bytecode interpreters for tiny computers 10/28/08 8:17 AM

http://lists.canonical.org/pipermail/kragen-tol/2007-September/000871.html Page 1 of 25

**bytecode interpreters for tiny computers**

**Kragen Javier Sitaker** kragen at pobox.com

*Fri Sep 28 04:30:05 EDT 2007*

Previous message: non-chalantly paper napkin

**Messages sorted by:** [ date ] [ thread ] [ subject ] [ author ]

(I originally tried posting this in March, but it's 61K so it was

rejected --- sorry about the size.)

Contents

--------

Introduction: Density Is King (With a Tiny VM)

Indirect Threaded 16-bit FORTH: Not Great For Density

Naive Lisp: Terrible For Density

Lua's VM: Four-Byte Register-Based Instructions

The MuP21 and F21 Instruction Sets

Local Variables: Registers or Stacks?

Adapting the MuP21 Instruction Set to a Smalltalk-Like System

Speculative Case Study: An F21-like Squeak

Steve Wozniak's SWEET 16 Dream Machine

NanoVM: Java Bytecodes on the AVR

Code-Compact Data Structures

Library Design

Other Existing Small Interpreters

\* S21

\* Squeak?

\* pepsi/Albert/the golden box

\* The Basic STAMP

\* UCSD P-System

\* PICBIT

\* F-83

\* Various combinator-graph-reduction machines

Conclusions: Squeak Rules, We Should Be Able To Do Python In A Few K

Introduction: Density Is King (With a Tiny VM)

----------------------------------------------

I've previously come to the conclusion that there's little reason for

using bytecode in the modern world, except in order to get more

compact code, for which it can be very effective. So, what kind of a

bytecode engine will give you more compact code?

Suppose I want a bytecode interpreter for a very small programming

environment, specifically to minimize the memory needed for a program;

say, on a 32-bit microcontroller with 40KiB of program flash, where

the program flash size is very often the limiting factor on what the

machine can do.

My dumb fib microbenchmark looks like this in Smalltalk:

fib: n

n < 2 ifTrue: [^1]

ifFalse: [^(self fib: n - 1) + (self fib: n - 2)]

And in Squeak bytecode:

9 <10> pushTemp: 0

10 <77> pushConstant: 2

11 <B2> send: <

12 <99> jumpFalse: 15

13 <76> pushConstant: 1

14 <7C> returnTop

15 <70> self

16 <10> pushTemp: 0

17 <76> pushConstant: 1

18 <B1> send: -

19 <E0> send: fib:

20 <70> self

21 <10> pushTemp: 0

22 <77> pushConstant: 2

23 <B1> send: -

24 <E0> send: fib:

25 <B0> send: +

26 <7C> returnTop

Or, as I translated to pseudo-FORTH, "n 2 < if 1 return then self n 1

- recurse self n 2 - recurse + return".

The metric of goodness for a CPU instruction set is a little different

from that for a bytecode interpreter. Bytecode interpreters don't

have to worry about clock rate (and therefore combinational logic path

length) or, so far, parallelism; they can use arbitrary amounts of

storage on their own behalf; they're easier to modify; their

fundamental operations can take advantage of more indirection.

Here are some examples of things a bytecode interpreter can do that a

hardware CPU might have more trouble with:

- you can have a very large register set (which is more or less what

Squeak's VM does, treating local variables as registers) without

incurring slow procedure call and return; MMIX suggests how this

could be done in hardware as well.

- you could imagine that every procedure could have its own register

set (as, perhaps, on the SPARC), and a few of the instructions could

access the contents of these registers; again, Squeak's VM does this

- you could have an instruction to create a new preemptively-scheduled

thread, perhaps switching between threads every instruction, as in

Core Wars or the Tera MTA;

- if the language is object-oriented, you could have a few

instructions for calling certain distinguished methods of self, or

the first argument, as in the Squeak VM;

- or, as a more general form of the same thing, entering some context

might reprogram certain instructions to do some arbitrary thing;

- you can do all kinds of tag tests and dynamic dispatch on

fundamental CPU operations, as in the Squeak VM, the LispMs, or

Python's bytecode;

- you can support associative array lookups, appending to

unbounded-size arrays, and the like, as fundamental machine

operations.

Indirect Threaded 16-bit FORTH: Not Great For Density

-----------------------------------------------------

I don't have a FORTH handy, but I think the definition looks something

like this in FORTH:

: FIB DUP 2 < IF DROP 1

ELSE DUP 1- RECURSE SWAP 2 - RECURSE + THEN ;

which I think, in an indirect-threaded FORTH, compiles to a dictionary

entry containing something like this:

DUP (2) < (IF) #3 DROP (1) (ELSE) #8 DUP 1- FIB SWAP (2) - FIB + ;

That's 18 threading slots, so 36 bytes, plus the overhead of the

dictionary structure, which I think is typically 2 bytes for a

dictionary that has forgotten the word names. Better than PowerPC

assembly (at 96 bytes) but not great, noticeably worse than Squeak.

Naive Lisp: Terrible for Density

--------------------------------

What if we interpret fib with a simple Lisp interpreter that walks

tree structures? We could define it as follows:

(labels fib (n) (if (< 2 n) 1 (+ (fib (- n 1)) (fib (- n 2)))))

That's 17 non-parenthesis tokens and 9 right parentheses, for a total

of 28 leaf-nodes on the cons tree. That means the tree contains 27

conses, for 54 memory-address-containing cells in interior nodes,

probably a minimum of 108 bytes. I conclude that while this

program-representation approach is very simple, it takes up a lot of

space. I don't think cdr-coding would help enough, since none of the

lists are very long; if you had 9 lists containing 25 pointers and 9

one-byte lengths or one-byte terminators, you still have 59 bytes.

Lua's VM: Four-Byte Register-Based Instructions

-----------------------------------------------

According to "The Implementation of Lua 5.0", Lua's virtual machine

has been register-based since 2003. They claim that their four-byte

register instructions aren't really much more voluminous than

stack-based instructions, perhaps in part because they're comparing to

stack-based instructions for a single-stack machine that has local

variable storage in addition to its stack.

Lua's register-based virtual machine is fairly small: "[O]n Linux its

stand-alone interpreter, complete with all standard libraries, takes

less than 150 Kbytes; the core is less than 100 Kbytes." They've

previously said that the compiler is about 30% of the size of the

core, which suggests that the rest of the core, including the bytecode

interpreter, is about 70KB.

They mention that it has 35 instructions, which would almost fit in 5

bits of opcode: MOVE, LOADK, LOADBOOL (converts to boolean and

conditionally skips an instruction), LOADNIL (clears a bunch of

registers), GETUPVAL, GETGLOBAL, GETTABLE, GETGLOBAL, SETUPVAL,

SETTABLE, NEWTABLE, SELF, ADD, SUB, MUL, DIV, POW, UNM (unary minus),

NOT, CONCAT (string concatenation of a bunch of registers), JMP, EQ,

LT, LE, TEST, CALL, TAILCALL, RETURN, FORLOOP, TFORLOOP, TFORPREP,

SETLIST, SETLISTO, CLOSE, and CLOSURE.

CALL passes a range of registers to a function and stores its result

in a range of registers; this implies that the virtual machine handles

saving and restoring of the stack frame. The paper uses the term

"register window" to compare it to what the SPARC does.

The comparison instructions skip the next instruction on success.

Here's their example code to show how much better the register machine

is:

local a, t, i LOADNIL 0 2 0

a = a + i ADD 0 0 2

a = a + 1 ADD 0 0 250

a = t[i] GETTABLE 0 1 2

The old stack machine compiled this as follows:

PUSHNIL 3

GETLOCAL 0

GETLOCAL 2

ADD

SETLOCAL 0

GETLOCAL 0

ADDI 1

SETLOCAL 0

GETINDEXED 2

SETLOCAL 0

It seems that you should be able to compile this on a two-stack

machine as NIL NIL DUP >R + 1+ R> NIL GETTABLE, which is 9

instructions instead of 11, and also clearly stupid, since nil is

neither a table nor a number. If you could really fit that into 6

bytes, it might be an improvement over the 12 bytes of their current

scheme or the 11 bytes of their previous one. It might be better to

try more realistic code fragments.

The paper also discusses an interesting implementation of closures, in

which captured variables migrate into heap-allocated structures upon

function return.

The MuP21 and F21 instruction sets

----------------------------------

The MuP21 was implemented in 6000 transistors, including an NTSC

signal generator and a controller for external DRAM, so it ought to be

possible to emulate its behavior with a fairly small amount of

software. Here's the instruction set:

Transfer Instructions: JUMP, CALL, RET, JZ, JCZ

Memory Instructions: LOAD, STORE, LOADP, STOREP, LIT

ALU Instructions: COM, XOR, AND, ADD, SHL, SHR, ADDNZ

Register Instructions: LOADA, STOREA, DUP, DROP, OVER, NOP

COM is complement. The CPU has an A register, accessed with LOADA and

STOREA, that supplies the address for LOAD and STORE; I think LOADP

and STOREP increment it as well. I think JCZ jumps if the carry bit

is zero. (Each register on the stack has its own carry bit; the "21"

refers to the 20-bit memory word size, plus the extra bit.)

The F21 had 27 instructions to the MuP21's 24. (Only 23 are listed

above, hmm.) They were renamed:

Code Name Description Forth (with a variable named A)

00 else unconditional jump ELSE

01 T0 jump if T0-19 is false w/ no drop DUP IF

02 call push PC+1 to R, jump :

03 C0 jump if T20 is false CARRY? IF

06 RET pop PC from R (subroutine return) ;

08 @R+ fetch from address in R, increment R R@ @ R> 1+ >R

09 @A+ fetch from address in A, increment A A @ @ 1 A +!

0A # fetch from PC+1, increment PC LIT

0B @A fetch from address in A A @ @

0C !R+ store to address in R, increment R R@ ! R> 1+ >R

0D !A+ store to address in A, increment A A @ ! 1 A +!

0F !A store to address in A A @ !

10 com complement T -1 XOR

11 2\* left shift T, 0 to T0 2\*

12 2/ right shift T, T20 to T19 2/

13 +\* add S to T if T0 is true DUP 1 AND IF OVER + THEN

14 -or exclusive-or S to T XOR

15 and and S to T AND

17 + add S to T +

18 pop pop R, push to T R>

19 A push A to T A @

1A dup push T to T DUP

1B over push S to T OVER

1C push pop T, push to R >R

1D A! pop T to A A !

1E nop delay 2ns NOP

1F drop pop T DROP

T is top-of-stack; R is top-of-return-stack; S is the element right

under the top of stack. I think @R+ and !R+ are two of the three new

instructions; push and pop are probably the other one, since they

don't seem to be listed in the MuP21 list.

I'm not sure what +\* is for, but I'm guessing it was ADDNZ.

I'm not sure where the else, T0, and C0 instructions jump to; maybe

the next address on the operand stack.

Interestingly, there doesn't seem to be a straightforward way to get a

"1" onto the stack without using the # instruction, which is annoying

because that takes 25 bits of instructions. dup dup -or A! @A+ drop A

is another approach at 30 bits, but it clobbers the A register and

issues a useless memory reference. dup dup -or com 2\* com is another

25-bit approach.

So here's my dumb fib benchmark expressed in F21 code, according to my

limited understanding, and without trying to be very clever:

fib: dup #-2 + #returnone swap c0 dup #-1 + #fib call

swap #-2 + #fib call + ;

returnone: drop drop #1 ;

That loses pretty badly on literals; if we assume that # pushes its

value immediately and doesn't require any NOPs (e.g. to avoid having

multiple # instructions per word) then we have 22 instructions and 7

literals --- 6 words of instructions and 7 of literals, for a total of

32.5 bytes. Not the code density direction I was hoping this would

take me!

But it's possible to avoid the redundant literals:

fib: dup #-2 dup push + #returnone swap c0 dup #-1 + #fib dup push call

swap pop swap pop + swap call + ;

returnone: pop drop drop drop #1 ;

And actually #1 is somewhat redundant with #-2, being its bitwise

complement:

fib: dup #-2 dup push + #returnone swap c0 dup #-1 + #fib dup push call

swap pop swap pop + swap call + ;

returnone: drop drop pop com ;

That makes it 29 instructions but only 4 literals --- 8 words of

instructions, 4 of literals, for a total of 12 20-bit words, or 30

bytes. Still worse than the Squeak version on size --- and quite hard

to read! And some of the literals are still probably too close

together to work on a real machine.

If we were instead using three-instruction 16-bit words with a high

bit used to tag literals, we could maybe win a little more.

fib: #-2 dup push over + nop nop #returnone swap c0

dup #-1 + nop nop #fib dup push call

swap pop swap pop + swap call + ;

returnone: drop drop pop com ;

That's 33 instructions, but the four literals don't count, so 29

instructions or 10 16-bit instruction words, plus four 16-bit literal

words. That's 28 bytes, almost the same as the Squeak version, but

still worse! And that's with me trying to get clever with the

instruction reordering, too.

Now I begin to understand why Chuck Moore was getting to the point

where he would repeat FOO twenty times by doing : FOO5 FOO FOO FOO FOO

FOO ; FOO5 FOO5 FOO5 FOO5 instead of using a DO loop. Numbers are a

real pain on the F21! (But perhaps that's as it should be;

programming isn't about numbers, anyway.)

Local Variables: Registers Or Stacks?

-------------------------------------

Having two stacks removes the need for local argument vectors; you can

always shift the variables left and right between the call and return

stack, possibly swapping as you go, to get to the values you want.

(This could be shortened if there was a "repeat next instruction four

times" instruction: >R, >R >R, 4x >R R>, 4x >R, 4x >R >R, 4x >R 4x >R

R> R>, 4x >R 4x >R R>, 4x >R 4x >R, and so on; and similar in the

other direction.) It wasn't apparent to me which approach would use

less code, or whether it would depend on the number of arguments and

local variables.

I thought I'd see what the distribution is like in a body of real

code, so I ran the following code in Squeak 3.8-6665. (No doubt any

Smalltalk programmer could improve it.)

gatherMethodStats

"How common are methods with lots of temps?"

| totaldict tempdict argsdict update |

tempdict := Dictionary new. "Maybe not the best container."

argsdict := Dictionary new.

totaldict := Dictionary new.

update := [:dict :key | dict at: key put: (1 +

(dict at: key ifAbsent: [0]))].

Smalltalk allClassesDo: [:class |

(Array with: class with: class class) do: [:cl |

cl selectorsAndMethodsDo: [:sel :meth |

update value: tempdict value: meth numTemps.

update value: argsdict value: meth numArgs.

update value: totaldict

value: meth numTemps + meth numArgs.

]

]

].

^ {'temps' -> tempdict. 'args' -> argsdict. 'total' -> totaldict.}

In a fraction of a second, this returned the following (reformatted):

#('temps' -> a Dictionary(

0->18952 1->13665 2->6366 3->3697 4->2301

5->1492 6->939 7->676 8->426 9->346

10->196 11->193 12->139 13->99 14->60

15->47 16->46 17->30 18->15 19->20

20->12 21->11 22->15 23->6 24->3

25->5 26->4 27->3 28->5 32->1

33->2 37->1 39->1 50->1)

'args' -> a Dictionary(

0->26114 1->15903 2->4717 3->1712 4->756

5->309 6->138 7->64 8->37 9->13

10->8 11->2 12->1 13->1 )

'total' -> a Dictionary(

0->18952 1->3240 2->11976 3->3128 4->4290

5->1760 6->1947 7->999 8->983 9->558

10->521 11->293 12->276 13->155 14->196

15->115 16->81 17->57 18->52 19->31

20->43 21->24 22->20 23->11 24->17

25->9 26->6 27->9 28->7 29->5

30->6 33->1 35->1 36->1 38->1

42->1 44->1 46->1 62->1 )

)

That's out of 49775 methods; so roughly 95% of these methods have 8 or

fewer arguments and temporaries, 90% have 6 or fewer, 75% have 3 or

fewer, and 69% have 2 or fewer. That suggests that in a codebase like

Smalltalk, it would probably be a marginal cost to use two stacks in

the bytecode instead of a local-argument vector.

But probably the methods that have a lot of local variables and

arguments are longer, so inefficiency in implementing those methods

might cause inefficiency out of proportion to their number. How much

does that skew the results? The CompiledMethod class has initialPC

and endPC methods which return the bounds of its bytecode, so I

changed the code to count bytecodes rather than methods:

gatherMethodStats

"How common are methods with lots of temps?"

| totaldict tempdict argsdict update |

tempdict := Dictionary new.

argsdict := Dictionary new.

totaldict := Dictionary new. "Maybe not the best container."

update := [:dict :key :incr |

dict at: key put: (incr + (dict at: key ifAbsent: [0]))].

Smalltalk allClassesDo: [:class |

(Array with: class with: class class) do:

[:cl | cl selectorsAndMethodsDo: [:sel :meth || methbytes |

methbytes := meth endPC - meth initialPC + 1.

update value: tempdict value: meth numTemps value: methbytes.

update value: argsdict value: meth numArgs value: methbytes.

update value: totaldict value: meth numTemps + meth numArgs

value: methbytes.

]

]

].

^ {'temps' -> tempdict. 'args' -> argsdict. 'total' -> totaldict.}

This counted 1 334 542 bytecodes:

'total' -> a Dictionary(

0->171811 1->95663 2->182923 3->117718 4->125591

5->92320 6->92671 7->72908 8->61526 9->49807

10->46568 11->36943 12->27096 13->21759 14->27821

15->17379 16->13309 17->10390 18->9528 19->7659

20->10162 21->5969 22->5238 23->3087 24->5804

25->5229 26->2915 27->3747 28->2551 29->2217

30->3014 33->12 35->405 36->505 38->341

42->357 44->235 46->336 62->1028 )

50% of them are defined in a context with 4 or fewer locals and args;

60% with 6 or fewer; 70% with 7 or fewer; 80% with 10 or fewer; 90%

with 14 or fewer; 95% with 27 or fewer. That's not quite as

encouraging as the raw method counts, but it still suggests that the

approach is viable and probably does not need the "4x" instruction I

suggested earlier. (Even in a method with 14 local variables, all of

which are simultaneously live, with really random access, I think the

average distance from the variable you're currently at to the variable

you want is only a third of 14, or 4.7.)

Adapting the MuP21 Instruction Set to a Smalltalk-Like System

-------------------------------------------------------------

Maybe I could follow the MuP21's lead and use five-bit zero-operand

instructions for a two-stack abstract machine. Probably I should pack

them five to a 32-bit word, or three to a 16-bit word; the left-over

bits can be used for tagging immediate data in the instruction stream,

as in Leong, Tsang, and Lee's MSL16 FPGA-based FORTH CPU.

The appeal of the 5-bit instructions is that, say, my sample fib

program could perhaps be expressed in less than 26 bytes, or 13 16-bit

words: 39 instructions or 16-bit literals. Can we do that? Clearly

it depends on the instruction set. An ideal FORTHish instruction set

for the sample dumb fibonacci program would make it simply

dup 2 return-1-if-less-than dup 1- recurse swap 2- recurse + ;

which is 11 instructions in length, 8 bytes, with 9 distinct

instructions. Some of these instructions --- dup, swap, +, and ; ---

would clearly be included in any FORTH-like CPU; others --- 1-,

return-1-if-less-than, 2, 2-, and recurse --- are less likely. Here's

a version with a more likely instruction set:

dup 1 swap 2 - negative? conditional-return pop

dup 1- literal(fib) call swap 1- 1- literal(fib) call + ;

call, literal, and pop are also almost certain to exist; this version

uses additionally only 1, 2, -, negative?, conditional-return, and 1-.

It contains 17 non-literal instructions and two literals, so it would

be 16 bytes if literals were two bytes.

For this function, we don't really need 2 or - as instructions; "2 -"

can be rewritten just as easily as "1- 1-". That brings the required

instruction repertoire down to 9 regular instructions, plus literal.

The only dubious instruction in the remaining repertoire is negative?,

and it's only dubious because the MuP21 doesn't know about negativity.

I think it amounts to testing the carry bit, which is actually

probably a pretty reasonable thing to either have an operation to test

or to have conditional-return test.

Following the MuP21/F21 model, maybe we could improve on Squeak's

bytecode by avoiding the use of a special space and special

instructions for local variables, by avoiding the need for message

argument counts (and by supporting multiple return values), and

probably by putting references to message selectors inline in the

bytecode rather than in a separate literal table. My instance of

Squeak currently only has 30474 different message selectors, so 16

bits for the selector identifier would probably accommodate many more

years of evolution.

These erasures would not be at the cost of safety --- in Smalltalk,

the argument signature of a method is implicit in the selector, and as

long as the bytecode compiler was bug-free, whatever bogus method got

called would pop the right number of arguments and push a single

return value.

Probably the stack manipulation instructions (# dup over push pop nop

drop) and the control-flow instructions (call else T0 C0 ret) would

stay the same, with the addition of a "send" instruction; it would

probably also be good to keep the A register around in some form, as

the destination for messages, which implies keeping the A and A!

instructions as well, for a total of 14 fixed instructions. The

"send" instruction could simply leave the object reference in A during

the call, and expect it to be preserved --- an "A push" sequence

before clobbering it and a "pop A!" sequence before returning is

probably not too much to ask.

Smalltalk's blocks might have a little difficulty in this environment

--- to access method-local variables, to answer from their containing

method, and to call methods on self; none of these are difficulties in

the cases where the compiler inlines the control structure, of course.

It is, of course, possible to make them into full-fledged objects, as

the abstract semantic models of Smalltalk and Scheme do.

As an alternative to the complete omission of a literal table, a

literal table could override on a per-element basis the elements of a

default literal table that defined the meanings of most of the

instructions; the most common instruction meanings would be at one end

of the table, while the literals would override meanings (with

messages) starting from the other end. Probably messages to call and

constants to push have roughly equal frequencies, in which case we

could use the low bit of the instruction to distinguish them. If the

stack-manipulation and control-flow instructions are non-overridable

at 11 instructions, that gives us a maximum literal-table size of 18

redefinable instructions, which would default to a statically-determined

set of the most common constants and messages in the system.

Literals that overflowed the literal table could still be used inline

with the # instruction, possibly followed by call or send. If, as

previously suggested, the # instruction were merely a high bit in a

16-bit word, the disadvantage relative to a literal-table entry would

be fairly small --- more a 15-bit size limitation than anything else.

If we trust our byte-compiler (or some Java-like type-inferencing

stack-effect verifier), we can probably do type erasure and avoid the

overhead of tag bits here, at least for message selectors.

Larger constants can be constructed fairly easily with a sequence of

multiple literal words and some combining operation such as

(\x y -> x \* 8192 + y).

I said the control-flow operators should stay the same, but probably

the T0 and C0 conditional branches aren't quite the right thing for

Smalltalk; maybe ifFalse and ifNotNil instead.

Speculative Case Study: An F21-like Squeak

------------------------------------------

It might be worthwhile to profile the selector and constant usage of,

say, Squeak, to see what the 18 default literals would be, how many

literals are used more than once in a method (and therefore might

benefit from being put into a literal table). From that perhaps I

could estimate the literal table size of each method in the new

regime, and then I could hand-translate a few methods to see if they

were smaller.

For now I am going to look at a sort of worst-case: suppose we only

had dup, over, push, pop, nop, drop, and literals for stack

manipulation; send, jump/else, ifTrue, ifNotNil, ifFalse, and ret for

control flow; A and A! for changing the destination of messages; and

everything else were done by message sends getting their messages from

inline literals (rather than a literal table), with the literals

distinguished from instructions by a high bit in a 16-bit word. From

results with this worst-case, we can estimate how much better some

piece of code would be if all its selectors and constants had

bytecodes assigned to them.

That gives us 14 opcodes, so we can stuff them four to a 16-bit word

normally, let the high bit be 0 for literals, and make sure that "nop"

has its high bit be zero so we can insert it in the instruction stream

where necessary.

Furthermore, let's assume that we have to handle all blocks, other

than those for ifTrue, ifNotNil, and simple loops, by lambda-lifting.

Lambda-lifting means that we turn each block into, effectively, an

object class; when we instantiate that class, we send it the variables

over which the block is closed, and it stores them in instance

variables.

If the block modifies the variables, we will have to extract their

current values from the block thenceforth, since without further

control-flow analysis there's no way to tell when the block might be

invoked by some apparently unrelated message send.

Here's a method chosen at pseudorandom,

CompiledMethod>>copyWithTrailerBytes: bytes.

| copy end start |

start := self initialPC.

end := self endPC.

copy := CompiledMethod newMethod: end - start + 1 + bytes size

header: self header.

1 to: self numLiterals do: [:i |

copy literalAt: i put: (self literalAt: i)].

start to: end do: [:i | copy at: i put: (self at: i)].

1 to: bytes size do: [:i | copy at: end + i put: (bytes at: i)].

^ copy

In Squeak's bytecode, with suggested bytecode translations

interspersed, and a display of the two stacks separated by a : after a

\.

37 <70> self

38 <D0> send: initialPC

#initialPC send \ bytes start : retaddr

39 <6B> popIntoTemp: 3

40 <70> self

41 <D1> send: endPC

#endPC send \ bytes start end : retaddr

42 <6A> popIntoTemp: 2

43 <43> pushLit: CompiledMethod

#CompiledMethod push \ bytes start end : CompiledMethod retaddr

44 <12> pushTemp: 2

45 <13> pushTemp: 3

46 <B1> send: -

A push \ bytes start end : self CompiledMethod retaddr

A! \ bytes start : self CompiledMethod retaddr

dup push \ bytes start : start self CompiledMethod retaddr

#- send \ bytes end-start : start self CompiledMethod retaddr

47 <76> pushConstant: 1

48 <B0> send: +

A push \ bytes end-start : end start self CompiledMethod retaddr

A! #1 #+ send \ bytes end-start+1 : end start self CompiledMethod retaddr

49 <10> pushTemp: 0

50 <C2> send: size

over A!

#size send \ bytes end-start+1 bytessize : end start self CompiledMethod ret..

51 <B0> send: +

push A! pop #+ send \ A=bytes; end-start+1+bytessize : end start self Com...

52 <70> self

pop pop A pop A! \ A=self; es1b end start bytes : CompiledMethod retaddr

53 <D4> send: header

#header send \ A=self; es1b end start bytes selfheader : CompiledMethod ret...

54 <F2> send: newMethod:header:

55 <69> popIntoTemp: 1

A push push push push push A! \ A=es1b; : end start bytes selfheader self C..

pop pop pop A \ end start bytes es1b : selfheader self CompiledMethod retaddr

pop pop pop A! \ A=CompiledMethod; end start bytes es1b selfheader self : ret..

push #newMethod:header: send \ end start bytes copy : self retaddr

56 <70> self

57 <D5> send: numLiterals

pop A! #numLiterals send \ A=self; end start bytes copy numlits : retaddr

So far, we're at 46 pseudo-FORTH operations and 11 literals (10

distinct), or about 46 bytes of this "worst-case" code, nearly half of

which is literals. That compares poorly to Squeak's 21 bytes up to

this point; even if all the literals in our pseudo-FORTH were

instructions, Squeak's bytecodes would still be slightly smaller up to

this point! (Not counting the Squeak method's 32-byte literal table,

most of which is for the part of the method we haven't gotten to yet.)

Squeak doesn't seem to be getting a big space advantage from its

literals table, since none of the literals have been repeated yet

(except for #+, which would probably be an opcode in either case).

If I could evaluate subexpressions of a method call in an arbitrary

order, the above might be smaller (I could avoid "push push push push

push"), but I wouldn't count on it.

This method is at the median of about four parameters and named

temporaries, but it also has to deal with unnamed temporaries.

Now we're about to start a loop, from 1 to numlits. The last method

send in the loop is to "copy", so we're going to arrange to have it in

the A register when we enter the loop as well.

58 <6D> popIntoTemp: 5

59 <76> pushConstant: 1

60 <6C> popIntoTemp: 4

\ A=self; end start bytes copy numlits : retaddr

push #1 \ A=self; end start bytes copy : 1 numlits retaddr

A push A! \ A=copy; end start bytes : self 1 numlits retaddr

pop \ A=copy; end start bytes self : 1 numlits retaddr

61 <14> pushTemp: 4 ; loop starts here

\ A=copy; end start bytes self : i numlits retaddr

62 <15> pushTemp: 5

63 <B4> send: <=

A pop dup A! \ A=i; end start bytes self copy i : numlits retaddr

pop dup #<= send \ A=i; end start bytes self copy i numlits stillgoing : retaddr

64 <AC 0D> jumpFalse: 79

#79 ifTrue \ A=i; end start bytes self copy i numlits : retaddr

66 <11> pushTemp: 1

67 <14> pushTemp: 4

68 <70> self

69 <14> pushTemp: 4

70 <E7> send: literalAt:

push push push A! \ A=self; end start bytes : copy i numlits retaddr

pop pop dup \ A=self; end start bytes copy i i : numlits retaddr

#literalAt: send \ A=self; end start bytes copy i selfati : numlits retaddr

71 <F6> send: literalAt:put:

A push push push A! \ A=copy; end start bytes : i selfati self numlits retaddr

pop dup pop #literalAt:put: send \ A = copy; end start bytes i trash : self nu..

72 <87> pop

drop

73 <14> pushTemp: 4

74 <76> pushConstant: 1

75 <B0> send: +

76 <6C> popIntoTemp: 4

#1 #+ send \ A=copy; end start bytes i+1 : self numlits retaddr

Now we have to get the stack back to the state for starting the loop,

which turns out to be more work than I'd like:

A push A! \ A=i+1; end start bytes copy : self numlits retaddr

pop A \ end start bytes copy self i+1 : numlits retaddr

push push A! \ A=copy; end start bytes : self i+1 numlits retaddr

pop \ A=copy; end start bytes self : i+1 numlits retaddr

77 <A3 EE> jumpTo: 61

#61 else

So here we are at the end of the first loop. 44 more ordinary

instructions (22 bytes), plus 8 literals. Squeak, by contrast, did

the whole loop in just 21 bytes. Again, even if all the literals went

away, Squeak's bytecode design would still be tighter.

It might be possible to do a better job of arranging things on the

stack so that the computation feels less like programming a Turing

machine --- run over here to fetch that, run back there to put it down

--- and it seems like there's probably an initial state for the loop

that doesn't require 9 instructions to re-establish it at the end.

Still, if there were any code where we'd expect the two-stack machine

to shine, it would be stuff like this --- where we only have three

variables (and a loop limit) accessed inside the loop.

I also made a bunch of mistakes, but I don't think they undermine my

basic conclusion: the two-stack machine design is not

density-competitive with a design with a local-variable vector.

Here's the rest of the Squeak bytecode, which I haven't bothered to

translate:

79 <13> pushTemp: 3

80 <6C> popIntoTemp: 4

81 <14> pushTemp: 4

82 <12> pushTemp: 2

83 <B4> send: <=

84 <AC 0D> jumpFalse: 99

86 <11> pushTemp: 1

87 <14> pushTemp: 4

88 <70> self

89 <14> pushTemp: 4

90 <C0> send: at:

91 <C1> send: at:put:

92 <87> pop

93 <14> pushTemp: 4

94 <76> pushConstant: 1

95 <B0> send: +

96 <6C> popIntoTemp: 4

97 <A3 EE> jumpTo: 81

99 <10> pushTemp: 0

100 <C2> send: size

101 <6D> popIntoTemp: 5

102 <76> pushConstant: 1

103 <6C> popIntoTemp: 4

104 <14> pushTemp: 4

105 <15> pushTemp: 5

106 <B4> send: <=

107 <AC 0F> jumpFalse: 124

109 <11> pushTemp: 1

110 <12> pushTemp: 2

111 <14> pushTemp: 4

112 <B0> send: +

113 <10> pushTemp: 0

114 <14> pushTemp: 4

115 <C0> send: at:

116 <C1> send: at:put:

117 <87> pop

118 <14> pushTemp: 4

119 <76> pushConstant: 1

120 <B0> send: +

121 <6C> popIntoTemp: 4

122 <A3 EC> jumpTo: 104

124 <11> pushTemp: 1

125 <7C> returnTop

You could make the argument that the abstract machine Smalltalk

presents to the user is more like a register machine than a stack

machine, and that this may account for the code being awkward when you

translate it to a stack machine. If I were programming this

originally in a Forth dialect, I probably would structure the code a

little differently, but I doubt it would make that much of an

improvement in the code size, unless we used some kind of auxiliary

non-stack storage for local variables --- at which point we're pretty

much back to Squeak bytecode.

Steve Wozniak's SWEET 16 Dream Machine

--------------------------------------

Steve Wozniak's SWEET16 16-bit virtual machine, included as part of

Integer BASIC, supposedly doubled the code density of the 6502. The

virtual machine itself was 300 bytes of 6502 assembly, implementing

these instructions; here "#" means "[0-F]".

0x1# SET: load immediate 0x2# LD: copy register to accumulator

0x3# ST: copy accumulator to register 0x4# LD: load byte indirect w/ increment

0x5# ST: store byte indirect w/incr 0x6# LDD: load two bytes ind w/incr

0x7# STD: store two bytes ind w/incr 0x8# POP: load byte indirect w/predecr

0x9# STP: store byte ind w/predecr 0xA# ADD: add register to accum

0xB# SUB: subtract register from acc 0xC# POPD: load 2 bytes ind w/predecr

0xD# CPR: compare register w/acc 0xE# INR: increment register

0xF# DCR: decrement register 0x00 RTN to 6502 mode

0x01 BR unconditional branch 0x02 BNC branch if no carry

0x03 BC branch if carry 0x04 BP branch if positive

0x05 BM branch if minus 0x06 BZ branch if zero

0x07 BNZ branch if nonzero 0x08 BM1 branch if -1

0x09 BNM1 branch if not -1 0x0A BK break (software interrupt)

0x0B RS return from sub (R12 is SP) 0x0C BS branch to sub (R12 is SP)

0x01-0x09 and 0x0C have a second byte which is a signed 8-bit

displacement. If you want a 16-bit jump, you can push it on the stack

and RS.

That's it, 28 instructions, 300 bytes of machine code to implement

them. And I thought the 6502 was already reasonable on code density,

so this was apparently quite a win.

It's pretty terrible compared to Squeak's bytecode, though. I think

our fib microbenchmark should do fine, since it's all arithmetic and

local jumps. Let's assume a calling convention that puts the first

argument in R0 and returns the return value in R0. (I don't care

where other arguments go; they can go hang, because this function only

has one.) Here's my first attempt, which may be buggy:

FIB DCR R0 ; subtract 2 by decrementing twice

DCR R0

BM BASE ; if it was <2, go to the base case

INR R0 ; re-increment, so R0=n-1

STD @R12 ; save a copy of n-1 on the stack

BS FIB ; recurse; now R0=fib(n-1)

ST R4 ; save fib(n-1) so we can retrieve n-1

POPD @R12 ; now we have n-1 in R0

ST R3 ; stick n-1 in R3 so we can use R0 to save fib(n-1) on stack

LD R4 ; now R0=fib(n-1) again

STD @R12 ; and we push fib(n-1) on the stack

LD R3 ; now R0=n-1

DCR R0 ; now R0=n-2

BS FIB ; now R0=fib(n-2)

ST R3 ; we have to get it out of the way so we can pop fib(n-1)

POPD @R12 ; great, R0=fib(n-1) and R3=fib(n-2)

ADD R3 ; now R0 = fib(n)

RS ; return

BASE SUB R0 ; base case: R0=R0-R0=0

INR R0 ; increment

RS ; return

That's 21 instructions, three of which have parameter bytes, so 24

bytes. It may be possible to cut this by a couple of bytes, but not

more, so it's not really a win over Squeak's system. But it's not a

huge loss. (As I said, the code may be buggy, but it's probably good

enough for size estimation.)

Suppose we were trying to translate

CompiledMethod>>copyWithTrailerBytes: bytes from earlier. You could

imagine starting like this, with self in R1 and bytes in R2, and a

calling convention that requires us to preserve all registers,

including arguments (but, naturally, lets us use the stack), except

for R0.

CWTB LD R3 ; We don't have to save anything to preserve it

STD @R12 ; from the call, because of the calling

; convention, but we do need a place to keep the

; return value.

BS \*+1 ; These four instructions, 7 bytes, are a far call.

SET INITIALPC

STD @R12

RS

ST R3 ; store return value in R3 (start)

LD R4 ; now clear out R4 to receive "end"

STD @R12

BS \*+1

SET ENDPC

STD @R12

RS

...

POPD @R12

ST R4

POPD @R12

ST R3

RS

We're only two lines of code into the method, and we're already at 24

bytes, where the Smalltalk had used six bytecodes and two literals for

a total of 14 bytes; and that's glossing over the issue of polymorphic

sends for now, assuming that you could compile each "virtual function"

into a real function that you could far-call. "end - start + 1 +

bytes size", if we write it monomorphically for 16-bit integers, looks

something like this:

LD R7 ; clearing out another reg

STD @R12

LD R4 ; end

SUB R3 ; - start

INR R0 ; + 1

ST R7

LD R8 ; a temp slot for expression result

STD @R12

LD R1 ; also we have to change "self"

STD @R12

LD R2 ; bytes

ST R1 ; self <- bytes

BS \*+1

SET SIZE

STD @R12

RS

ADD R7 ; pedantically, this is "bytes size + (end - start + 1)"

ST R8

That's 18 instructions plus three parameter bytes, for 21 bytes.

Squeak's version was from byte 44 to byte 51, 8 bytecodes, referring

to no literals. Unsurprisingly, I guess, SWEET 16 was roughly

equivalent on "fib", but much worse on more realistic code.

NanoVM: Java Bytecodes on the AVR

---------------------------------

NanoVM is an AVR implementation of Java bytecode; it is about 7100

bytes of AVR machine code and includes garbage collection, arithmetic,

inheritance, presumably polymorphism, and needs about 400 clock cycles

per Java bytecode, plus 256 bytes of RAM for the VM. However, it only

supports "a small subset of the Java language", without "exceptions,

threads, floating point arithmetic and various other things like

e.g. inheritance from native classes."

As I posted previously, the Java bytecode instruction format looks

like it's in the same ballpark with Squeak's, but the bytecode file

format may have some hefty overhead; adding a second copy of the "fib"

method to Fib.java, under a different name, inflated the .class file

by 82 bytes, from 577 bytes to 659, even though javap -c only shows 15

bytecode instructions occupying 21 bytecode slots in the method.

So it's probably possible to fit a bytecode engine similar to Squeak's

into 8 kilobytes of ROM, but 4 kilobytes may be pushing it. Two

orders of magnitude performance loss is heavy but may be acceptable.

Code-Compact Data Structures

----------------------------

If you want a flexible but painfully slow language to run on a machine

without much code space, you probably need some built-in way to

represent common data structures so that you don't have to implement

them yourself in your user-level code. The usual set present in

modern high-level languages (JavaScript, Python, Lua, Perl) includes

numbers, (generally immutable) strings, dictionaries, and mutable,

growable lists or arrays.

Symbols, as in Lisp or Smalltalk, are probably a very useful

optimization in this setting; they allow you to throw away the keys to

your dictionaries if you never print them out.

You may be able to save code space by implementing strings as arrays

or lists of character or integer objects, but the run-time space cost is

terrible; this may be OK if you never or rarely have strings.

Lua's dictionary ("table") implementation uses some hashing technique

I don't understand to be able to operate with a load factor of 100%; I

don't know how much code it needs. FORTH's dictionary structure is

probably the simplest efficient growable dictionary structure: an

eight-entry hash table with separate chaining. If you have

space-efficient resizable arrays, you could perhaps store each chain

in one of those instead.

Here's a working implementation in my pidgin OCaml of these growable

arrays and hash tables. I wrote it in OCaml because I don't have an

assembler handy, that's the only language implementation I have handy

that produces reasonably compact assembly code, and I haven't written

enough code in any assembly language to be able to write this in

assembly from memory. It's 42 lines of OCaml code and comes out to

477 instructions, and it omits only two necessarily polymorphic sends:

one for hashing in get\_table, and one for equality testing in hsearch.

I'm pretty dissatisfied with the number of instructions there --- I

feel like I could do better, by a factor of 2 or 3 --- but the example

at least provides an upper bound that doesn't look insane.

(\* code-compact data structures. \*)

(\* The point of this file is not to provide data structures you'd want

to use in OCaml (OCaml provides other data structures) but to see

how much assembly code they compile to. \*)

(\* growable array \*)

type 'a ary = { mutable a: 'a array; mutable n: int;

mutable allocated: int } ;;

exception Out\_of\_bounds of int ;;

(\* emptyary: 21 PowerPC instructions \*)

let emptyary () = { a = [||]; n = 0; allocated = 0 } ;;

(\* aryappend: 121 PowerPC instructions \*)

let aryappend a i =

(if a.n = a.allocated then

let newalloc = a.allocated \* 2 + 1

in let newary = Array.make newalloc i

in (

(\* normally we would use Array.blit here, but the point is

to count instructions \*)

for i = 0 to a.n - 1 do newary.(i) <- a.a.(i) done ;

a.a <- newary ;

a.allocated <- newalloc

)

) ;

a.a.(a.n) <- i ;

a.n <- a.n + 1

;;

(\* boundscheck: 30 PowerPC instructions \*)

let boundscheck a n =

if n < 0 || n >= a.n then raise (Out\_of\_bounds n)

else () ;;

(\* aryat: 40 PowerPC instructions \*)

let aryat a n = boundscheck a n; a.a.(n) ;;

(\* aryatput: 40 PowerPC instructions \*)

let aryatput a n i = boundscheck a n; a.a.(n) <- i ;;

(\* end of growable array code, totaling 252 PowerPC instructions,

which I think is probably 1008 bytes of machine code. \*)

(\* hash table. Specialized for integer keys because OCaml doesn't

support polymorphic sends to integers. \*)

type 'a hashtable = (int \* 'a) ary array ;;

exception Key\_not\_found of int ;;

(\* hashint: 2 PowerPC instructions \*)

let hashint i = i land 7 ;;

(\* hsearch: 41 PowerPC instructions \*)

let rec hsearch tbl k i =

if i = tbl.n then raise (Key\_not\_found k)

else let kk, v = aryat tbl i

in if kk = k then i else hsearch tbl k (i+1) ;;

(\* get\_table: 35 PowerPC instructions \*)

let get\_table h i = h.(hashint i) ;;

(\* hashput: 53 PowerPC instructions \*)

let hashput h i nv =

let tbl = get\_table h i and newpair = (i, nv)

in try let pos = hsearch tbl i 0 in aryatput tbl pos newpair

with Key\_not\_found \_ -> aryappend tbl newpair ;;

(\* hashget: 17 PowerPC instructions \*)

let hashget h i =

let tbl = get\_table h i

in let (\_, v) = aryat tbl (hsearch tbl i 0)

in v ;;

(\* hashhaskey: 32 PowerPC instructions \*)

let hashhaskey h i =

try ignore(hashget h i); true with Key\_not\_found \_ -> false ;;

(\* newtable: 45 PowerPC instructions \*)

let newtable () =

let rv = Array.make 8 (emptyary ())

in for i = 1 to 7 do rv.(i) <- emptyary () done ; rv ;;

(\* end of hash table code, totaling 225 PowerPC instructions,

which I think is probably 900 bytes of machine code. \*)

Here's a pidgin Squeak version of just the resizeable array:

'From Squeak3.8 of ''5 May 2005'' [latest update: #6665]'!

Object subclass: #Tinyarray

instanceVariableNames: 'a n allocated'

classVariableNames: ''

poolDictionaries: ''

category: 'My stuff'!

!Tinyarray commentStamp: '<historical>' prior: 0!

Tiny OrderedCollection to see how small we can make stuff.!

!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:32'!

add: i

"max bytecode = 59"

n = allocated ifTrue: [| newalloc newary |

newalloc := allocated \* 2 + 1.

newary := Array new: newalloc.

1 to: n do: [:ii| newary at: ii put: (a at: ii)].

a := newary.

allocated := newalloc.

].

a at: (n+1) put: i.

n := n + 1.! !

!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:33'!

at: nn

"max bytecode = 16"

self boundscheck: nn.

^ a at: nn.! !

!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:33'!

at: nn put: i

"max bytecode = 17"

self boundscheck: nn.

^ a at: nn put: i.! !

!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:37'!

boundscheck: nn

"max bytecode = 43"

(nn < 1 or: [nn > n]) ifTrue: [Error new

signal: 'subscript out of bounds: ', nn printString].

! !

!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:34'!

initialize

"max bytecode = 20"

a := {}.

n := 0.

allocated := 0.! !

!Tinyarray methodsFor: 'as yet unclassified' stamp: 'kjs 3/11/2007 16:36'!

size

"no bytecode --- quick-return field 1.

Would probably be max bytecode = 6 otherwise."

^ n.! !

This puts all the code for the class (except its instance variable

list) in 161 bytes, including ten four-byte literals, so we'd be down

20 bytes if the literals were 16-bit instead. This is noticeably less

than the probably 1008 bytes OCaml wanted for essentially the same

code. The corresponding Tinyhash is 301 bytes of methods, including

65 (20%!) for the 'keys' method I left off of the OCaml version and

100 bytes of literal pointers. As variants, I did a version using no

hashing at all, just Associations in a Tinyarray, which was 199 bytes,

a version that used parallel arrays instead of Associations, which

was 329 bytes (9.3% larger). A minimal variant of the Association

class is 31 bytes (of methods) so I suspect that including

Associations is probably a win --- they don't have to save much code

anywhere else to be an absolute win.

(I also wrote a version that was just an alist, without Associations

or hashing, which was 131 bytes, and a hash table wrapped around that,

which was 154 bytes plus a 27-byte Tinyarray>>#do: method, which all

adds up to be 312 bytes, slightly larger than the non-alist-based

hash-table version; but Tinyarray>>#do: is likely to be useful in

other contexts as well.)

(All of this code depends on very little from Array --- it doesn't

fall back on its bounds-checking at all, doesn't query it for its

size, just uses #at: and #at:put: and allocates new arrays of fixed

size --- so it should be able to run atop a very primitive Array

implementation.)

Library Design

--------------

Historically people have often made the mistake of thinking that

computers were for computing --- that is, arithmetic, with numbers.

But the number-handling in many programs is confined the lower levels.

Consider these numbers, again from Squeak:

method name calling methods

sinh 0

ln 13

tan 15

squared 40

raisedTo: (a power) 55

signal: (raising an error) 78

sin 78

-> (creating a pair) 114

signal (raising an error) 119

on:send:to: (metaprograms) 117

\\ (integer modulo) 263

addMorph: (GUI design) 333

next (stream input) 493

bitAnd: (bit twiddling) 609

value (invoking a thunk) 674

notNil (testing for nil) 702

collect: (mapcar) 862

// (integer divide) 912

nextPut: (stream output) 968

add: (incremental constr) 1055

\* (multiply) 1919

at:put: 1990

<= (numeric comparison) 2186

@ (creating a pair) 2372

, (sequence concatenation) 2399

do: (sequence iteration) 2508

at: (collection indexing) 3290

- (arithmetic) 3684

new (instantiating class) 5150

+ 5186

= (comparison) 5342

This is out of about 50 000 methods.

Transcendental functions are really unpopular. Most of the math isn't

all that popular, in fact, compared to things like operations on

sequences. #+ is an exception, but it's used for several things

besides arithmetic on numbers: sound mixing, pointer arithmetic, color

mixing, date/time manipulation (which is arguably numerical),

animation compositing, and voice composition. I suspect that the

surprising popularity of #- is due to Smalltalk's unfortunate decision

to index its collections from 1 and use closed intervals everywhere,

resulting in lots if "-1" in otherwise clean, arithmetic-free code.

The methods for things like iteration and conditional testing are

probably more widely used than any of the above, but the compiler

inlines them, so I can't get statistics easily.

In general, a system that provided no arithmetic \*at all\* would be of

limited use, but it's possible to get surprisingly far without

floating-point or transcendental functions, let alone complex numbers;

you probably get +, -, and < almost for free. It's likely that you'd

be better off devoting precious ROM space to some kind of flexible

collection classes than to transcendental functions, rational numbers,

or possibly even division.

Other Existing Small Interpreters

---------------------------------

\* S21

Jeff Fox's S21 simulator for the MuP21 microprocessor (1995-1998) is a

187KB MS-DOS EXE file. It includes the simulator for the processor, a

a single-stepping debugger with an MS-DOS console user interface, and

it itself is running inside the FPC FORTH virtual machine, which also

includes an interactive development environment with a program editor,

a virtual memory system, and cooperative multitasking.

\* Squeak?

On this Intel Mac, the Squeak virtual machine binary is 967868 bytes.

I don't understand how that's possible, since I thought it was built

from just the Interpreter and ObjectMemory classes, which are only

10 000 lines of code in total, about the same as the Lua interpreter.

\* pepsi/Albert/the golden box

Ian Piumarta's "pepsi" system, the lowest-level substrate for which

Alan Kay's NSF-funded reinvention-of-programming system, is 144 lines

of C code and compiles to 1451 bytes, or 1602 bytes with inline caches

enabled. It provides very efficient, very dynamic method dispatch,

and a minimal object system, but no bytecode interpreter.

\* The Basic STAMP

The Basic STAMP uses a sort of bytecode for executing BASIC on a PIC;

most of the variable-length instructions are less than a byte long.

Chuck McManis's 1994 article "Decoding the BASIC Stamp" describes

them. They're really little more than a tokenized BASIC. I have no

idea how big the STAMP ROM is.

\* UCSD P-System

The UCSD P-System, a stack-based bytecode interpreter with compilers

from Fortran and Pascal, ran on a lot of small microcomputers back in

the day. Z8080:INTERP.TEXT, the UCSD Pascal Interpreter for the Z-80

and Intel 8080A by Peter A. Lawrence and Joel J. McCormack, is about

6100 lines of 8080 assembly when you include all the things it

.INCLUDEs. (I found this in a file called I5Z80Interp.TXT.) This

includes floating-point math, set arithmetic, transcendental

functions, booting from disk, character terminal addressing,

single-stepping, virtual memory, a tokenizer for Pascal, binary search

trees, I/O interfaces for CP/M, and all kinds of such nonsense.

However, I don't have an 8080 assembler handy on this Mac, and there

are a fair number of macros (on one hand) and conditional compilation

(on the other), so I'm not sure how big that actually ends up being.

It's 5000 non-comment non-blank lines.

The PDP-11 interpreter ("mainop.mac" or "I.5-PDP-Interp.TXT") seems to

be just the bytecode interpreter itself, and it's only around a

thousand lines of PDP-11 assembly.

The virtual machine instructions listed in the assembly are quite

similar to the set of P-Code instructions in Steven Pemberton and

Martin Daniels's book, "The P-Code Machine" and also Jensen and

Wirth's 1973 "ASSEMBLER AND INTERPRETER OF PASCAL CODE", (PROGRAM

PCODE(INPUT,OUTPUT,PRD,PRR)). Said Jensen and Wirth code is 775

non-comment non-blank lines of Pascal, which suggests very roughly

about 4000 assembly instructions.

\* PICBIT

PICBIT is Feeley and Dubé's bytecoded Scheme implementation for PIC

microcontrollers. It uses a register-based virtual machine with six

registers for object references, plus PC and number-of-args registers;

it has 17 instructions, which are inadequately explained in their

paper, "PICBIT: A Scheme System for the PIC Microcontroller," but

which reflect a very Schemey view of the world, with a "continuation"

register, a CALL instruction that's really JMP-and-adjust-nbargs, and

so on. One of their example programs was 38 lines of Scheme (about

1000 bytes), which compiled to 2150 bytes of bytecode. Larger

programs were inflated by less than half as much, but they were still

much larger than I would expect with other bytecode systems.

I venture to guess that this suggests that their register-based

virtual machine bytecode was a memory hog. Dubé's earlier BIT

bytecode system, after which they modeled their system, used

stack-based bytecode, which was less than half as big on their five

example programs.

However, the entire R4RS Scheme library only took 11248 bytes of

bytecode, so it seems likely that you can get a relatively powerful

set of primitives into not very much space with the general approach

of using bytecode. With BIT, it was under 8000 bytes.

They report 37000 bytecodes per second, but it's not clear whether

they mean on a 10MHz PIC microcontroller or extrapolated to a 40MHz

microcontroller.

\* F-83

F-83 was an indirect-threaded FORTH programming system available on

most microcomputers in 1983, at a time when many of them had 64kiB of

total address space. I have the F-83 books somewhere, but I don't

have a copy of the sources here, and I don't know how big the whole

system was, but I think it was normally all in memory at once --- the

assembler, the FORTH compiler, the indirect-threaded-code interpreter,

the interactive interpreter, the full-screen editor, the decompiler,

and the virtual-memory system (which couldn't be effectively used for

code).

\* Various combinator-graph-reduction machines

I haven't looked very much at these, but I suspect that laziness may

reduce program size. (Perhaps the same thing could be said of

backtracking and concurrency.)

Conclusions: Squeak Rules, We Should Be Able To Do Python In A Few K

--------------------------------------------------------------------

Most of the approaches I looked at are considerably more compact than

PowerPC machine code on the toy "fib" problem, by a factor of 2-4,

with Squeak among the best; this includes Java (?), Squeak,

indirect-threaded 16-bit FORTH, SWEET 16, and the F21 CPU. A couple

(OCaml bytecode, PICBIT) sound much worse on density. Beating a

factor of 4 (25 bytes for the "fib" program) looks difficult. The

factor-of-4 improvement in code density looks realistic from the

Tinyhash and Tinyarray examples, and based on previous work, I think a

virtual machine for some Squeak-like bytecode can be contained in

4000-8000 bytes of machine code, with a rich library requiring a few

thousand more bytes.

Probably Squeak's approach, using a stack for expression intermediate

results (to keep instruction size down), a local-variable vector for

slightly-longer-term usage (to cut down on stack-manipulation noise

words), and a local-literal vector for constants and linkage, with

nearly all instructions contained in a single byte, is the best one

known for tight bytecode. I suspect that conditional return may be a

more compact control-flow primitive than conditional jump, but it

could make compiler implementation more challenging.

I'm going to assert that polymorphism, even (especially!) on

"fundamental" operations that have machine instructions assigned to

them, increases code density. In the case that you don't have any

polymorphism in your program, it costs you very little code density

(none in bodies, maybe a couple of bytes each in definitions), and in

the case where you do, it saves you conditionals. This should hold a

fortiori for multiple dispatch.

If we are going for absolute minimum run-time code size, it's perhaps

best to have a small kernel written in machine code (probably in a

stack-oriented fashion, such that you can put CALL instructions one

after the other with no intervening setup) that implements a fairly

primitive stack-based virtual machine, atop which a more Squeak-like

virtual machine is implemented. (They need not be separate abstract

machines --- perhaps unimplemented bytecodes will trap into a

user-defined instruction handler.) For example, the hash tables and

growable arrays mentioned previously should probably be mostly

implemented in this level; in Squeak bytecode, they need around 500

bytes.

Library design probably makes a big difference in how few literals you

have to use --- if most of the messages in your system belong to a few

small interfaces like #at: and #at:put: or arithmetic, you'll have a

much easier time with the bytecode.

With this approach, it should be possible to get a very slow language,

with flexibility something like Python's, into maybe 2000-6000 bytes

of a microcontroller's ROM. This should allow you to interactively

get out-of-memory errors with great convenience and flexibility.