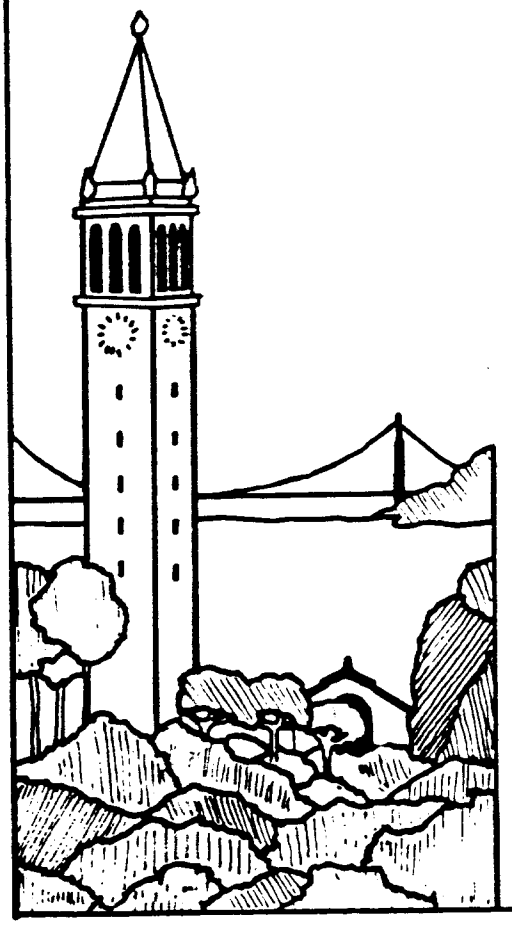
Smalltalk-80 to SOAR Code

William R. Bush



Report No. UCB/CSD 86/297

June 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Smalltalk-80 to SOAR Code

William R. Bush

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley

ABSTRACT

The SOAR (Smalltalk On A RISC) project at Berkeley has developed a RISC-style processor designed to execute the Smalltalk-80 language efficiently. This document describes the translator that converts Smalltalk bytecodes into SOAR instructions. Experience with the translator justifies early SOAR assumptions and design decisions.

Smalltalk-80 to SOAR Code[1]

This document describes the Smalltalk-80[2] to SOAR code translator. The first section discusses the implementation of the translator. Section two presents performance results for the translator. The third section describes issues in register allocation, one of the most important aspects of the translator. Section four deals with the experience of developing a moderate system in Smalltalk. The reader is assumed to be comfortable with Smalltalk-80 language concepts and terminology, and have some understanding of the Smalltalk-80 virtual machine.

1. Translation

The purpose of the SOAR (Smalltalk On A RISC) project is to produce an efficient Smalltalk implementation through minimal extensions to the Berkeley RISC architecture [SOAR-Arch]. The SOAR processor has the basic structure of RISC[3] as well as RISC-style register windows.[4] Smalltalk processing is enhanced through tagging basic data items (small integers and pointers) and trapping on certain exceptional conditions (which allows register-to-register integer operations and efficient garbage collection). The SOAR architecture does not implement the Smalltalk-80 virtual machine [BlueBook]; it therefore does not interpret Smalltalk-80 bytecodes. The Smalltalk-80 language must instead be translated into SOAR instructions.

Three approaches to translation were possible: produce a new compiler that would take Smalltalk-80 source and produce SOAR code; modify the existing Smalltalk compiler to produce SOAR instructions rather than Smalltalk-80 bytecodes; or add a backend stage to the existing Smalltalk-80 compiler that would take compiled

---

[1]This work was supported in part by Xerox PARC.

[2]Smalltalk-80 is a trademark of Xerox Corporation.

[3]The RISC architecture is register-oriented, with loads and stores for moving data between registers and memory, and register-to-register instructions for all operations on data [RISCs].

[4]Each procedure invocation has its own set of registers, allocated in a stack discipline and overlapped with both the last invocation and the next one (for passing parameters) [VLSI-RISC].

bytecode methods and produce SOAR methods. The backend approach was chosen for several reasons. First, the stack-based bytecodes that the Smalltalk-80 compiler emits are the basis of the Smalltalk-80 virtual machine, which is the semantic specification of the Smalltalk-80 language. Thus the bytecodes are well-defined and provide a good interface. Second, the semantics of Smalltalk preclude most standard optimizations. For example, expressions cannot even be reordered,[5] since the identity of the receiver must be preserved. Also, bytecode operation is relied on by the Smalltalk system; in particular, copies of variable values are assumed to be made as if the bytecode stack were present.[6] The result of these optimization constraints is that bytecodes are an acceptable intermediate representation. Third, the Smalltalk-80 virtual image is quite malleable, and tracking changes in it to keep a new or modified standard compiler current could pose a maintenance problem. At Xerox, for example, a new image used by about 20 researchers is released roughly every two months, and the changes in a new image can be substantial and subtle. Fourth, keeping bytecode methods allows mixed-mode debugging, with some methods interpreted as bytecodes and some compiled to SOAR. The main disadvantage of the bytecode-to-SOAR conversion approach is compilation speed -- it is guaranteed to be slower than the standard compiler alone. However, the decision to build a converter was justified by later experience.

1.1. Conversion

The basic idea behind the converter is to simulate a method's runtime stack during conversion. This is a conventional idea that is found in p-code translators ([Sopaipilla], for example). Stack operations can be divided into two classes: those that only manipulate the stack (for example, push a local variable on the stack, or add the top two stack elements and push the result), and those that store or otherwise use stack elements (for example, pop the top of stack into a local variable, or jump if the

[5]To be precise, neither operand nor operator evaluation can be reordered. Both arguments to a send and a sequence of sends must be evaluated in a predefined order.

[6]For example, a method can temporarily preserve values (from receiver variables) from modification during a send (if they are altered as a side effect of the send) by using the stack.

top of stack is equal to true). Only operations of the latter type require actual values and provide concrete non-stack destinations, so they trigger code generation. Consider, for example, the sequence

```
push A
pop into B
```

During conversion, the push causes the source to be remembered. The pop supplies a destination, so the description of the source on the top of the stack is retrieved and the appropriate transfer code, based on the nature of the source and sink, is generated.[7] Now consider

```
push A
push B
send + (add)
pop into C
```

When the add is encountered the two top elements on the simulated stack are combined into one new element that contains the two source operands and the add operator. When the pop is encountered and the destination of the add is finally known, the add instruction is generated.[8] Code is not generated when the add is encountered because its ultimate destination is not then known; since that destination may be a register, it would be foolish to put the result of the addition into a temporary register, only then to move it to its final location. Bytecodes that only affect the simulated stack are the pushes of various types of variables, and in-line sends -- operators that are sends in Smalltalk but are register-to-register instructions in SOAR (arithmetic, logical, and relational operators). All other bytecodes (pop into,

---

[7]If A and B were both registers, the resulting SOAR code would be

```
add     rzero,rA,rB
```

which causes the contents of register A to be moved into register B by adding zero (the special register rzero always contains zero) to A and putting the result in B. If A and B were both receiver variables (in memory), the resulting SOAR code would be

```
load    (rR)oA,rT
store   rT,(rR)oB
```

where rR is a register holding the object pointer of the receiver, oA and oB are the offsets of A and B in the receiver, and rT is a temporary register.

[8]If A and B were both registers, the resulting SOAR code would be

```
add     rA,rB,rC
```

store into, send, return, jump, and conditional jump) cause code to be generated and either provide destinations (implicitly in the case of send -- the arguments go in certain registers) or use temporaries.

The converter knows the best code sequences for transferring every variable (receiver, literal, method) into and out of a register. Given this knowledge, when moving a value from a source to a destination the converter considers whether either or both are registers. If both are, a register-to-register move is produced; if one is, it is used as the source or destination in the optimal register transfer code of the other; if neither are, a temporary register is allocated and used in the optimal register transfer code of both. This technique, combined with the generate-no-code-before-its-time (before its destination is known) approach discussed above, results in optimal code for data movement and in-line send expression evaluation.

Sometimes values must be copied before their final destination is known. This occurs for two reasons. First, the Smalltalk system occasionally assumes a stack implementation and modifies variable values presuming that stack copies exist. This requires that all values on the simulated stack be in actual temporaries before a send is done. Second, jumps require the merging of threads of control, which is accomplished by forcing the simulated stack elements into temporaries before the jump. Since the temporaries are allocated in a stack discipline, uniformity is achieved.[9]

---

[9]For example, since Smalltalk is an expression language and ifTrue:ifFalse: is expanded inline, the fragment

```
v1 <- (a1 ifTrue: [ v2 <- a1 ] ifFalse: [ v2 <- a2 ]) + a2.
```

(where a1 and a2 are method argments, and v1 and v2 are receiver variables) translates into the bytecode sequence

```
        push a1
        jump-if-false L1
        push a1
        store into v2
        jump L2
L1:     push a2
        store into v2
L2:     push a2
        send +
        pop into v1
```

At L2 either a1 or a2 is on the top of the stack, depending on the value of a1. With arguments a1 in r13 and a2 in r12, and with receiver variables v1 in (rR)4 and v2 in (rR)5, the following SOAR code is

The generate-no-code-before-its-time approach does assume that physical copies of values need not be made for every push and pop. For example, the sequence

```
push A
store into B
push C
pop into A
pop into D
```

must not occur,[10] since it relies on the stack copy of A for the pop into D. With the converter, A is not copied but simply moved into D when the pop is encountered.

The converter is essentially one pass, but occasionally must make two passes over the bytecodes. This occurs when the number of converter temporaries exceeds the number of available SOAR registers and a spill area in memory must be created for them. Since spill areas are created only at the beginning of methods and the number of temporaries can only be known after conversion, a few methods must be converted again when the number of temporaries is known. This number is the only information kept from one pass to the next, and if a spill area for the temporaries need not be created, then the first pass is complete in itself.

Blocks are processed with the same lazy look-behind approach that is used for producing optimal register transfer code. Blocks, and their surrounding code, have the form

---

generated. Note that r11 holds the first operand to the addition (the equivalent to the top of the stack).

```
        skip    ne r13,#false
        jump    L1
        store   r12,(rR)5
        add     rzero,r13,r11
        jump    L2
L1:     store   r12,(rR)5
        add     rzero,r12,r11
L2:     add     r11,r12,r0
        store   r0,(rR)4
```

[10]The sequence 'must not occur' in the sense that, if it should occur, incorrect SOAR code will be generated. The bytecode compiler does not currently generate such seqences, nor do they make sense with the current Smalltalk-80 semantics.

```
        push N
        send blockCopy
        jump L
        <code for block>
L:      ...
```

where N is the number of block arguments. As N and the blockCopy operation are encountered they are pushed on the simulated stack. When a jump is seen the top of the stack is examined; if it is a blockCopy then block copy code is produced and the block is processed. Block code is handled like method code, except than each block requires its own simulated stack (equivalent to its block context), and receiver and method variables are accessed differently.

1.2. Optimizations

Although, as mentioned above, many optimizations are not allowed, a few straightforward optimizations are performed. One involves comparisons and conditional jumps. Consider the bytecode sequence

```
push A
push B
send < (compare)
jump-if-true L
```

When the jump is encountered the top of the stack is either true or false, depending on the result of the comparison. This behavior could be duplicated in SOAR code, with the result of the comparison being either a true or false, which would in turn be tested for the jump. It is simpler, however, to combine the comparison and the jump test, and avoid the intermediate boolean result. Thus

```
add     rzero,#true,rT
skip l  rA,rB
add     rzero,#false,rT
skip n  rT,#true
jump    L
```

(assuming A and B are in registers) becomes

```
skip ge rA,rB
jump  L
```

This optimization is done using look-behind. No code is generated when the compare is encountered; it is simply put on the simulated stack. When the conditional jump is encountered the top of stack is checked for a relational operation. The optimized code is then generated, reversing the skip test. If the top of stack had not been a relational operation it would have been moved into a temporary register and then tested against true.

Another optimization involves using registers when possible. Consider

```
push A
store into B
pop into C
```

where A is a receiver variable that resides in memory and B is a local register variable (the nature of C is irrelevant). Rather than fetch A twice, once for B and once for C, the converter fetches A once into B and then uses B as the source for the move into C. This optimization requires that B not be changed after the first fetch of A. For example, the sequence

```
push A
store into B
push Z
pop into B
pop into C
```

must not occur.

The converter also puts small (plus or minus 128) integer constants in-line because the SOAR instruction format allows it. The Smalltalk-80 bytecode set only treats a few values (-1, 0, 1, 2) as special, putting all other integer constants in the literal frame. Inserting moderate-sized integers in-line avoids references to the literal frame, which require a load in SOAR. It is possible to generate the special small integer code because the literal frame is read-only.

In addition, the converter generates in-line cache code for sends, which avoids most send lookups. See [SOAR-Daedalus] for details.

1.3. Implementation

The converter is written in Smalltalk and consists of five main pieces, which correspond to five class hierarchies under class Object.

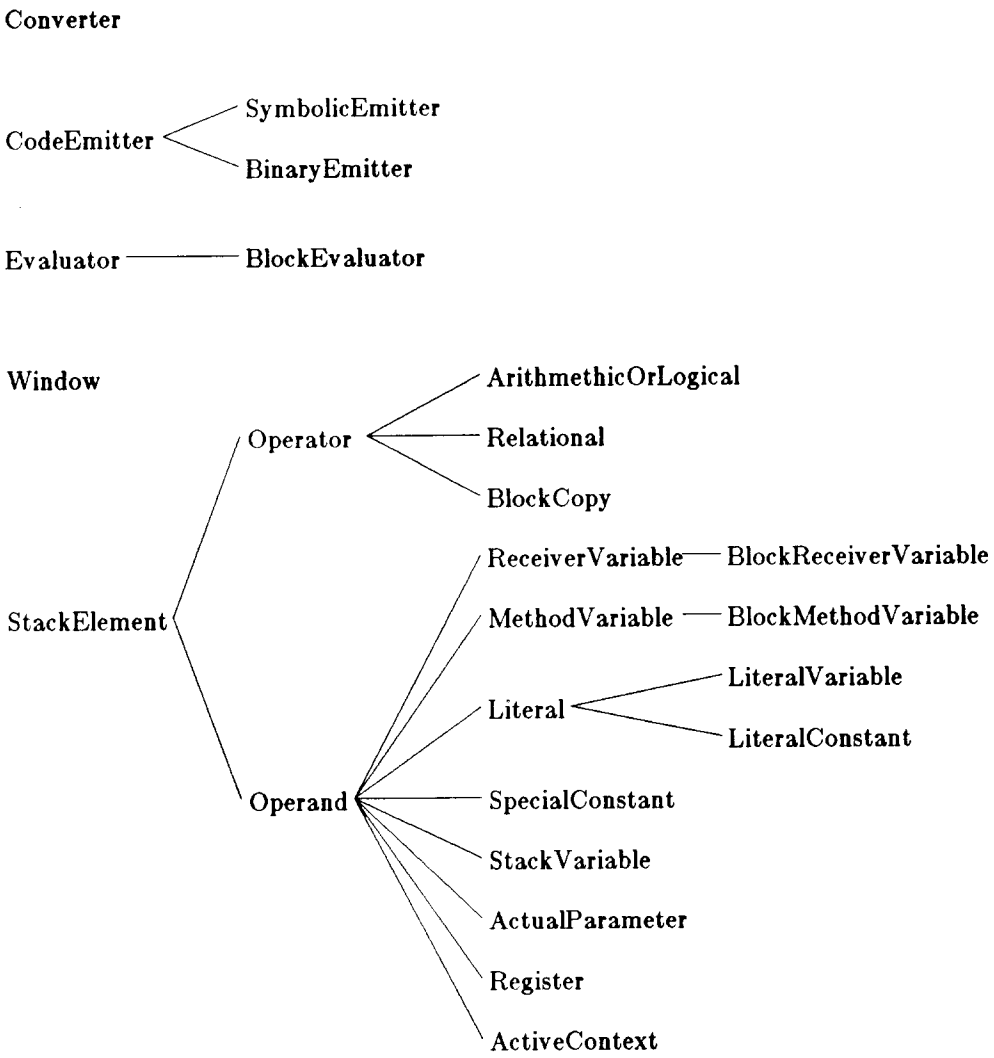


Figure 1: Smalltalk Class Structure of the Converter

The Converter class understands bytecode methods. It is responsible for parsing and interpreting the bytecodes in a source method. It also controls the overall conversion process, and manages method-based information (such as method size statistics).

The Emitter classes understand SOAR instructions and methods. They produce either a symbolic method that can be given to the SOAR assembler, or a binary method compatible with the SOAR debugger.

The Evaluator class knows what individual bytecodes do. It manipulates the simulation stack, pushing and popping elements, and generates code for send, return, and jump. Its subclass understands blocks -- in particular it knows about method and block register windows and how to generate special block code.

The Window class manages information about register windows. It knows about window dimensions and special purpose registers, and does register allocation. It knows when there are too many variables to put in registers, and decides which are to be put in memory.

The StackElement class hierarchy knows how to generate code to move data into, between, and out of registers. The evaluateTo: method in StackElement determines the optimal movement path (based on whether the source and destination are in registers or memory), and then either generates a register-to-register transfer or invokes the appropriate register-based getInto: or putFrom: method defined in the corresponding element subclass.

2. Converter Performance

Three performance results of the converter are of particular interest: conversion speed, the ease of modifying the converter to run in a new virtual image, and code expansion. In general, the design decisions made appear to be justified by the results.[11] The measurements in this section were made using two different virtual

[11]All results were obtained on a Dorado [Dorado], a very fast personal workstation developed at Xerox.

images -- the one licensed by Xerox and the one used daily by researchers at Xerox.

There are two measures of converter speed, objective and subjective. Objectively the converter is reasonably fast.

image	mean time (in milliseconds)	total number of methods
licensed	50	4770
experimental	68	8069

These times are for the converter only, operating on existing bytecode methods.[12] In comparison, the compile macro benchmark provides statistics for the other phases of compilation.[13]

phase	cumulative time (milliseconds)
syntax	320
bytecode method generation	450
SOAR converter pass one	610
SOAR method generation	760

Note that most methods (94% for the licensed image, 91% for the experimental image) require only one pass.

Subjectively, the converter does not intrude on interactive system use; most of the time compilation is invoked interactively one method at a time. The standard compiler was modified to include an invocation of the converter (which generated a SOAR method and then discarded it, returning the bytecode method), and the slowdown was not perceived (on a Dorado). A special cursor indicated when the converter was running, noticeable only on long methods.

Modifying the converter to run in the experimental Xerox image was trivial. A few simple changes were necessary to produce symbolic output and converter statistics output. One unexpected problem arose, however. The Smalltalk compiler in the experimental image occasionally produced incorrect code that did not affect the running system but broke the decompiler and the converter. This points up a weakness in the backend approach -- dependence on the standard compiler's output code. This is generally a reasonable assumption, however, since the system depends on

[12]The difference between the licensed and experimental image times is probably due to the longer average length of the experimental image's methods and the greater number of its two-pass methods.

[13]These times were obtained with the licensed image.

it. In contrast, the language in the experimental image is still considered by the researchers to be experimental and is in the process of changing in subtle ways that would require subtle changes to a pure SOAR compiler or a modified standard compiler.

One of the virtues of bytecodes is that they are a compact representation. A major concern with the converter was the explosion in method size that could result from moving to four-byte word-sized instructions. Preliminary results with a trial SOAR compiler indicated potential expansion of up to 1000% [SOAR-292R-Citrin]. Fortunately, observed expansion with the converter is considerably lower[14]

image	bytes (mean)	words (mean)	expansion
licensed	32.5	40.6	500%
experimental	38.3	46.9	490%

The 10% difference may be significant. SOAR methods have a relatively high fixed overhead, and as methods become larger, as they did on average in the experimental image, the bytecode advantage becomes less.

3. Register Allocation

A SOAR register window consists of 16 registers, divided into two sets of 8, called the high set and the low set. Two registers in each set are used by the hardware for the receiver and return address, leaving 6 for the converter and runtime system to use for arguments, local variables, and temporaries. The high 6 belong to the executing method (persist during the method's dynamic existence). Currently the converter uses up to 4 of these for arguments and locals, and the remainder for temporaries. The low 6 are used for arguments to called methods and for transitory temporaries that will not persist during a send (because they become the callee method's high registers).

[14]The method lengths include literals and in-line cache instructions. To put these averages in perspective, total method storage for the bytecode licensed image is 155025 bytes, 774648 bytes for the SOAR version, while the total for the experimental image is 309043 bytes, and 1513744 for the SOAR version. Total image sizes are not available.

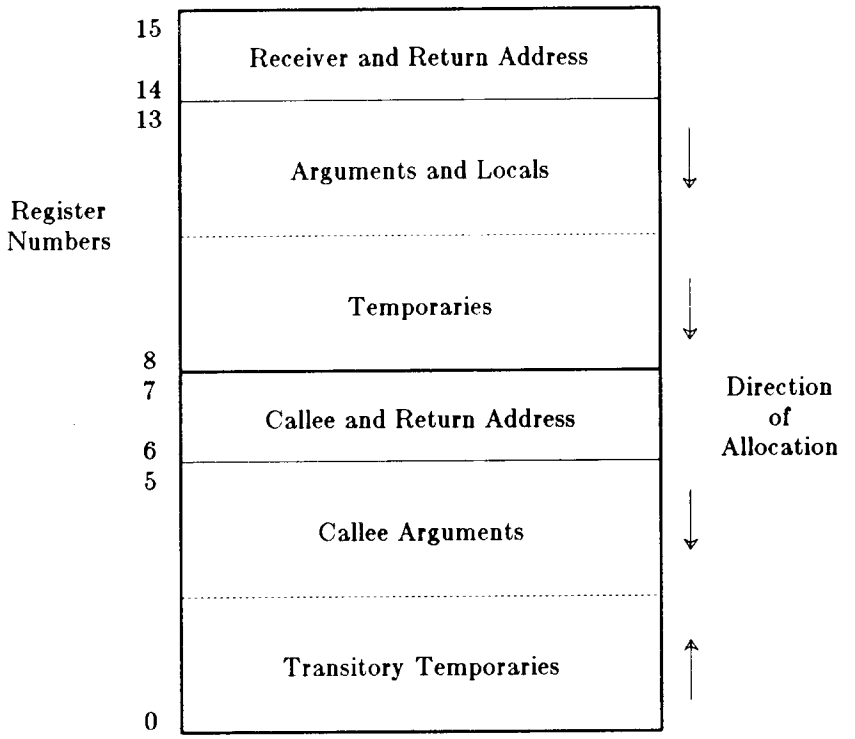


Figure 2: Register Allocation in a SOAR Register Window

All but the transitory temporaries need not be registers, and in fact may not be. An area of memory is available to hold excess arguments, local variables, and temporaries -- called spills in general. Three rules determine when to spill which values. First, if there are more than four arguments they are all spilled (otherwise none are). Second, if there are more local variables than will fit in the unused argument registers, they (the locals) are all spilled (otherwise none are). Third, whatever registers remain are used for temporaries; if there are more temporaries than registers only the excess temporaries are spilled. These spill rules are governed by two principles, simplicity (spilling entire classes of variables) and avoidance of moving established values (keeping arguments in registers if possible). For blocks, the block arguments are moved to the method's local variables, at which point all the block

window's registers are available for temporaries.

The transitory temporaries are used for expression evaluation, indirect pointers, and values that must be moved between two memory locations. They must be registers. They are allocated from the pool of low registers that are not being used to hold arguments for the next send. Some invariants are maintained in the converter code to insure optimal allocation: indirection temporaries are only used briefly (in a single sequence of instructions emitted at once), and other transitory temporaries are held only during the fetch phase of code generation (because destinations are always known before code generation is started).

A major concern was the extent to which values would have to be spilled, since memory operations are more expensive than register operations. Preliminary results indicated that 8 registers per method (16 to a window, overlapping caller and callee) would be adequate [SOAR-292R-Blakken], but the study did not take into account compiler temporaries. Static results from the converter support the current window size. In the licensed image only 9% of the methods spill, rising to 11% in the experimental image. For the licensed image,[15]

number of registers	methods using no more than that number
2	29%
4	65%
8	92%
16	99%
32	100%

One method would need 30 registers to avoid spilling (31 in the experimental image). Given that the window size must be fixed, 8 registers per method is a reasonable choice.[16]

[15]These numbers include the 2 special receiver and return address registers. Note that the percentage of methods that spill is higher than the percentage of methods that require more than 8 registers because the registers are divided up into two regions (arguments and locals, and temporaries) that have different spill disciplines.

[16]An alternative that would avoid spilling entirely would be variable sized windows, the sizes of which could be based on powers of two for hardware speed. It can be argued that low absolute limits should be put on the number of arguments, locals, and converter temporaries, since good Smalltalk code is (or should be) highly modular.

Some investigation was done (with the licensed image) into the effect of changing the number of registers used for arguments and local variables.

number of regs	nonspilling methods	mean number of spills	mean method size
3	86%	.59	41.06
4	90%	.45	40.64
5	92%	.35	40.32

This indicates that 5 registers may be better than the current 4. Both the mean number of spilled values per method and the mean method size are smallest for 5. These results are only indicative, however, since dynamic referencing behavior is more important -- if temporaries are referenced more frequently than the arguments and locals then they should be put in registers.

Another issue relates to the management of the spill area. Currently there is one large area of memory for spills, allocated in a stack discipline. Each method that spills values allocates a 32 word frame from the spill area through a global register. Another technique is to allocate a separate spill object for each method that spills. Each object would contain exactly the number of spills for its associated method. This scheme would require that one high register be used to point to the spill object. The effect of this change was explored, with the results indicating that, from the point of view of register use, there is no appreciable difference. The scheme does not change the number of methods that spill; rather it increases by one the number of spilled values for some of those methods that do spill. The mean spill size increased from .45 for the current technique to .48 for the spill object technique.

Since fixed-size memory allocation is typically faster, it is probably adequate.

4. Experience with Smalltalk

The converter is an implementation of a known solution to a well-understood problem. What I found interesting during the project was the Smalltalk-80 language and environment in which the converter was developed.

The Smalltalk-80 user interface is, as advertised, easy to learn, simple to use, and powerful in functionality. It is highly uniform (for example, each of the mouse buttons controls the same type of activity in all contexts, and there are a few basic

types of windows, such as editable text and lists), and has well-chosen primitives. I have never enjoyed the process of programming as much as I did using Smalltalk on a Dorado.

The Smalltalk-80 language strongly encourages a distinctive, elegant programming style. This style takes some time to acquire (about six months in my case), and is hard to appreciate without the Smalltalk environment.

In a conventional language, the code in a module often provides a service that defines a few data structures (not just one), and case and else-if statements encourage large procedures. Smalltalk programs tend to be super-modularized [Interfaces], with classes and methods being smaller and more limited in effect than their equivalents in a traditional procedural-stew language. My experience with the converter is not uncommon for neophyte Smalltalk-80 programmers. The first converter implementation was an (unwitting) attempt to use Smalltalk in the style of a conventional procedural language. The simulation stack elements were implemented as tagged symbols, and their processing was done through a case-like dispatch. The converter was one class. It worked, but clearly did not take advantage of Smalltalk's organizing features. The second implementation did away with the dispatch; instead, each type of stack element was represented by a separate class, and at the appropriate time was sent a message to convert itself. Deliberately, however, all code generation was centralized for maintainability. This proved to be clumsy, and the converter was reorganized (with virtually no changes to the methods) into its final form. Exposure to Smalltalk-80 system code undoubtedly affected my style, but I also feel that the language itself (much more than any other I've used) makes one aware of the difference between good code (clear, well-organized) and bad code (confusing, poorly organized).

The current converter is reasonably well-structured Smalltalk. The average method body is small (4 lines of code), and the system is evenly distributed among the class hierarchies.

class group	number of methods in the group	lines of code
converter	34	365
evaluator	69	413
emitter	59	247
window	45	227
operand	69	440
total	276	1692

A major consequence of Smalltalk's super-modularized style and polymorphism is that the Smalltalk-80 environment [OrangeBook] is needed to understand readily any substantial amount of code. In a conventional language procedures are larger, making the context in which one must understand a statement more localized and linear. In Smalltalk, methods are typically small, connectivity with other methods is high, and selectors can be ambiguous. The browser, through its senders-implementors-messages functions, and the debugger, via locus of control, are necessary to connect, disambiguate, and to a certain extent linearize Smalltalk program fragments. A large paper listing of Smalltalk code is almost intractable. My first experience with the virtual image was on paper, and it reminded me of the U.S. Internal Revenue Code. I have a law degree, and studying the tax code involved the same bewildering collection of small, interconnected fragments. One doesn't find a substantial piece that solves the problem one is concerned with; rather, one skims quantities of fragments, remembering vaguely what they do. When a critical mass has been scanned, one experiences the legal epiphany, and sees the framework and the relevance of the pieces, understanding in the process much more than the solution to the original problem. With the Smalltalk environment, on the other hand, one can easily find the set of relevant fragments -- but the system can only be easily studied interactively.

Unfortunately, the Smalltalk vehicle used for much of the converter's development was BS on a Sun 1.5, which could be used to get a sense of the Smalltalk system, but was too slow[17] (and unreliable, due to various Sun problems)[18] for

[17]It should be noted that BS was not intended to be fast, but rather an elegant UN*X implementation in C that would serve as a vehicle for testing storage management strategies.

[18]An amusing one beset a Sun named Phoenix, which caught fire. Interestingly, no workstation I've used (Dorado, Sun, or Apollo) has been as reliable as mainframe timesharing. I've lost nontrivial amounts of work (taking more than an hour to replace) on each.

substantial development[19] (the converter took as long as 6 hours to compile, about 5 lines per minute). The major impact that slow speed has on program development is inhibiting revision. When trivial errors take hours to repair, needed changes are often not undertaken.

Slow speed can also make a truly interactive system slower than an equivalent traditional one from the user's point of view. In particular, Smalltalk requires actual interactive mouse input (there is no mouse-ahead), while a conventional time-sharing system provides some input buffering, so that the user can ignore the machine for intervals when the machine is heavily loaded.[20] On the other hand, a fast interactive system really does provide more leverage. The BS Smalltalk-80 implementation is roughly 10 times slower than a Dorado, and I found my productivity roughly increase by that amount when I moved from one to the other.

Smalltalk on a Dorado is just plain fun.

Acknowledgements

I would like to thank Dave Ungar for BS and his interest in Smalltalk, Glenn Krasner for the use of his Dorado and his interest in this paper, and Greenbrier for (to paraphrase Knuth) several pleasant evenings.

[19]Due to bugs in the Sun 1.5 paging routines (causing the system to hang) and the disk software (resulting in corrupted files), making and keeping Smalltalk images was not feasible. Instead, the ASCII source of the converter was the basis of development. Each session was started with a pure image, into which the converter was loaded. When bugs were found corrections were made both to the code in the image and the code in the source. When BS ran out of space after four to eight hours, or when the system hung, the image was discarded and the process repeated.

[20]It is important for a highly interactive system to handle input fast enough that the user's mind doesn't wander. Greater speed is not necessary. A quarter to half a Dorado is adequate for Smalltalk.

Appendix: Converter Output

This appendix contains an extended conversion example. It is intended to give a sense of the code the converter produces.

The source for the example consists of 13 lines of Smalltalk-80 containing 15 statements. The method is named example:. It uses a receiver variable r, an argument a, and a local temporary l. It sends the messages m, m:, *, +, >, ifTrue:ifFalse, ifProbable:ifPossible:ifUnlikely:, whileTrue:, and to:do:.

```
example: a
        | l |
        l _ a. r _ l. l _ r.
        self m. super m. a m: l.
        self m; m.
        r _ a * l + a * r.
        r _ (a * l) + (a * r).
        r _ a > l.
        (a > l) ifTrue: [ l _ a ] ifFalse: [ ^ r ].
        (a = l) ifProbable: [ ^ 1 ] ifPossible: [ ^ 10 ] ifUnlikely: [ ^ 1000 ].
        [ false ] whileTrue: [ l * 2 ].
        1 to: 10 do: [ :l | l * 2 ].
        ^ self
```

The source code compiles to 120 bytes (including literals). The bytecode listing presented here was produced by the Smalltalk InstructionPrinter. Receiver variable r is Rcvr 0, argument a is Temp 0, and local variable l is Temp 1.

```
17 <10> pushTemp: 0
18 <69> popIntoTemp: 1
19 <11> pushTemp: 1
20 <63> popIntoRcvr: 0
21 <03> pushRcvr: 0
22 <69> popIntoTemp: 1
23 <70> self
24 <D0> send: m
25 <87> pop
26 <70> self
27 <85 00> superSend: m
29 <87> pop
30 <10> pushTemp: 0
31 <11> pushTemp: 1
32 <E1> send: m:
33 <87> pop
34 <70> self
35 <88> dup
36 <D0> send: m
37 <87> pop
38 <D0> send: m
39 <87> pop
40 <10> pushTemp: 0
```

```
41 <11> pushTemp: 1
42 <B8> send: *
43 <10> pushTemp: 0
44 <B0> send: +
45 <03> pushRcvr: 0
46 <B8> send: *
47 <63> popIntoRcvr: 0
48 <10> pushTemp: 0
49 <11> pushTemp: 1
50 <B8> send: *
51 <10> pushTemp: 0
52 <03> pushRcvr: 0
53 <B8> send: *
54 <B0> send: +
55 <63> popIntoRcvr: 0
56 <10> pushTemp: 0
57 <11> pushTemp: 1
58 <B3> send: >
59 <63> popIntoRcvr: 0
60 <10> pushTemp: 0
61 <11> pushTemp: 1
62 <B3> send: >
63 <9B> jumpFalse: 68
64 <10> pushTemp: 0
65 <81 41> storeIntoTemp: 1
67 <91> jumpTo: 70
68 <03> pushRcvr: 0
69 <7C> returnTop
70 <87> pop
71 <10> pushTemp: 0
72 <11> pushTemp: 1
73 <B6> send: =
74 <89> pushThisContext:
75 <75> pushConstant: 0
76 <C8> send: blockCopy:
77 <A4 02> jumpTo: 81
79 <76> pushConstant: 1
80 <7C> returnTop
81 <89> pushThisContext:
82 <75> pushConstant: 0
83 <C8> send: blockCopy:
84 <A4 02> jumpTo: 88
86 <23> pushConstant: 10
87 <7C> returnTop
88 <89> pushThisContext:
89 <75> pushConstant: 0
90 <C8> send: blockCopy:
91 <A4 02> jumpTo: 95
93 <24> pushConstant: 1000
94 <7C> returnTop
95 <83 62> send: ifProbable:ifPossible:ifUnlikely:
97 <87> pop
98 <72> pushConstant: false
99 <9D> jumpFalse: 106
100 <11> pushTemp: 1
101 <77> pushConstant: 2
102 <B8> send: *
103 <87> pop
104 <A3 F8> jumpTo: 98
106 <76> pushConstant: 1
107 <23> pushConstant: 10
```

```
108 <89> pushThisContext:
109 <76> pushConstant: 1
110 <C8> send: blockCopy:
111 <A4 05> jumpTo: 118
113 <69> popIntoTemp: 1
114 <11> pushTemp: 1
115 <77> pushConstant: 2
116 <B8> send: *
117 <7D> blockReturn
118 <F5> send: to:do:
119 <87> pop
120 <78> returnSelf
```

The bytecodes translate into 122 SOAR words. Self, the receiver, is in register 14. Receiver variable r is offset 4 indexed from the receiver (symbolically (r14)4). Argument a is in register 13, and the local variable l is in register 12. The receiver of a send, and the result it returns, are put in register 6, with the arguments of the send put in lower registers, starting with register 5. Two system routines are called, InitLookup and InitSuperLookup, which set up the inline cache, and two system routines are jumped to, methodBlockCopy and BlockMethodReturnTop. Four macros are used, MethodHeader, MethodPrologue, LoadLiteral, and WindowReturn. For and understanding of the runtime system, see [SOAR-Daedalus].

```
        MethodHeader
    BEGIN
        MethodPrologue(SOARTest,example:)
$literals:
        m
        m:
        ifProbable:ifPossible:ifUnlikely:
        10
        1000
        to:do:
        SOARTest
$Byte17:
$Byte18:
-- evaluate
        %add    r13,constant0,r12
$Byte19:
$Byte20:
-- evaluate
-- -- putReceiverVariable 0
        store   r12,(r14)4
$Byte21:
$Byte22:
-- evaluate
-- -- getReceiverVariable 0
        load    (r14)4,r12
$Byte23:
$Byte24:
-- Send m (0)
```

```
-- evaluate
        %add    r14,constant0,r6
        %call   InitLookup
        m
$Byte25:
$Byte26:
$Byte27:
-- Send Super m (0)
-- evaluate
        %add    r14,constant0,r6
        skip    always r0,r0
        SOARTest
        %call   InitSuperLookup
        m
$Byte28:
$Byte29:
$Byte30:
$Byte31:
$Byte32:
-- Send m: (1)
-- evaluate
        %add    r12,constant0,r5
-- evaluate
        %add    r13,constant0,r6
        %call   InitLookup
        m:
$Byte33:
$Byte34:
$Byte35:
$Byte36:
-- evaluate
        %add    r14,constant0,r11
-- Send m (0)
-- evaluate
        %add    r14,constant0,r6
        %call   InitLookup
        m
$Byte37:
$Byte38:
-- Send m (0)
-- evaluate
        %add    r11,constant0,r6
        %call   InitLookup
        m
$Byte39:
$Byte40:
$Byte41:
$Byte42:
-- Send * (1)
-- evaluate
        %add    r12,constant0,r5
-- evaluate
        %add    r13,constant0,r6
        %call   InitLookup
        *
$Byte43:
$Byte44:
$Byte45:
$Byte46:
-- Send * (1)
-- evaluate
```

```
-- -- getReceiverVariable 0
        load    (r14)4,r5
-- evaluate
-- -- add
        add     r6,r13,r6
        %call   InitLookup
        *
$Byte47:
-- evaluate
-- -- putReceiverVariable 0
        store   r6,(r14)4
$Byte48:
$Byte49:
$Byte50:
-- Send * (1)
-- evaluate
        %add    r12,constant0,r5
-- evaluate
        %add    r13,constant0,r6
        %call   InitLookup
        *
$Byte51:
$Byte52:
$Byte53:
-- evaluate
        %add    r6,constant0,r11
-- Send * (1)
-- evaluate
-- -- getReceiverVariable 0
        load    (r14)4,r5
-- evaluate
        %add    r13,constant0,r6
        %call   InitLookup
        *
$Byte54:
$Byte55:
-- evaluate
-- -- add
        add     r11,r6,r0
-- -- putReceiverVariable 0
        store   r0,(r14)4
$Byte56:
$Byte57:
$Byte58:
$Byte59:
-- evaluate
-- -- compare
        %add    rzero,trueOop,r0
        skip    gt r13,r12
        %add    rzero,falseOop,r0
-- -- putReceiverVariable 0
        store   r0,(r14)4
$Byte60:
$Byte61:
$Byte62:
$Byte63:
-- jumpIfFalse 4
        skip    gt r13,r12
        jump    $Byte68
$Byte64:
$Byte65:
```

```
-- evaluate
        %add    r13,constant0,r12
$Byte66:
$Byte67:
-- jump 2
-- evaluate
        %add    r13,constant0,r11
        jump    $Byte70
$Byte68:
$Byte69:
-- methodReturnTop
-- evaluate
-- -- getReceiverVariable 0
        load    (r14)4,r14
        MethodReturnTop2
$Byte70:
$Byte71:
$Byte72:
$Byte73:
$Byte74:
$Byte75:
$Byte76:
$Byte77:
-- blockCopy
-- evaluate
-- -- compare
        %add    rzero,trueOop,r11
        skip    eq r13,r12
        %add    rzero,falseOop,r11
-- BlockCopy
-- evaluate
-- -- getSpecialConstant 5
        %add    rzero,constant0,r5
        %add    pc,constant1,r4
        %jump   methodBlockCopy
        jump    $Byte81
$Byte78:
$Byte79:
$Byte80:
-- methodReturnTop
-- evaluate
-- -- getSpecialConstant 6
        %add    rzero,plus1,r14
        %jump   BlockMethodReturnTop
$Byte81:
$Byte82:
$Byte83:
$Byte84:
-- blockCopy
-- evaluate
        %add    r6,constant0,r10
-- BlockCopy
-- evaluate
-- -- getSpecialConstant 5
        %add    rzero,constant0,r5
        %add    pc,constant1,r4
        %jump   methodBlockCopy
        jump    $Byte88
$Byte85:
$Byte86:
$Byte87:
```

```
-- methodReturnTop
-- evaluate
-- -- getLiteralConstant 4
        %add    rzero,10,r14
        %jump   BlockMethodReturnTop
$Byte88:
$Byte89:
$Byte90:
$Byte91:
-- blockCopy
-- evaluate
        %add    r6,constant0,r9
-- BlockCopy
-- evaluate
-- -- getSpecialConstant 5
        %add    rzero,constant0,r5
        %add    pc,constant1,r4
        %jump   methodBlockCopy
        jump    $Byte95
$Byte92:
$Byte93:
$Byte94:
-- methodReturnTop
-- evaluate
-- -- getLiteralConstant 5
        LoadLiteral(r14,4)
        %jump   BlockMethodReturnTop
$Byte95:
-- Send ifProbable:ifPossible:ifUnlikely: (3)
-- evaluate
        %add    r6,constant0,r3
-- evaluate
        %add    r9,constant0,r4
-- evaluate
        %add    r10,constant0,r5
-- evaluate
        %add    r11,constant0,r6
        %call   InitLookup
        ifProbable:ifPossible:ifUnlikely:
$Byte96:
$Byte97:
$Byte98:
$Byte99:
-- jumpIfFalse 6
-- -- getSpecialConstant 2
        %add    rzero,falseOop,r0
        %skip   ne r0,falseOop
        jump    $Byte106
$Byte100:
$Byte101:
$Byte102:
-- Send * (1)
-- evaluate
-- -- getSpecialConstant 7
        %add    rzero,plus2,r5
-- evaluate
        %add    r12,constant0,r6
        %call   InitLookup
        *
$Byte103:
$Byte104:
```

```
-- jump -8
        jump    $Byte98
$Byte105:
$Byte106:
$Byte107:
$Byte108:
$Byte109:
$Byte110:
$Byte111:
-- blockCopy
-- evaluate
-- -- getSpecialConstant 6
        %add    rzero,plus1,r11
-- evaluate
-- -- getLiteralConstant 4
        %add    rzero,10,r10
-- BlockCopy
-- evaluate
-- -- getSpecialConstant 6
        %add    rzero,plus1,r5
        %add    pc,constant1,r4
        %jump   methodBlockCopy
        jump    $Byte118
$Byte112:
$Byte113:
-- evaluate
-- -- putTemporaryVariable 1
        store   r13,(r14)5
$Byte114:
$Byte115:
$Byte116:
-- Send * (1)
-- evaluate
-- -- getSpecialConstant 7
        %add    rzero,plus2,r5
-- evaluate
-- -- getTemporaryVariable 1
        load    (r14)5,r6
        %call   InitLookup
        *
$Byte117:
-- blockReturnTop
-- evaluate
        %add    r6,constant0,r14
        WindowReturn
$Byte118:
-- Send to:do: (2)
-- evaluate
        %add    r6,constant0,r4
-- evaluate
        %add    r10,constant0,r5
-- evaluate
        %add    r11,constant0,r6
        %call   InitLookup
        to:do:
$Byte119:
$Byte120:
-- methodReturnReceiver
        WindowReturn
-- 7 literals
-- 1 arguments
```

```
-- 2 temporaries
-- 3 stackoids
-- 120 bytes
-- 122 words
```

This is the symbolic output of the converter; all comments were generated automatically. Compare the operator sequence * + * (bytes 40-47) with the sequence * * + (bytes 48-55) -- the latter requires that the result of the first * be saved in a temporary (register 11). Compare the inline ifTrue:ifFalse: (bytes 60-70) with the ifProbable:... send (bytes 71-97), and the inline whileTrue: (bytes 98-104) with the to:do: send (bytes 106-119).

Appendix: Converter Statistics

This appendix contains the statistics obtained from the conversion of the licensed image; 4770 methods were converted in 238 seconds, 1137 containing blocks and 296 requiring 2 passes. The general form of each table entry is <value>:<number of methods with that value>.

This table displays the distribution of the lengths, in bytes, of the bytecode methods. These lengths are exclusive of the object header but include the literal frame.

Bytecode Method Sizes (bytes)						32.4776 mean			
2:305	3:81	4:11	5:127	6:194	7:152	8:220	9:152	10:160	11:193
12:178	13:183	14:121	15:116	16:113	17:109	18:105	19:102	20:66	21:89
22:76	23:58	24:62	25:53	26:70	27:58	28:63	29:42	30:36	31:44
32:28	33:31	34:40	35:51	36:37	37:35	38:26	39:26	40:38	41:30
42:17	43:25	44:26	45:17	46:31	47:31	48:24	49:22	50:18	51:19
52:21	53:18	54:21	55:25	56:16	57:15	58:16	59:14	60:14	61:15
62:16	63:6	64:19	65:13	66:23	67:21	68:13	69:13	70:15	71:13
72:11	73:12	74:10	75:10	76:13	77:9	78:13	79:8	80:6	81:7
82:12	83:7	84:9	85:10	86:8	87:3	88:7	89:8	90:10	91:8
92:5	93:8	94:5	95:6	96:8	97:7	98:6	99:2	100:4	101:6
102:7	103:8	104:5	105:4	106:2	107:6	108:2	109:5	110:6	111:4
112:1	113:1	114:4	115:4	116:6	117:3	118:3	119:4	120:5	121:6
122:4	123:2	124:3	125:2	126:4	127:3	128:4	129:4	130:2	131:2
132:3	133:2	134:5	135:3	136:5	137:4	138:2	139:1	140:2	141:4
142:2	143:1	144:3	145:2	147:3	148:2	149:4	150:1	151:1	152:4
153:1	154:2	155:4	157:4	158:2	159:2	160:3	161:1	162:2	163:2
164:2	165:2	166:1	167:1	168:1	170:1	171:3	175:1	176:1	177:2
178:3	179:2	180:2	181:5	182:1	184:2	189:1	190:2	191:1	192:2
196:1	197:1	198:1	200:2	201:1	202:2	203:1	204:2	205:1	207:1
208:2	210:1	212:1	214:1	216:1	217:2	222:1	223:1	224:1	227:1
229:2	230:1	238:1	239:1	240:2	242:1	245:1	246:2	251:1	253:1
256:2	260:1	261:1	265:1	268:1	270:2	272:1	279:1	280:1	281:2
285:1	289:1	301:1	302:1	309:1	324:2	337:1	349:1	353:1	388:1
402:1	487:1	637:1							

This table displays the distribution of the lengths, in words, of the SOAR methods produced by the converter. These lengths are exclusive of the object header, but include the literal frame and all code, including the method prologue.

SOAR Method Sizes (words)						40.6451 mean			
7:325	8:178	9:59	10:50	11:207	12:114	13:239	14:166	15:133	16:117
17:151	18:144	19:152	20:126	21:118	22:75	23:82	24:92	25:96	26:77
27:83	28:76	29:49	30:55	31:50	32:75	33:35	34:54	35:42	36:40
37:43	38:40	39:34	40:31	41:37	42:14	43:34	44:24	45:27	46:21
47:25	48:39	49:30	50:29	51:24	52:32	53:22	54:19	55:19	56:21
57:19	58:23	59:25	60:20	61:21	62:18	63:11	64:15	65:19	66:19
67:16	68:20	69:14	70:13	71:8	72:8	73:10	74:17	75:16	76:12
77:8	78:13	79:18	80:13	81:10	82:13	83:13	84:12	85:11	86:6
87:6	88:7	89:6	90:2	91:8	92:8	93:6	94:4	95:6	96:9
97:13	98:4	99:5	100:8	101:5	102:13	103:7	104:4	105:3	106:5
107:12	108:3	109:5	110:7	111:3	112:10	113:6	114:5	115:7	116:3
117:5	118:6	119:5	120:5	121:4	122:3	123:2	124:5	125:4	127:8
128:3	129:5	130:4	131:5	132:3	134:4	135:4	136:3	137:3	139:1
140:4	141:4	142:2	143:1	144:3	145:4	146:5	147:3	148:4	149:2
150:2	151:2	152:5	153:1	154:2	155:5	156:1	157:3	158:2	159:1
160:2	161:2	162:3	163:3	164:2	165:1	166:2	167:3	169:1	170:1
171:2	172:1	173:1	175:2	176:3	177:2	178:3	179:1	180:2	182:2
183:2	184:3	185:1	186:2	187:1	189:1	190:1	191:4	192:3	193:1
194:3	195:3	196:2	197:3	198:4	199:1	200:3	201:1	202:2	204:1
205:2	206:2	207:1	208:2	209:1	210:1	211:1	212:1	213:2	214:1
215:1	217:2	218:1	219:1	220:1	222:1	226:1	227:1	228:1	229:1
231:2	232:2	233:2	234:2	236:1	237:1	238:1	241:2	244:1	245:1
246:1	247:1	248:2	250:1	254:1	258:2	261:1	262:1	263:1	265:1
267:1	272:1	277:1	278:1	280:1	281:1	283:1	284:1	286:1	288:1
291:1	294:2	296:1	299:1	305:2	309:1	312:1	316:2	317:1	321:1
322:1	328:1	333:1	335:1	339:1	341:1	342:1	350:1	359:1	364:1
367:2	374:1	376:1	383:1	410:1	447:1	452:1	611:1	684:1	

This table presents the distribution of the number of literals in the literal frame.

Literal Frame Sizes					4.27841 mean				
0:776	1:873	2:729	3:595	4:410	5:244	6:190	7:166	8:131	9:88
10:75	11:76	12:74	13:47	14:33	15:38	16:29	17:21	18:25	19:21
20:14	21:22	22:10	23:9	24:12	25:6	26:9	27:7	28:4	29:3
30:3	31:1	32:2	33:6	34:3	35:1	37:4	38:1	39:2	41:1
42:2	44:1	45:2	50:1	58:1	61:1	62:1			

This table presents the distribution of the number of method arguments.

Argument Counts					0.82914 mean					
0:2293	1:1639	2:495	3:187	4:75	5:51	6:16	7:7	8:5	12:1	13:1

This table presents the distribution of the number of named local method variables (the number of Smalltalk method temporaries minus the number of arguments).

Local Counts					0.675262 mean				
0:3414	1:583	2:351	3:171	4:98	5:58	6:32	7:25	8:12	9:10
10:9	11:2	12:2	15:1	18:1	19:1				

This table presents the distribution of the number of Smalltalk method temporaries.

Method Temporaries					1.5044 mean				
0:1842	1:1306	2:636	3:393	4:231	5:134	6:70	7:57	8:37	9:22
10:18	11:7	12:3	13:3	14:5	15:1	17:2	18:1	21:1	22:1

This table presents the distribution of the number of unnamed converter temporaries.

Converter Temporary Counts						0.821174 mean				
0:2523	1:1240	2:655	3:206	4:58	5:50	6:16	7:11	8:7	9:2	10:2

This table presents the distribution of total register requirements (Smalltalk method temporaries plus converter temporaries).

Register Requirements						2.32558 mean			
0:1402	1:1040	2:665	3:496	4:360	5:258	6:189	7:99	8:83	9:51
10:39	11:31	12:21	13:11	14:7	15:7	16:3	17:1	18:1	19:1
22:1	23:1	25:1	26:1	28:1					

This table presents the distribution of spill area sizes.

Spill Sizes					0.449497 mean				
0:4324	1:54	2:43	3:36	4:73	5:103	6:53	7:30	8:21	9:8
10:10	11:3	12:2	13:1	14:4	16:1	19:2	20:1	22:1	

References

[BlueBook]

Smalltalk-80 -- The Language and its Implementation; Adele Goldberg and David Robson; Addison-Wesley; 1983

[Dorado]

'The Dorado: A High-Performance Personal Computer -- Three Papers'; *Xerox CSL-81-1*; January 1981

[Interfaces]

'The Effect of Modularization and Comments on Program Comprehension'; S.N Woodfield, H.E. Dunsmore, and V.Y.Shen; *Proceedings of the Fifth International Conference on Software Engineering*; March 1981; pp 215-223

[OrangeBook]

Smalltalk-80 -- The Interactive Programming Environment; Adele Goldberg; Addison-Wesley; 1984

[RISCs]

'Reduced Instruction Set Computers'; David A. Patterson; *Communications of the ACM*; January 1985; pp 8-21

[SOAR-Arch]

SOAR Architecture; A. Dain Samples, Mike Klein, and Pete Foley; 24 September 1981

[SOAR-Daedalus]

Daedalus: Software for SOARing on a Sun; A. Dain Samples; 28 February 1985

[SOAR-292R-Blakken]

'Register Windows for SOAR'; John Blakken; *Smalltalk On A RISC -- Proceedings of CS929R*; April 1983; pp 126-140

[SOAR-292R-Citrin]

'Implementing a Smalltalk Compiler'; Wayne Citrin and Carl Ponder; *Smalltalk On A RISC -- Proceedings of CS929R*; April 1983; pp 167-185

[Sopaipilla]

'Sopaipilla Maintenance Manual'; David K. Wall and Erik T. Gilbert; *Stanford University CSL Technical Note 158*; March 1978

[VLSI-RISC]

'A VLSI RISC'; David A. Patterson and Carlo H. Sequin; *Computer*; September 1982; pp 8-21