Module 7: Functions in the virtual machine

Introduction

This week you'll head back to the virtual machine. You'll implement *register windows* and function calls, including optimized tail calls.

- *What am I doing?*
  - Add support for functions to your VM state: register windows and a call stack.
  - Implement `call`, `tailcall`, and `return` instructions.
- *Why am I doing it?*
  - A high-level language needs functions (or methods).
  - Register windows are a fabulous trick that is very efficient and that enables us to avoid worrying about register allocation and spilling in the UFT.
- *How?*
  - Before lab you'll do some reading on procedure calls and register windows.
  - Before lab you'll also get your SVM and UFT ready: to prepare to use your lab time productively, you'll tell the SVM how to load instructions `call`, `tailcall`, and `return`; you'll set up placeholders for `call`, `return`, and `tailcall` instructions in your `vmrun` function; and you'll enable `idump` debugging in `vmrun` (if you haven't already).

    And so you can try using the assembly code distributed to support the lab, you'll tell your assembly-language parser that blank lines are OK.
  - In lab you'll extend your VM state with a call stack and a register-window pointer, and you'll implement the `call` instruction (for calling VM functions only). There's a decent chance that you also can do the `return` instruction.

– After lab, you'll implement tail calls, and you'll add overflow checks to both `call` and `tailcall` instructions.

– At the end of the week, you'll deliver a virtual machine that works with first-order functions.

The module step by step

Before lab

(1) *Update your SVM source code.* I fixed a show-stopping bug in `disasm.c`. To get the fix, you'll need to `git pull`. I don't expect any merge conflicts.

(2) *Review hardware procedure calls (recommended).* You'll want to remind yourself how calls and returns work and how they interact through a call stack. A reminder about passing parameters in registers won't hurt either.

If you're not sure what to review, try one of these suggestions:

- For a high-level view of a call stack, there's a decent short video from CS 50. What the speaker calls "a procedure on pause" is more usually called "a suspended activation."
- For a slightly lower-level view that mentions machine registers, complete with a little animation, there's a blog post by Connor Stack.
- You could just dive straight into the comparison section of the procedure-call handout, which includes an example of C code compiled to AMD64 assembly language.

(3) *Read about how to implement calls (essential).* Implementing calls is the entire module, and I don't expect you to have grasped it all before lab. But in order to be productive during lab, you will need to have some idea what is going on.

1. To develop your **overall understanding**, read my handout on Understanding Procedure Calls.
2. To master the **details of instructions' behavior and coding,** read the operational semantics.

Before lab, take notes on two topics:

- *What goes into an activation* (a frame on the call stack) and how you want to represent it
- The *actions of a call instruction*

For both sets of notes, the operational semantics will be useful.

(4) *Add procedure-call instructions to the SVM loader (essential).* In lab, you'll implement instructions you can run, but you need to come into lab already being able to load them.

You need three instructions: call, tail call, and return. Each needs an internal and an external opcode, a parsing function, and an unparsing template. The choices I recommend are as follows:

Opcode in `.vo`	Parser	Unparsing template
`call`	`parseR3`	`rX := call rY (rY+1, ..., rZ)`
`return`	`parseR1`	`return rX`
`tailcall`	`parseR2`	`tailcall rX (rX+1, ..., rY)`

Add the opcodes to `opcodes.h` and the instructions to `instructions.c`.

(5) *Add procedure-call cases to* `vmrun` *(essential).* For each opcode you added in the previous step, add a case to your `vmrun` function. For now, each case can simply `assert(0)`.

(6) *Make sure your SVM has a debugging mode (recommended).* Check your `vmrun` function to be sure you can configure it to call `idump` on every instruction. Here's an example of how to use debugging mode and what the results look like, computing four factorial:

Trace of SVM computing and checking four factorial (click arrow to reveal)

```
nr@homedog ©150vm/s/uft> uft fo-vo fact.scm | env SVMDEBUG=decode svm
[  0 @ $r0  ] r0 := function 0x560d973244f0
[  1 @ $r0  ] _G["!"] := r0                                  r0 = function 0x560d973244f0
[  2 @ $r0  ] r0 := _G["!"]
[  3 @ $r0  ] r1 := 4
[  4 @ $r0  ] r0 := call r0 (r0+1, ..., r1)                  r0 = function 0x560d973244f0  r1 = 4
[  0 @ $r0  ] r2 := 0
[  1 @ $r0  ] r2 := r1 == r2                                 r1 = 4  r2 = 0
[  2 @ $r0  ] if r2 then                                     r2 = #f
[  4 @ $r0  ] r2 := _G["!"]
[  5 @ $r0  ] r3 := 1
[  6 @ $r0  ] r3 := r1 - r3                                  r1 = 4  r3 = 1
[  7 @ $r0  ] r2 := call r2 (r2+1, ..., r3)                  r2 = function 0x560d973244f0  r3 = 3
[  0 @ $r2  ] r2 := 0
[  1 @ $r2  ] r2 := r1 == r2                                 r1 = 3  r2 = 0
[  2 @ $r2  ] if r2 then                                     r2 = #f
[  4 @ $r2  ] r2 := _G["!"]
[  5 @ $r2  ] r3 := 1
[  6 @ $r2  ] r3 := r1 - r3                                  r1 = 3  r3 = 1
[  7 @ $r2  ] r2 := call r2 (r2+1, ..., r3)                  r2 = function 0x560d973244f0  r3 = 2
[  0 @ $r4  ] r2 := 0
[  1 @ $r4  ] r2 := r1 == r2                                 r1 = 2  r2 = 0
[  2 @ $r4  ] if r2 then                                     r2 = #f
[  4 @ $r4  ] r2 := _G["!"]
[  5 @ $r4  ] r3 := 1
[  6 @ $r4  ] r3 := r1 - r3                                  r1 = 2  r3 = 1
[  7 @ $r4  ] r2 := call r2 (r2+1, ..., r3)                  r2 = function 0x560d973244f0  r3 = 1
[  0 @ $r6  ] r2 := 0
[  1 @ $r6  ] r2 := r1 == r2                                 r1 = 1  r2 = 0
[  2 @ $r6  ] if r2 then                                     r2 = #f
[  4 @ $r6  ] r2 := _G["!"]
[  5 @ $r6  ] r3 := 1
[  6 @ $r6  ] r3 := r1 - r3                                  r1 = 1  r3 = 1
[  7 @ $r6  ] r2 := call r2 (r2+1, ..., r3)                  r2 = function 0x560d973244f0  r3 = 0
[  0 @ $r8  ] r2 := 0
[  1 @ $r8  ] r2 := r1 == r2                                 r1 = 0  r2 = 0
[  2 @ $r8  ] if r2 then                                     r2 = #t
[  3 @ $r8  ] goto $PC + 7
[ 10 @ $r8  ] r0 := 1
[ 11 @ $r8  ] return r0                                      r0 = 1
[  8 @ $r6  ] r0 := r1 * r2                                  r1 = 1  r2 = 1
[  9 @ $r6  ] return r0                                      r0 = 1
[  8 @ $r4  ] r0 := r1 * r2                                  r1 = 2  r2 = 1
[  9 @ $r4  ] return r0                                      r0 = 2
[  8 @ $r2  ] r0 := r1 * r2                                  r1 = 3  r2 = 2
[  9 @ $r2  ] return r0                                      r0 = 6
[  8 @ $r0  ] r0 := r1 * r2                                  r1 = 4  r2 = 6
[  9 @ $r0  ] return r0                                      r0 = 24
[  5 @ $r0  ] check "(! 4)", r0                              r0 = 24
[  6 @ $r0  ] r0 := 24
[  7 @ $r0  ] expect "24", r0                                r0 = 24
[  8 @ $r0  ] halt
The only test passed.
nr@homedog ©150vm/s/uft>
```

To turn this mode on in your own SVM, you'll need to take these steps:

- In `vmrun`, initialize two variables when `vmrun` starts:

  ```
  const char *dump_decode = svmdebug_value("decode");
  const char *dump_call   = svmdebug_value("call");
  (void) dump_call;  // make it OK not to use `dump_call`
  ```

- In `vmrun`, immediately after decoding an instruction, before using `switch` on the opcode, if `dump_decode` is set, call `idump`:

  ```
  if (dump_decode)
     idump(stderr, vm, pc, instr, window_position, pRX, pRY, pRZ);
  ```

  The parameters are as follows:

`vm`	The VM state
`pc`	The current program counter (as an integer)
`instr`	The instruction word just decoded

`window_position`	An integer saying where the current register window points
`pRX`, `pRY`, `pRZ`	Pointers to the **values** stored in registers X, Y, and Z

The `window_position` can be any number that meaningfully tells *you* the position of the register windows. All `idump` does with it is print it.

- If you want to write out additional information at call, return, and tail-call instructions, do so only when `dump_call` is not `NULL`.

To make these codes compile, your `vmrun.c` will need to `#include` both `"svmdebug.h"` and `"disasm.h"`.

(7) *Update your assembly-language parser to support blank lines (lightly recommended).* In file `asmparse.sml`,

- In the `instruction` parser, change each *single* occurrence of `eol` to accept *one or more* newlines: `many1 eol`.
- Change function `parse` to accept initial blank lines:

```
val parse =
  Error.join o
  P.produce (Error.list <$> (many eol >> many instruction)) o
  List.concat
```

These changes will enable your parser to handle assembly code with blank lines and comment lines.

Recompile your UFT.

Lab

(8) *Define a procedure activation.* Using abstract-machine state from the operational semantics, put fields in the definition of `struct Activation` in file `vmstack.h`.

(If you're not sure how to get started here, you could consider jumping forward to step (10) and then coming back.)

(9) *Enlarge the VM state.* Increase the total number of registers in your VM state. And add support for a *finite* stack of activation records.[1] Sizes are up to you, but you probably want capabilities that are comparable to other interpreters. For the call stack, here are some data points:

[1] It is possible to allocate procedure activations dynamically and to keep running until `malloc` fails, but this plan is terrible, on two grounds: (1) It makes calls inefficient. (2) If a user runs a VM program that has an infinite recursion, it's not sensible for the SVM to run until it gobbles up as much address space as the operating system will allow it—the sensible response is to cap the size of the call stack, and when the call stack overflows, to issue an informative error message.

- The interpreters used in CS 105 support recursion to depth of around 5000 (the exact number can depend on the size of a single activation record).
- The standard interpreter for the Icon programming language defaults to a stack of 10,000 words, which I'm guessing might be ballpark of 2,000 activations.
- In 2010, a standard Python interpreter had a default stack limit of 1000 activations.

For the register file, I don't have data, but 10 registers per activation might be in the right ballpark.

(10) *Implement call.* When given a function, the `call` should push a new activation onto the call stack, shift the register window, and start executing the new function from instruction 0. **For a precise specification, consult the operational semantics.** When given a non-function, the `call` instruction should report a checked run-time error (step (12)).

If you're not sure how to approach this step, especially if you skipped ahead from step (8), ask yourself these questions:

- How do I transfer control to the function I wish to call (the "callee")?
- How do I adjust the register window to ensure that the callee sees its own value in register 0, its first parameter in register 1, and so on?
- What do I need to **remember** so that when the call completes, I can continue executing my own code in the caller? (It's the stuff you need to remember that goes into an activation record.)

These questions are mostly answered by the operational semantics. It helps to relate the notations in the operational semantics to the variables in the code: what is R, what is I, and so on.

The operational semantics doesn't account for the limited space allocated to procedure activations and register windows. A `call` might overflow either the call stack or the register file, but for lab, don't worry about overflow—you'll deal with it in step (14).

In the implementation of the `call` instruction, I recommend you define these local C variables:

`destreg` The number of the register that is meant to receive the result of the call. For the `call` instruction this is the X field.

`funreg` The number of the register that holds the function being called. For the `call` instruction this is the Y field.

`lastarg` The number of the register that holds the last argument (or in the case where there are no arguments, again the function being called). For the `call` instruction this is the Z field.

Then write your code in terms of these variables.

Test your `call` instruction using the testing handout:

A. See if control is transferred: the `callt` test calls a function that runs `check` and `expect`, then halts.

B. See if an argument can be passed and the register window works correctly. The `callarg` test code puts a function in register 200 and an argument in register 201. For the test to pass, the callee must see the actual parameter in register 1.

C. See if the SVM detects when a function is called with the wrong number of arguments. Use the `callwrong.vserr` test.

Don't overlook the run-time disassembler built into the SVM. As mentioned in step (6), your `vmrun` function should have the option of calling `idump` after every instruction is fetched. This feature is great for debugging, and if you followed the advice in step (6), yours should be set up to be enabled by running `env SVMDEBUG=decode svm`.

Caution!

Beware of trying to implement call by making a recursive call to `vmrun`. It can be made to work, but there are risks:

- I hope you'll exit the course with a strong grasp of recursion. But if you rely on C recursion instead of implementing it yourself, you might not grasp it as well.
- You won't be able to implement tail calls this way.
- You won't be able to spit out a stack trace on error.
- When we get to garbage collection in module 11, you'll be screwed and you'll have to do the calls over.

(11) *Implement return.* Implement the `return` instruction. Informally, this instruction should grab the return value, pop the caller's activation off the stack,[2] then restore the instruction stream, program counter, and register window. It should then set the destination register and continue executing in the caller's code. **For a precise specification, consult the operational semantics.**

Once your `return` is implemented, test it using the testing handout:

D. See if control is transferred there and back, and if the register window is restored correctly. Test `returnt` is rigged so that if the `return` doesn't restore the proper program counter, code will hit some `error` instructions.

E. See if a value can be returned correctly. Test `returnv` tests that value 1852 is correctly returned from a function.

[2] Or if the stack is empty, signal a checked run-time error

F. Test recursion. Test `fact` recursively computes 5 factorial.

After lab

Undefined functions

(12) *Report calls to undefined functions.* When testing, it's all too common to forget that the UFT does not have the same predefined functions built in, the way the `vscheme` interpreter does:

```
$ echo '(gcd 39 27)' | vscheme -v
3
echo '(gcd 39 27)' | uft fo-vo | svm
Run-time error:
    Tried to call nil; maybe global `gcd` is not defined?
```

That diagnostic is invaluable. To get it, find the place in your `vmrun.c` where your `call` instruction can detect `nil` in the function position. To find the name of the global variable form which the function was set, if any, call `lastglobalset` on `funreg`. The function is documented in file `disasm.h`.

Tail calls

If you want a super-quick review of tail calls, you could do worse than my lecture notes from CS 105. For a detailed explanation of tail-call optimization, I recommend a blog post by Richard Wild.

(13) *Implement tail call.* Implement the `tailcall` instruction. Informally, this instruction should update the instruction stream, use `memmove` to shift register values into place, and start executing the new function from instruction 0. A tail call can't overflow the call stack, but it *can* overflow the register file (you'll deal with overflow in step (14)). **For a precise specification, consult the operational semantics.**

A tail call has no `destreg`, but you can and should define local variables `funreg` and `lastarg`, which for a tail call are instruction fields X and Y. If you have used these variables wisely, you can copy, paste, and edit code from your `call` instruction in step (10).

The registers that have to be shifted are the function register and all the arguments: registers `funreg` to `lastarg`, inclusive. Beware of off-by-one errors.

Once your `tailcall` is implemented, test it using the testing handout:

G. Test `tail` makes a `tailcall r0(r1)`, so it has a chance of working even if the registers aren't moved. The test counts down from 555,000 by ones, making a tail call at each step. The `uscheme` and `vscheme` interpreters can't run this test, because they don't optimize tail calls. The `uschemeplus` interpreter can run it.

Run this test using `valgrind`, which may detect if you have a bug that makes your call stack overflow.

H. Test `tailm` uses tail calls to add 12 to 99, 1.2 million times. This one requires that the registers be shifted into place correctly. The test runs over 10 times faster than the same test in `uschemeplus`.

Stack overflow

(14) *Detect overflow.* The SVM has two data structures that can overflow:

- A `call` instruction can overflow the call stack by attempting to push an activation when the stack is full. This situation should be handled by a call to `runerror`.

  A `tailcall` instruction does not push an activation record, so it cannot overflow the call stack.

- A `call` instruction can overflow the register file by attempting to shift the current window in such a way that not all register names (0 to 255) point to correctly allocated memory. This situation should also be handled by a call to `runerror`.

  A `tailcall` can also overflow the register file.

Extend your `call` and `tailcall` instructions so they detect overflow.

Overflow can be caused in two ways: run out of registers while still having plenty of activation records, and run out of activation records while still having plenty of registers. Demonstrate your understanding with these test files:

`overstack.vs` Contains a program that, when executed, overflows the call stack of your SVM. (A good way to do this is with a deep recursive call.)

`overreg.vs` Contains a program that, when executed, *does not* overflow the call stack of your SVM, but *does* overflow the register file. It is sufficient if the register file overflows *before* the call stack. (A good way to do this is to adapt the previous program so the recursive function uses register 255, which will then eat up 256 registers per call.)

`overtail.vs` Contains a program that is as similar as possible to the program in `overstack.vs`, but in which the recursive call is a `tailcall`. This program should run to completion without error.

What and how to submit

(16) On Monday, *submit the homework.* In the `src/uft` directory you'll find a file `SUBMIT.07`. That file needs to be edited to answer the same questions you answer every week.

To submit, you'll need to copy your working tree to the department servers. We recommend using `rsync`, but `scp` also works.

Now log into a department server, change to your working tree, and submit your entire `src` directory:

```
provide cs106 hw07 src
```

(17) On Tuesday, *submit* your reflection. Create a plain text file `REFLECTION`, which will hold your claims for project points and depth points.

For each project point you claim, write the number of the point, plus whatever is called for in the section "How to claim the project points"—usually a few sentences.

Now copy your `REFLECTION` file to a department server and submit it using `provide`:

```
provide cs106 reflection07 REFLECTION
```

Learning outcomes

Outcomes available for points

You can claim a project point for each of the learning outcomes listed here. Instructions about how to claim each point are found below.

1. *Operational semantics.* You understand the operational semantics of register windows.
2. *Operational semantics.* You understand the operational semantics of the `return` instruction.
3. *Overflow.* You can build an SVM that detects stack overflow, preventing memory errors.
4. *Optimized tail calls.* You can use tail calls to write a deeply recursive function that avoids overflow.
5. *Invariants, part I.* You know the invariants satisfied by your representation of a VM function.
6. *Invariants, part II.* You know how VM-function invariants are used in your `vmrun` function.
7. *Performance of register windows.* You understand the benefits and costs of register windows.

8. *Error detection.* You know how to provide a useful diagnostic when things go wrong at run time.

You can claim depth points for doing things with the call stack.

9. *Stack trace [3 points].* When a run-time error occurs, your code produces a stack trace. To make the stack trace useful, you extend the representation of `FUNCODE` to include a string that encodes the function's name (if known) and source-code location (if known). The extension is supported by your K-normal form, assembly code (including parser), object code, SVM loader, and `vmrun`.

   This work may count toward depth points or toward module 12, but not both.

10. *Error checking and recovery [4 points].* You implement a `check-error` instruction, which works by putting a special frame on the call stack. If that frame is returned to by normal execution, the `check-error` test fails. But if an error occurs while that frame is on the stack, the test succeeds.

    This work may count toward depth points or toward module 12, but not both.

11. *Code improvements for tail calls, part I: Register targeting. [3 points]* You define an optimization pass for K-normal form which changes the register bindings so that every tail call is made to a function in register 0 with arguments in registers 1 to N. This is done not by adding new bindings (which the VM can to more efficiently) but by changing the registers used in existing bindings. Such a renaming is always possible, and the cost can be limited to at most a single new VM instruction.

    In native code, this optimization is important. In a VM, its payoff is small. But it enables the next optimization, which has high payoff in both settings.

    This work may count toward depth points or toward module 12, but not both.

12. *Code improvements for tail calls, part II: Optimized tail recursion [2 points].* Building on the previous step, your UFT recognizes when a tail call is a *recursive* call, and it optimizes that tail call into a `goto` instruction. (A good heuristic for recognizing that a tail call is recursive is that it calls a function in register 0.)

    This optimization is essential for functional languages: a tail-recursive function is optimized into a loop.

    This work may count toward depth points or toward module 12, but not both.

13. *Curried functions [5 points].* You implement automatic currying as described in the handout on understanding procedure calls.

The strategy I recommend is called "eval/apply" and is described by Simon Marlow and Simon Peyton Jones in a landmark 2006 paper. The paper has way more detail than you need, but the important bits will involve what new species of value and what new species of activation record are useful for implementing curried functions.

This work may count toward depth points or toward module 12, but not both.

14. *Exceptions [5 points].* Implement exceptions and exception handlers. I recommend defining two VM instructions that resemble the "long label" and "long `goto`" features from chapter 3 of my book, which I can give you. Then build exceptions on top of that.

    This work may count toward depth points or toward module 12, but not both.

15. *Coroutines [10 points].* A coroutine is a synchronous, cooperative, lightweight, concurrent abstraction. It's like a thread, only simpler. A coroutine is represented by a suspended VM state. Coroutines need at least the following:

    - A new form of value whose payload is a VM state.
    - Instructions that create a coroutine (from a function) and wait for one to terminate.
    - Instructions that transfer control between coroutines. These can be *symmetric* (as in Icon or *asymmetric* (as in Lua).

    Coroutines are a good first step toward fully preemptive concurrency with synchronous communication, as in Erlang or Go.

    This work may count both *toward depth points* and *module 12. If it is counted toward both, the depth points will be halved.*

How to claim the project points

Each of the numbered learning outcomes 1 to 8 is worth one point, and the points can be claimed as follows:

1. To claim this point, give us the number of the line of code in file `vmrun.c` that implements $R \oplus r_0$ in the rule for `call`.
2. To claim this point, give us the number of the line of code in file `vmrun.c` that computes $\sigma(R(r))$ and the line of code that updates location $R'(r')$ in σ.
3. To claim this point, submit files `overstack.vs` and `overreg.vs` from step (14), and confirm that they assemble, load, and detect overflow by the following commands:

```
uft vs-vo overstack.vs | valgrind svm
uft vs-vo overreg.vs   | valgrind svm
```

Confirm that

- Each command causes the `svm` to terminate with a sensible error message.
- The error messages are different in the two cases.
- No memory errors are detected by `valgrind`.

4. To claim this point, submit the `overtail.vs` file from step (14), along with a *brief* explanation—one or two sentences at most—of its similarities to and differences from `overstack.vs`.

5. To claim this point, identify one important invariant that the representation of `struct VMFunction` must satisfy. (*Hint:* If any information is redundant, it must be internally consistent.)

6. To claim this point, explain how the invariant in the previous step is exploited in the implementation of `vmrun`.

7. To claim this point, answer these two questions:

   - Explain what, if any, is the *performance benefit* of using register windows in the SVM, as opposed to a fixed set of 256 VM registers.
   - Explain what, if any, is the performance *cost* (in your VM code) of using register windows in the SVM, as opposed to a fixed set of 256 VM registers.

8. When your SVM runs code that involves a call to an undefined function, as in the Scheme example `(wait---what? 42)`, the error message should show the name of the function. To claim this point, identify the line numbers in `vmrun.c` where this error message is issued. There should be at least two locations: one for ordinary calls and one for tail calls.