processed to produce a term correctly representing the clause.

Thus, a decompilation mechanism would require interrupt handler ($int) clauses roughly of the form

```
$int(Goal) :-
        obtain previously globally stored information;
        save Goal  with previously stored information;
```



```
        set an interrupt.
```

Unlike the tracing example, for the decompiler, t is not necessary to call Goal at the end of the $int clause. This is since we are decompiling a clause, and so the elements of other clauses applying to Goal in the body of the given clause are not of interest in this case. All we want to do is obtain the Goals making up the given body.

A ^C trapper could be written which keeps the current Goal pending. Then the user could be given a choice of turning on the trace mechanism, calling a break package which would continue the original computation when it returned, or even stop the computation altogether.

These discussions indicate that the interrupt mechanism will be used for a variety of purposes. Conceptually, it would be nice to think of different kinds of interrupts being issued. That is, in order for all the foregoing operations to take place, the interrupt handler needs to know which interrupt has been issued. This is done using the *magic value*. *Magic* is a global variable, which is passed as the first argument to the $int/2 call by the underlying mechanism. That is, whenever the TR-H < wm_safety check traps the computation, the expression in the global variable Magic (think of it as a register) is packaged up as a term which is passed as the first argument of the generated call to $int; the second argument of the call is the Goal which was interrupted:

```
$int(Magic,Goal)
```

This enables the system to distinguish among different interrupts. One the one hand, the user/systme programmer provides $int/2 clauses which are keyed to the first argument: certain clauses for handling the debugger, certain clauses for handling decompilation, etc.

On the other hand, these same clauses must be able to set and change the value of the Magic register to insure that the correct interrupt is seen as occuring in various contexts.

One observation is important. If the interrupt handlers ($int) are to be written in Prolog itself, there must be a means of turning the interrupt mechanism off. Otherwise, the calls in the interrupt handler clauses themselves will be interrupted, causing loops or at least undesired behavior. {But also note that one may not want to turn off the function of the gc part of the interrupt mechanism. Thus, one may want to view the interrupts as hierarchically stratified or prioritized: gc interrupts could run even when the next level of interrupts are turned off.}

The Magic "register" must be a global entity since one cannot predict which user program clauses may be required to have access to it (i.e., when an interrupt will occur). If Magic is allowed to contain arbitrary Prolog terms, the Magic register will at the same time provide a means of storing the global information required by the different interrupt mechansims, as described informally above. That is, Magic can be used to store the debugger's depth and call information, or the decompiler's partially constructed list of goals from the target clause body. That is, information can be passed back from an interrupt, such as the accumlated goals from a clause which is being decompiled.

In order to write code such as the decompiler in Prolog, several primitive routines are needed. The system programmer must be able to set and examine the magic value. This is done with the *setMagic/1* and *getMagic/1* calls. The programmer must also be able to interrupt the next call. This is done with the *ouch/0* call. When an interrupt occurs, further interrupts are turned off. Interrupts are turned back on with *resetint/0*. For example, the goal

```
:- ouch, a.
```

will call

```
$int(Magic,a)
```

If the clause

```
b :- ouch.
```

is called by the goal

```
:- b,a.
```

then

```
$int(Magic,a)
```

will be called once again, since after b returns, a/0 is the next goal called. Finally, there must be a way of calling a goal without interrupting it, but setting the interrupt so that the the goal after the goal called will be interrupted. This is done with *ocall/1*. If a/0 is to be called and the next call after it is to be interrupted, the call

```
:- ocall(a)
```

is used. For example, if a/0 is defined by

```
a :- b.
```

then

```
:- ocall(a).
```

will call

```
:- $int(Magic,b).
```

not

```
:- int(Magic,a).
```

However, if a/0 is merely the fact

```
a.
```

then the call

```
:- ocall(a),b.
```

will end up calling

```
:- $int(Magic,b).
```

As an extended example of the use of these routines, a clause decompiler will now be developed. This version is the so-called *procedure-level decompiler*. It is not the version actually used in the present ALS Prolog system, which is a *clause-level decompiler*. The latter is discussed in Section 12. The code sketch presented above gave the general idea:

```
$int(Goal) :-
    obtain previously globally stored information;
    save Goal  with previously stored information;
    set an interrupt.
```

As indicated, the Magic register will be used to accumulate the list of Goals occurring the body of the clause being decompiled. So the form of the $int/2 clause for the decompiler might be roughly this:

```
$int(decomp(GoalsSoFar,Final),NewGoal) :-
    setMagic(decomp([NewGoal | GoalsSoFar],Final)),
    resetint,
    ouch.
```


Notice that the list of goals from the body is being accumulated in reverse order. It would be possible to use a difference list to accumulate them in correct order. The action of this clause is:

the magic value has the new goal added to it.,

the interrupt mechanism is reset (so the next goal will be caught),

the trigger is set for the next call.

To start the decompiler, a call should be made which will match the head of the target clause to be decompiled (and no others). This will cause the clause to be run. However, if the Magic value is set for the decompiler, and if the interrupt mechanism is invoked, each subgoal of the target clause will be interrupted, added to the accumulated list, and then discarded before it can be run. Let $source(Head,Body) be the top-level of the deompiler. So it will be given Head, and will produce the Body of the first clause whose head matches the given Head. Thus, the $source/2 clause which starts the decompilation process would be of the form:

```
$source(Head,Body) :-
    setMagic(decomp([],Body)),
    ocall(Head).
```

First, the magic value is set to the decompiler interrupt with an initially empty body and a variable in which to return the completed body of the decompiled clause. Then the Head is called as a goal using ocall/1, which will make sure that the next goal called after Head will be interrupted. But the next goal called after Head will be the first goal in the body of the first clause whose head matches Head. The $int/2 clause above will then catch all top-level subgoals in this body. The head code for the clause whose head matches Head will bind any variables in Head from values in the head of the matching clause, and all variables that are in both the head and body of the clause will be correct in the decompiled clause, since an environment has been created for the clause. Since the clause is actually running, each subgoal will pick up its variables from the clause environment. If the decompiler should ever backtrack, the procedure for Head will backtrack, going on to the next clause, which will be treated in the same way.

The only tricky thing is the stopping of the decompiler. With the code we have thus far, the decompiler will be something like the Sorcerer's Apprentice, sucking up and decompiling all goals handled by the system once the decompiler is turned on. That is, suppose $source occurs in the following context:

```
....a, $source(g,B), c,d,....
```

When $source is called, the decompilation Magic value is set, and the first interrupt is issued. It is clear from the code above that all the calls occurring in the body of a matching clause for g will be accumulated, and then c,d,... will also be called, interrupted, and accumulated, etc., etc. {In fact, after the current top-level goals are accumulated, it will accumulate the show_answers call, and then any further goals submitted, etc.} So we must find a way of turning of the decompiler interrupt when we run off the end of the body for the clause matching g. The best method would be to utilize a distinguished system goal which the decompiler would recognize as signaling the end of a clause. However, is very desirable that the decompiler be able to decompile itself -- this sort of thing is the essence of meta-circular system programming. So rather than having a special goal which would always stop the decompiler, some way must be found to make only a particular call to this special goal cause the decompiler to stop. This will be accomplished if the clauses above are changed to:

```
$int(decomp(ForReal,Goals,Final),$endSource(Variable)) :-
        ForReal == Variable,!.

$int(s(ForReal,Goals,Final),Goal) :-
        setMagic(decomp(ForReal,[Goal|Goals],Final)),
        resetint,
        ouch.

$source(Head,Body) :-
        setMagic(decomp(ForReal,[],Body)),
        ocall(Head),% jump in, accumulate list, & return with goals list
        $endSource(ForReal).% marks exit from decompilation
```

Here, $source/2 carries the variable ForReal in its environment. This variable is carried through the interrupts through the magic value. If the interrupted goal is ever $endSource(ForReal), the decompiler stops. Note that $endSource(ForReal) will be caught when $source/2 is called, since all subgoals are then being caught. Thus, in this case, the `ForReal' argument will come from $source/2's environment. Otherwise, the interrupted goal is added to the growing list of subgoals, and the compuation continues. If $source/2 is called by $source/2, a new (and different) environment will be in place, and the first $endSource/1 encountered will be caught and stored, but not the second one.

A complete decompiler which handles choice points properly is presented below:

```
export $source/3.

$source(Module,Head,Clause)
    :-
        % Source of the ForReal variable used to mark the end
        % of extraction.
    $source(ForReal,Module,Head,Clause).

$source(ForReal,Module,Head,Clause)
    :-
        % Set the s/2 interrupt.
    setMagic(s(ForReal,Back)),
        % Start extracting.
    ocall(Module,Head),
        % This goal is never run, but is used to stop the extracter.
    $endSource(ForReal),
        % Make the clause to be returned.
    fixBody(Back,Head,Clause),
        % And set a normal interrupt.
    setMagic(gc).
                % $source has been failed all the way. Have to cleanup % s/2
interrupt pending % from the last clause of the % extracted procedure.

$source(ForReal,_,_,_)
    :-
            % Stop the decompiler. This goal is never run.
    $endSource(ForReal),
            % Got to clean up choice point in s/2 interrupt.
    setMagic(fail),
    fail.
            % The s/2 interrupt is used as the entry point in the %
interrupt code for % the source code extracter. It used % used for entering a
```



```
clause for the !first time.!

    % Going into a clause for the first time. $int(s(ForReal,Final),_,Goal)
    :-
    $source0(ForReal,[],Final,Goal).
            % Catch a failure that means $source is failing all
            % the way out.

$int(s(_,_),_,_)
    :-
            % See if ultimate failure has been requested.
    getMagic(fail),
            % Yep. Do it.
    !,fail.
            % $source has backtracked and we need to see if there is % another matching % clause in the procedure.

$int(s(ForReal,Final),_,_)
    :- !,
            % Set interrupt for entry into a clause.
    setMagic(s(ForReal,Final)),
            % And fail into it.
    ocall(fail).

            % For s/3 interrupt, we are in the middle of source % extracting a clause.

$int(s(ForReal,Goals,Final),_,Goal)
    :- !,
    $source0(ForReal,Goals,Final,Goal).

/*-----------------------------------------------------------------
   When the source extracter was called, a variable was given which is used to
identify the end of extraction. This variable is ForReal, and the $endSource
term will have this variable as its argument when the decmpiler is brought to
an end. Here we check it. This is done so that we can extract the extracter.
  -----------------------------------------------------------------*/

$source0(ForReal,Final,Final,$endSource(Check))
    :-
    Check == ForReal,!. % Not at the end.

$source0(ForReal,Goals,Final,Goal)
    :-
            % Set s/3 interrupt with the latest goal tacked on.
    setMagic(s(ForReal,[Goal|Goals],Final)),
            % Since the continuation pointer points into the clause
            % that is being extracted, all we have to do is return % from this
one without starting up the sub-goal we have
            % just extracted, after setting up the next interrupt.
    resetint,
    ouch.
```

12. 12Decompilation

Files: builtins.pro builtins.c

13. 13Debugger

Files: debugger.pro (differ for different versions, should come together)

386 Version (uses interrupt handler in Prolog)

INITIAL CORE VERSION -- NO SPY POINTS, ETC.

```
%
% yad: Yet another debugger
%
% Written by Keith Hughes
%
% Uses the new interrupt facilities available for the 386 Prolog
%

module builtins.

export trace/0.

trace :-
     setMagic(debug0),
     ouch.

notrace.
%
% We first trap on debug0 so that we can initialize the debugger properly.
%

% Next call was a notrace. Stop.
$int(debug0,_,notrace) :- !.
% Must have a goal to run.
$int(debug0,Module,Goal) :-
               % We want call number and depth at 1.
     setMagic(debug1(1,1)),
               % Start with the first goal.
     dogoal(1,1,Module,Goal).
% Need a choice point so if goal fails completely.
$int(debug0,Module,Goal) :-
               % Stop the debugger.
     notrace,
               % Will need interrupts for other things.
     resetint,
     fail.

% Time to stop the debugger.
$int(debug1(_,_),_,notrace) :- !.
% Ahh. A goal to trace.
$int(debug1(Box,Depth),Module,Goal) :-
               % Calculate the new depths and box numbers and
               % reset the values in Magic.
     NewDepth is Depth+1,
     NewBox is Box+1,
     setMagic(debug1(NewBox,NewDepth)),
               % And trace the current depth and box.
     dogoal(NewBox,NewDepth,Module,Goal).

dogoal(Box,Depth,Module,Goal) :-
```

```
    write('Current real goal is'),write(Goal),nl,
            % Get a goal form for printing.
    goalFix(Goal,XGoal),
            % Give the call port.
    showGoal(Box,Depth,call,Module,XGoal,Execute),
            % And do the call.
    execute(Execute,Module,Goal),
            % Don't want to trace coming out. Might even sneak
            % in through a choice point.
    notrace,
            % Give the exit and redo calls.
    exitOrRedo(Box,Depth,Module,XGoal,Cont),
            % And continue through the continuation pointer.
    continue(Cont).
dogoal(Box,Depth,Module,Goal) :-
            % Don't want to debug this clause.
    notrace,
    goalFix(Goal,XGoal),
    showGoal(Box,Depth,fail,Module,XGoal,Fail),
            % Fail as the user wishes.
    fail(Fail).

%
% Depending on what the user wants, we either go ahead and trace the goal
% by making sure the interrupt will fire the next time, or we just call
% the goal and watch what happens.
%

% Traceable goal?
execute(1,Module,Goal) :- !,
            % Got to trace this one.
    resetint,
    ocall(Module,Goal).
% Execute the goal with no trace.
execute(0,Module,Goal) :-
    Module:Goal.

%
% continue/1 decides how the compuation will continue. If the debugger is to
% continue running, then we must restart the debugger for the continuation
% pointer. If not, we just want to continue without resetting things up.
%

% Do we still want the debugger?
continue(1) :- !,
            % Re-establish the debugger
    resetint,
    ouch.
% Just let the computation proceed.
continue(0).

%
% fail/1 decides how the compuation will continue. If the debugger is to
% continue running after the failure, then we must restart the debugger for
% the failure. If not, we just want to fail without resetting things up.
%

% Do we still want the debugger?
fail(1) :- !,
```



```
            % Re-establish the debugger
    resetint,
    ocall(builtins,fail).
% Just let the computation fail.
fail(0) :-
    fail.

%
% This procedure handles printing out the exit and redo ports.
% When it is first called, the exit port is printed. If failure occurs
% then we go to the second clause, which will print a redo port.
%

exitOrRedo(Box,Depth,Module,Goal,Cont) :-
    showGoal(Box,Depth,exit,Module,Goal,Cont),
            % Depth must go down.
    NewDepth is Depth-1,
            % Not guaranteed that Box in head is right.
    getMagic(debug1(OldBox,_)),
    setMagic(debug1(OldBox,NewDepth)).
exitOrRedo(Box,Depth,Module,Goal,_) :-
            % Don't want to trace this call.
    notrace,
    showGoal(Box,Depth,redo,Module,Goal,Fail),
            % Fail as the user requests.
    fail(Fail).

showGoal(Box,Depth,Port,Module,Goal,Response) :-
            % Save where user was writing and put at what
            % we want.
    seeing(See),
    telling(Tell),
    see(user),
    tell(user),
            % Show 'em what the port looks like.
    writeGoal(Box,Depth,Port,Module,Goal),
            % And find out what do do.
    getResponse(Port,Response),
            % Restore the I/O channels that the user had.
    seeing(See),
    telling(Tell).

%
% Prompt the user for what to do at this port.
%

getResponse(Port,Response) :-
            % Make sure that we bother with this one.
    leashed(Port),!,
            % Give prompt and get input.
    write(' ? '),
    getLine(Line),
    doResponse(Line,Response,Port).
% Not a leashed port. Keep debugging.
getResponse(_,1) :-
            % Move to the next line, though, so it looks nicer.
    nl.

%
```