TR-0747

A Portable and Reasonably Efficient Implementation of KL1

by
T. Chikayama

March, 1992

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
(03)3456-3191～5
Telex ICOT J32964
Minato-ku Tokyo 108 Japan

Institute for New Generation Computer Technology

A Portable and Reasonably Efficient Implementation of KL1

Takashi Chikayama
Institute for New Generation Computer Technology

Abstract

An implementation scheme of a concurrent logic programming language KL1 by compiling into C language code is investigated. The implementation can be portable with this scheme. An experimental implementation with the scheme also showed reasonable efficiency.

This paper describes merits of the scheme, difficulties in efficiency and solutions to the efficiency problems. An outline of the experimental implementation and preliminary evaluation results are also given.

1 Introduction

A concurrent logic programming language KL1[4] was chosen as the interface between the hardware and software of the parallel inference system under development in the Japanese Fifth Generation Computer Systems project. The language has been proved to be a practical tool of parallel processing software research through the development of the operating system PIMOS[2] and various application software on an experimental parallel inference machine, Multi-PSI[3]. Other implementations on more powerful parallel machines are ongoing.

Such implementations, however, have a serious disadvantage that they are not portable; although they are efficient, they run only on specially devised hardware. A portable byte-code implementation for software development does exist besides, but is too inefficient for practical experimentations. To solve the problem, a scheme of compilation from KL1 to C is investigated. As C, one of the languages whose implementations are most widely available, is chosen as the intermediate target language, the implementation becomes quite portable. An experimental implementation also showed reasonable efficiency.

In what follows, the merits of the scheme, difficulties of efficient translation of a language like KL1 into C and our solutions to the problems are given in the next section, an outline of the experimental implementation is drawn in section 3, and preliminary results of evaluating the implementation is given in 4 followed by concluding remarks.

2 Pros and Cons for Compilation into C

This section first points out merits of the scheme to compile KL1 programs into C code. Then difficulties in obtaining reasonable efficiency with such a scheme are investigated. Finally, our solutions to the efficiency problems are described.

2.1 Merits of Compilation into C

There are various merits in the scheme of compiling into C, in which most important ones are the following.

Portability The greatest merit is that the implementation becomes quite portable. As reasonable C compiler can be found on almost any hardware and operating systems these days, porting such an implementation usually requires re-compilation only.

Low-Level Optimization Another important merit is that some C compilers provide very good low-level optimization. If machine code were to be generated directly, the compiler must be aware of the characteristics of the hardware architecture to obtain reasonable performance. By letting C compiler take care of such low-level issues, the language implementation can almost completely forget about them.

Linkage with Programs in Other Languages Still another important merit is that linking KL1 programs with programs written in C becomes quite easy. In addition, in Unix-like systems where C is "the" language, most other programming languages provide certain interface with C programs. Thus, KL1 programs can also be linked with them without much effort.

2.2 Efficiency Problems

Although compiling programs of another language into language C has the above-mentioned merits, it is not easy to realize efficient implementation when the implemented language is not similar to C, as in case of KL1. Typical efficiency problems are as follows.

Costly function calls The language C and its implementations are designed having in mind that functions are not too small. Although overhead of the function invocations themselves and parameter passing are made rather small in recent implementations, dividing programs into functions makes program analysis more difficult, resulting in less optimal object code. When computation required within one function invocation is large enough, the cost is not problematic. However, like in Prolog, predicates of KL1 is usually very small, usually as small as one line of C. Many of them are recursive, prohibiting inline expansion. If each predicate of KL1 were compiled in to a function of C, the cost of function calls would dominate the total computation cost.

Inability to control register allocation In implementations of languages like KL1, accesses to certain global data, such as the heap top pointer, are made very frequently. It might be best to put such data on some dedicated register if the machine code is to be generated directly. In most C implementations, however, such control of register allocation is not possible. Some compilers (such as gcc) allow this, but using the feature is disadvantageous for portability; obtaining reasonable efficiency on many systems is an important part of "portability".

Cost of provision for interrupts A multi-processor implementation should process interrupts from other processors. Interrupts can be handled as "signals" in Unix-C systems. Interrupt handling requires operations such as allocation of memory area, enqueueing of goals or giving values to variables. Data accessed in these oprations are frequently referenced and altered also in normal processing within a processor. Thus, certain locking on data or inhibition of interrupts will be required, which are usually quite costly in conventional operating systems.

Large object code size When compiled into C and then into the machine code, the object code size tends to become very large, sometimes resulting in performance worse than interpretive code, due to increase of the object code working set.

2.3 Solutions

Our solutions to the above-described efficiency problems are as follows.

Costly function calls To avoid function call costs, one whole "module" that defines a set of closely related predicates, rather than one single predicate, is compiled into one function of C. As far as predicates within the same module are calling one another, no function calls will be made. Arguments can be passed through variables local to the function, which might be allocated on machine registers by C compilers.

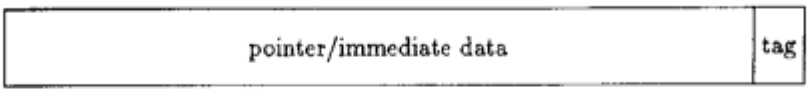

Figure 1: Tagged Pointer

Inability to control register allocation To make accesses to crucial global data with lowest cost, such data are cached to local variables, which C compilers may allocate on registers. Even if not enough registers are available, accesse to local variables are cheaper than to global variables with most modern processor architectures.

Cost of provision for interrupts Signal handlers are made to set a certain flag, which will be examined at certain timing convenient for normal processing. If the flag is found to be set, then the interrupt handling routine will be invoked. This will synchronize interrupt handling with normal processing, without always preparing for interrupts. Further, this flag check is combined with another mandatory check: heap limit check to invoke the garbage collector when needed. Thus, this synchronization can be made virtually without any additional cost. See the following section for details.

Large object code size As modes of variable references are much more easily found for KL1 programs than in Prolog, the code size will not become too large without global analysis. By generating code to call run-time routines for exceptional cases, the object code size can be made reasonable, while normal cases can be expanded in line. Nevertheless, the code size may be still too large for software of practical size. This problem might be solved by a hybrid implementation combining native code compilation with source code interpretion or compilation to byte-coded abstract machine code which will be emulated.

3 Experimental Implementation

This section briefly describes an experimental implementation to evaluate the scheme.

3.1 Data Representation

Every KL1 term is represented as a 32-bit word (Figure 1). The upper 30 bits are used as the pointer or immediate data and the remaining lowest 2 bits are used as data type tags.

The 2 tag bits are used to distinguish the following types.

Variable Reference: The pointer part has the address of a variable value cell, which may be either instantiated or not. Uninstantiated variables are represented as a word with the variable reference tag, pointing to itself, as in WAM.

Atomic Data: For atomic data, 2 more lowest bits of data part are used as the tag extension, which distinguish symbolic atom, integer or floating point data.[1] The remaining 28 bits represents the value.

Cons: The pointer part has the address of a two-word memory block, which holds car and cdr of the cons cell.

Functor: The pointer part has the address of a memory block of a functor structure. The first word of the block contains the functor identifier, from which number of arguments can be known. The rest of the block are the argument values.[2] Any special data structures, such

[1] Floating point numbers and symbolic atoms are not fully implemented in the experimental implementation.
[2] Functors also are not fully implemented yet.

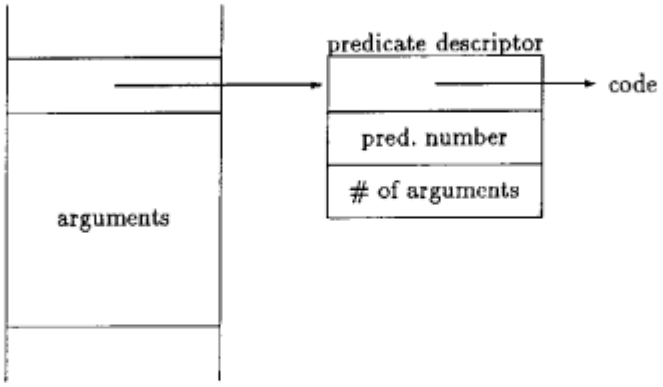

Figure 2: A Goal in a Goal Stack

as character strings, I/O streams, or "merger" structure can be identified by giving them special functor identifier.[3]

3.2 Goal Management

Possibly executable goals are put into a goal stack. A goal stack is a memory block allocated in the heap area. In each reduction step, the top-most goal is popped up from the stack, reduced according to the program, and resultant children goals, if any, are pushed back into the stack.

An entry of a goal stack has, as its first word, a pointer to a predicate descriptor. The predicate descriptor contains a pointer to the module code (a C function), the predicate number within the module, and the number of arguments. The rest of the block contains the arguments of the goal (Figure 2).

There can be multiple goal stacks in the heap, each corresponding to one priority level. The priority mechanism has been found to be very useful through application software research on the Multi-PSI system in describing various algorithms including speculative computation.[4]

When one goal stack overflows, another one is allocated in the heap and they are linked together. The linkage is actually made by pushing in its bottom a special goal which changes the goal stack pointer. Thus, no explicit check is required to detect empty goal stack. Overflow check, on the other hand, is needed.

An uninstantiated variable with goals awaiting for its instantiation is also represented by a pointer with the variable reference tag. The pointer part references a memory block that describes a list of suspended goals. The first word of the block contains special indicator which can be used to distinguish from normal instantiated variables. When such a variable is instantiated during unification, goals suspended should be waken up and pushed into the corresponding goal stacks. This may require allocation of new goal stacks, which in turn may need garbage collection. To avoid this, if more than one goals are suspended, a special goal for awaking such goals will be pushed into the current goal stack. Thus, one unification will allocate at most only one new goal stack in the heap.

3.3 Heap Area Management

The heap area is organized as shown in figure 3.

The heap area is used from both sides. Memory allocations are usually made at the heap pointer downwards. At the end of each reduction, whether the heap pointer exceeds the heap limit or not is checked out. If it does, the garbage collection routine will be called in normal cases.

[3] Built-in mergers, that allow messages to pass through with constant delay for arbitrarily number of input streams, is a crucial feature of the KL1 language.

[4] The priority mechanism is not implemented yet on the experimental implementation.

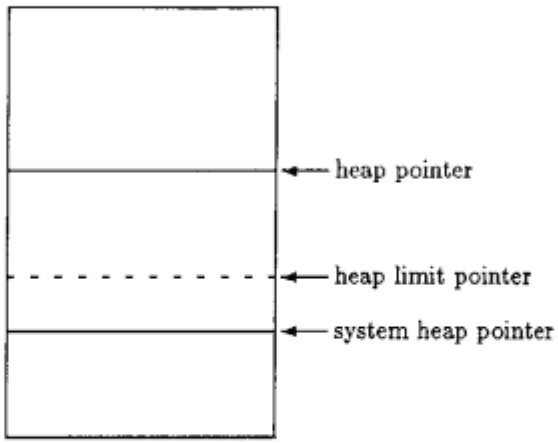

Figure 3: Heap Area

The current garbage collector adopts the copying scheme with certain modification to cope with directly referenced structure elements.

Exceptional allocations, such as allocation of a new goal stack, is made from the other side upwards at the system heap pointer. Thus, the heap pointer can be kept cached in a local variable even when such allocation is required.[5]

When some allocation is made at the bottom, not only the system heap pointer but also the heap limit pointer is moved upwards by the same amount to keep a certain amount of gap between the system heap pointer and the heap limit pointer. The size of the gap is made greater than the maximum amount of memory allocation in one reduction from both sides,[6] so that the area allocated from both sides will never collide with each other during one reduction.

3.4 Interrupt Handling

When some interrupt (actually a signal in Unix) takes place, the interrupt handling routine will set some flag in a global variable and modifies the heap limit pointer so that the next heap limit check at the end of reduction will find it. The check routine can examine the flag and tell whether a garbage collection or interrupt handling (or both) is needed.[7] Thus, no extra check of interrupts is required during normal processing.

The same mechanism can be used to notify newly available goals with higher priority than the currently processed ones.[8]

3.5 Compilation

A prototype compiler from KL1 to C is written in Prolog.[9] It generates simple clause indexing code, but no global analysis whatsoever is made.

One module is compiled into one function of C, which has the following parts.

- Jump according to the top-most goal to the top of the predicate.
- Statement blocks corresponding to predicates defined in the module, each of which contains the following.
  - Statements to pop goal arugments to local variables.

[5] Note that simple unification may require allocation of new goal stack for goals awaken by variable instantiation.
[6] This can be controlled during compilation.
[7] Interrupt handling is not implemented yet.
[8] This can happen during unification.
[9] The current compiler is incapable of compiling almost anything but "naive reverse".

```
:- module(nrev).
a([], Y, Z) :- Y = Z.
a([W|X], Y, WZ) :- WZ = [W|Z], a(X, Y, Z).
n([], R) :- R = [].
n([H|T], R) :- true | n(T, RT), a(RT, [H], R).
```

Figure 4: Source Code of the Tested Program

- Statements to index a clause.
- Statements corresponding to the bodies of clauses, including memory allocations, unifications, goal creations and possibly a tail recursive invocation within the module. When a predicate within the module is to be called, arguments are set up in local variables and the control is tranferred to the top of that predicate's part, after checking the heap limit. Otherwise, when the recursive call is to go outside of the module or when no body goal exists, the control transfers to the last part of the function.
- Statements for suspension and failure. This part is jumped in from the indexing code.

• Statements for proceeding when no invocations of predicates within the module is in the body of the selected clause. First, the heap limit is checked and the garbage collector or the interrupt handler will be invoked. Then, the top of the goal stack is examined, and if the top-most goal has the predicate descriptor with the pointer to the same module as one currently running, the control is tranferred back to the predicate dispatching part. Otherwise, the function returns and the top-level routine will dispatch to the function of the module for the top-most goal.

The statements for unification expanded inline is only for cases when one of the arguments of the unification[10] is an uninstinatiated variable *without* goals awaiting for its instantiation. All other cases are treated by an invocation of the general unification subroutine.

4 Performance Evaluation

4.1 Systems

The implementation has been ported to the following systems.

Sun-3/260 running Sun UNIX 4.2 Release 3.5, compield with GCC.

SparcStation 1+ running SunOS Release 4.1.1, compiled with manufacturer-provided CC.

SparcStation 2 running SunOS Release 4.1.1, compiled with manufacturer-provided CC.

Sequent Symmetry running Dynix V3.0, compiled with GCC.

4.2 Tested Program

The only program tested so far is "naive reverse" of thirty elements. The source code is as given in Figure 4.

4.3 Execution Speed

The execution speed is measured by running thirty element naive reverse programs repeated 2,000 times, which requires 992,000 reductions. The execution speed is greatly affected by the heap size, but best figures are as shown below.

[10] Left hand side, in the current compiler.

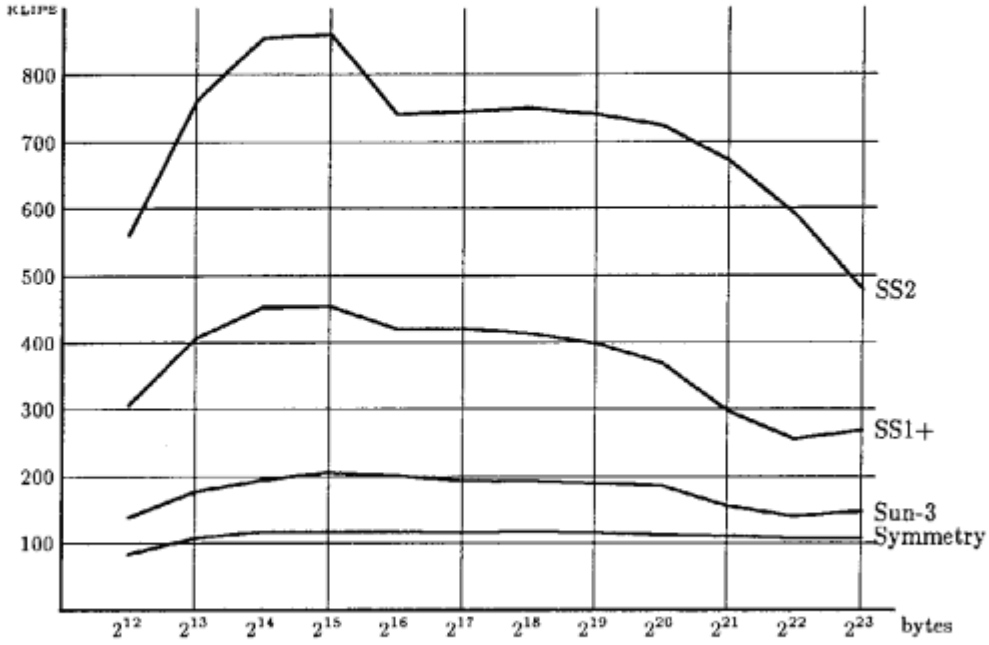

Figure 5: Effect of Heap Size to Execution Speed

System	Performance
Sun-3	205,809 LIPS
SS1+	455,045 LIPS
SS2	861,111 LIPS
Symmetry	116,431 LIPS

The tighest loop of the program is in the "append" predicate. The machine code traces of the loop are given in appendices.

As stated above, the performance figures are affected greatly by the heap size (Figure 5). With small heap, frequent garbage collection degrades the performance. With large heap, on the other hand, increased working set seems to make problems; frequent cache miss hit may be making system run slower. This tendency is found monst strong with SparcStation 2, where the speed ratio of the cache and main memory is larger.

In case of SparcStation 2, the highest figure obtained was about 861 KLIPS with heap size 32 KB, which is about half the size of the cache memory. As the current implementation uses copying scheme for garbage collection, it means that the whole heap area almost fits in the cache. With 8,192 KB of heap, the performance degrades to about 479 KLIPS, which is less than 56% of the highest figure.

This seems to indicates that incremental garbage collection such as MRB[1] is profitable in KL1 implementations. The performance degradation of 56% means almost 80% increase in execution time. Even if instruction steps were to be increased by incremental garbage collection by 50%, it would pay off if the working set size could be kept much smaller by the effort.

The MRB scheme was not used in this experimental implementation mainly because one extra tag bit needed for the scheme could not be allocated in a tagged word without slowing down processing drastically. Specially divised hardware architectures, like Multi-PSI or PIM's have no problem here.

More compilation time analysis should enable reusage of memory area without run-time anlaysis. A sample coding (by hand) assuming such an analysis showed that the performance of 1,048 KLIPS on SparcStation 2.

4.4 Object Code Size

The object code sizes generated by the C compiler is shown in Table 1. These figures include code for entry and leaving C functions. The sizes are not prohibitively large, but if a large programs are to be run on the system, using slower but much more compact interpretive or byte-compiled code together with native code might be profitable.

Table 1: Object Code Size of Naive Reverse

System	Text Size	Data Size
Sun-3	516 bytes	24 bytes
SS1+ & 2	800 bytes	24 bytes
Symmetry	536 bytes	24 bytes

5 Conclusion

A scheme for portable implementation of KL1 on systems with conventional architecture is described. The scheme employs the strategy to compile KL1 source code into C for increased portability, rather than directly generating machine code tailored for each machine architecture. The preliminary implementation shows that reasonably efficient implementation is possible even by such a scheme.

Further efforts in various directions are yet to be made for a system practical for parallel software research, including the following.

- Full implementation of various language features
- Multi-processor implementation
- Providing tools to support software development

References

[1] Takashi Chikayama and Yasunori Kimura. Multiple reference management in flat GHC. In *Proceedings of 4th International Conference on Logic Programming*, 1987.

[2] Takashi Chikayama, Hiroyuki Sato, and Toshihiko Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of FGCS'88*, pages 230–251, Tokyo, Japan, 1988.

[3] Yasutaka Takeda, Hiroshi Nakashima, Kanae Masuda, Takashi Chikayama, and Kazuo Taki. A load balancing mechanism for large scale multiprocessor systems and its implementation. In *Proceedings of FGCS'88*, Tokyo, Japan, 1988. Also in New Generation Computing 7–2, 3 (1990), pp. 179–195.

[4] Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, December 1990.

A Trace of append loop on SUN-3

```
append_3_loop:
 deref_and_switch(arg0, ...);
       1  L10:   btst #1,d2          ;arg0 has REF or ATOM tag?
       2         jne L60             ;no, branch
       3  L60:   btst #0,d2          ;arg0 has CONS tag?
       4         jne L20             ;yes, don't branch
 work0 = makecons(allocp);
       5         movel a2,d1         ;heap top pointer to work0
       6         addql #2,d1         ;put on CONS tag
 *allocp++ = car_of(arg0);
       7         movel d2,a4         ;arg0 to a4 reg
       8         movel a4@(-2),a2@+  ;car of it to heap top
 work1 = makeref(allocp);
       9         movel a2,d3         ;heap top pointer to work1
 *allocp++ = work1;
      10         movel a2,a2@        ;make self reference
      11         addqw #4,a2         ;advance heap pointer
 unify(arg2, work0);
      12         movel a1,d0         ;examine arg2
      13         moveq #3,d4         ;prepare tag mask
      14         andl d4,d0          ;get tag part of arg2
      15         jne L23             ;if REF, don't branch
      16         cmpl a1@,a1         ;if self referencing var
      17         jeq L22             ;do branch
      18  L22:   movel d1,a1@        ;store new cons to value cell
 arg0 = cdr_of(arg0);
      19         movel d2,a4         ;prepare new arg0 for recursion
      20         movel a4@(2),d2     ;by taking cdr of arg0
 arg2 = work1;
      21         movel d3,a1         ;new arg2 will be the new variable
 execute(append_3_loop);
      22         cmpl _heaplimit,a2  ;check heap overflow
      23         jcs L10             ;with no problem, loop
```

The identifiers in the source program are slightly modified for readability.

B Trace of append loop on Sequent Symmetry

```
append_3_loop:
 deref_and_switch(arg0, ...);
       1  L61:  testl $2,%ebx          ;arg0 has REF or ATOM tag?
       2        je L18                ;no, don't branch
       3        testl $1,%ebx         ;arg0 has CONS tag?
       4        jne L20               ;yes, don't branch
 work0 = makecons(allocp);
       5        leal 2(%esi),%eax     ;heap top with CONS tag to work0
 *allocp++ = car_of(arg0);
       6        movl -2(%ebx),%edx    ;get car of arg0
       7        movl %edx,(%esi)      ;move it to heap top
       8        addl $4,%esi          ;advance heap top pointer
 work1 = makeref(allocp);
       9        movl %esi,-4(%ebp)    ;heap top pointer to work1
 *allocp++ = work1;
      10        movl %esi,(%esi)      ;make self reference
      11        addl $4,%esi          ;advance heap pointer
 unify(arg2, work0);
      12        testl $3,-8(%ebp)     ;arg2 has REF tag?
      13        jne L23               ;yes, don't branch
      14        movl -8(%ebp),%edx    ;load arg2 to reg
      15        cmpl (%edx),%edx      ;arg2 is self referencing?
      16        je L22                ;yes, do branch
      17  L22:  movl -8(%ebp),%edx    ;load work0 to reg
      18        movl %eax,(%edx)      ;store new cons to value cell
 arg0 = cdr_of(arg0);
      19        movl 2(%ebx),%ebx     ;let arg0 be cdr of arg0
 arg2 = work1;
      20        movl -4(%ebp),%edx    ;load work1 to reg
      21        movl %edx,-8(%ebp)    ;store to arg2
 execute(append_3_loop);
      22        cmpl _heaplimit,%esi  ;check heap overflow
      23        jb L61                ;with no problem, loop
```

The identifiers in the source program are slightly modified for readability.

C Trace of append loop on SparcStations

```
append_3_loop:
 deref_and_switch(arg0, ...);
       1  LY14:   andcc %i5,2,%g0                ;arg0 has REF or ATOM tag?
       2          be LY17                        ;no, don't branch
       3          andcc %i5,1,%g0                ;arg0 has CONS tag?
       4          bne,a L77038                   ;yes, don't branch
 work0 = makecons(allocp);
 *allocp++ = car_of(arg0);
       5          mov 17,%i5                     ;delay slot not executed
       6          ld [%i5-2],%o7                 ;get car of arg0
       7          add %i4,2,%i0                  ;make cons pointer
       8          st %o7,[%i4]                   ;store car of arg0 to heap top
       9          inc 4,%i4                      ;advance heap top pointer
 work1 = makeref(allocp);
      10          mov %i4,%i2                    ;heap top pointer to work1
 *allocp++ = work1;
 unify(arg2, work0);
      11          mov %i1,%i3                    ;test arg2
      12          andcc %i3,3,%g0                ;whether it has REF tag
      13          st %i2,[%i4]                   ;store work1 to heap top
      14          bne L77022                     ;arg2 is REF, don't branch
      15          inc 4,%i4                      ;increment heap pointer
      16          ld [%i3],%l0                   ;get contents of value cell
      17          cmp %l0,%i1                    ;self reference?
      18          be,a LY16                      ;yes, do branch
      19          st %i0,[%i3]                   ;store value (in delay slot)
 arg0 = cdr_of(arg0);
      20   LY16   ld [%i5+2],%i3                 ;take cdr of arg0
 arg2 = work1;
 execute(append_3_loop);
      20          sethi %hi(_heaplimit),%l1      ;load heap limit
      21          ld [%l1+%lo(_heaplimit)],%l1   ;on a register
      22          mov %i2,%i1                    ;set up arg 2
      23          cmp %i4,%l1                    ;compare with heap pointer
      24          blu ,a LY14                    ;branch if no problem
      25          mov %i3,%i5                    ;set up arg0 (in delay slot)
```

The identifiers in the source program are slightly modified for readability.