Module 8: Translating functions

Contents

Introduction

This week you'll complete your code generator by adding support for functions and calls.

- *What am I doing?*
  - Generate code for functions and function calls, including optimized tail calls.
  - Demonstrate, via tests, that functions work correctly.
- *Why am I doing it?*
  - To be able to run Scheme functions, not just assembly-language functions.
- *How?*
  - Before lab you'll develop equations of translation for function bodies (tail position).
  - During lab you'll implement instructions for calls (including tail calls) and returns.
  - After lab, you'll clean up any remaining issues with primitives, and you'll test your system.
  - On Monday night, you'll submit a UFT and SVM that can translate and run Scheme code in K-normal form with function calls, including optimized tail calls. You'll also submit tests.

The module step by step

Before lab

(1) *Download updates.* Update your git repository in two steps:

- Be sure all your own code is committed. Confirm using `git status`.
- Update your working tree by running `git pull` (or you might possibly need `git pull origin main`). **Expect merge conflicts in the SVM code:**
  - Conflicts in `disasm.c` in the implementation of `fprintfunname`
  - Conflicts in `instructions.c` in the implementation of `isgetglobal`
  - Possible conflicts in `vmstate.c` around function `initialize_global`

  If you have have merge conflicts around `fprintfunname` or `isgetglobal`, then quite likely you have implemented them correctly, in which case you will want to keep your versions.

  You may also encounter a few merge conflicts in `asumutil.sml`, but if so, these should be easy to resolve.

(2) *Study tail position.* In the translation handout from module 6, study these two sections:

- The rules of tail position
- Translating expressions in tail position.

If you want to know more, "tail position" is not a useful search term. A better search term is "optimized tail calls." There is a nice blog post by Richard Wild.

(3) *Write equations of translation for tail position.* The section on translating expressions in tail position describes a translation informally. Define that translation formally by writing a set of translation equations, which you will bring to lab.

- Make sure you write an equation for every syntactic form of K-normal form.
- Make sure that every bullet point in the section is implemented by one or more of your equations.

Lab

You will extend your code generator to support functions and function calls. To help you, I have added a few utility functions to file `asmutil.sml`: `A.call`, `A.areConsecutive`, `A.return`, and `A.tailcall`. Relevant functions are mentioned at the *end* of each step.

(4) *Check your equations for tail position.* Working in groups of three, check your equations from step (3), in two steps:

- Reach consensus in your group about what the equations should be.
- Once you have reached consensus, share your conclusions with me. Or if you can't reach consensus, share your disagreements with me.

(5) *Prepare to code function* `toReturn'`. Replace the body of `toReturn'` with the expression below, changing the value constructor `FUNCODE` as needed so it matches your definition of the form `funcode (x₁, …, xₙ) => e` in `knf.sml`:

```
(case e
  of K.FUNCODE _ => Impossible.exercise "codegen for funcode"
   | _ => Impossible.exercise "toReturn")
```

(6) *Generate code for calls.* In file `codegen.sml`, in your functions `toReg'`, `toReturn'`, and `forEffect'`, fill in translations of the function-call form `x(x₁, …, xₙ)`.

A. Do `toReg'` first, because there you have a destination register. And be aware of the limitations in encoding the `call` instruction! In the instruction

```
r := call x(x₁, …, xₙ)
```

The destination register `r` and the function register `x` are arbitrary, but the other registers are strictly constrained. If any parameters are present, it must always be true that $x_1 = x + 1$ and for every i, $x_{i+1} = x_i + 1$. That is, **the register numbers `x` through `xₙ` must be consecutive.** If they are not, there is a bug in your UFT, and your code generator should call `Impossible.impossible`.

Take advantage of the new utility functions `A.call` and `A.areConsecutive`, which are defined in file `asmutil.sml`.

B. Using your equations, extend `toReturn'` with a case that generates code for the function-call form `x(x₁, …, xₙ)`.

Take advantage of the new utility functions `A.tailcall` and `A.areConsecutive`, which are defined in file `asmutil.sml`.

C. The `forEffect` case is a little weird. The SVM `call` instruction needs a destination register.[1] If the call is being executed only for side effect, but the return is going to update a destination register, what destination register should be updated? Pick any register that the call could kill, and use it as the destination register.

(7) *Generate code that returns values.* Using your equations, add cases to `toReturn` to handle all the remaining value constructors of K-normal form. In some cases, you may be able to use `toReg'` as a helper function.

After this step, only the `funcode (x₁, …, xₙ) => e` form will remain to be implemented.

Fun fact: code generated by `toReturn` always exits the current procedure; it never "falls through." So when an `if` expression occurs in tail position, it is not necessary to label the end of the `if` expression or to include a `goto` instruction targeting that label.

Take advantage of the new utility function `A.return`, which is defined in file `asmutil.sml`.

(8) *Generate code for function bodies.* Finish your code generator by adding cases that translate function bodies (`funcode (x₁, …, xₙ) => e`).

D. Update `toReg'` so that given `funcode (x₁, …, xₙ) => e`, it calls `toReturn` on the function's body.

Similarly update `toReturn`. You may be able to use `toReg'` as a helper function.

E. Confirm that your mutual recursion passes this sanity check:

- Function `toReturn` calls itself, `toReg'` and `forEffect'`.
- Function `toReg'` calls itself, `toReturn`, and `forEffect'`.
- Function `forEffect'` calls itself and `toReg'` but not `toReturn`.

After lab

Wrap up your code generator

(9) *Generate good code for misused primitives.* In any language implementation, **bad code must not take down the translator.** As long as the bad code is not run, a user should be able to translate and load the program as usual. Therefore, if a primitive is used in the wrong context or with the wrong number of arguments, nothing bad should happen unless the primitive is actually run.

[1] We could define a `call` instruction that doesn't need a destination register, but there would be little point.

To understand the context in which a primitive is used, you can pattern-match on it:

- If a primitive matches `P.SETS_REGISTER _`, then an application of the primitive makes sense only in `toReg'`.
- If a primitive matches `P.HAS_EFFECT _`, then an application of the primitive makes sense only in `forEffect'`.

Your code generator should handle user code in which primitives are misused in these ways:

F. *A register-setting primitive is used for effect.* No problem! By design, register-setting primitives don't have any effect, so the primitive is implemented by an empty sequence of instructions.

G. *A effectful primitive is used to set a register.* This is what some language definitions call "an unchecked run-time error" and others call "undefined behavior." We can do as we like. Here are some choices:

- Evaluate the primitive for its effect and leave the register unset.
- If the primitive takes any arguments, set the result register to equal the first argument.
- Set the result register to `nil`.
- Replace the primitive with a call to `error`.

H. *A primitive is used with the wrong number of arguments (optional).* Each primitive has an *arity* which can be queried with function `P.arity`. That arity should match the number of actual parameters. A case where arity doesn't match might look like this:[2]

```
(println (if #f (+ 0) 'fooled-you!))
```

This code runs just fine under `vscheme` or `uscheme`, but if your UFT doesn't check the arity of primitives, it will generate `.vo` code that will not load.

In cases like these, the most sensible thing to do is probably to convert the primitive application to `error`, with a literal error message. That change will shift the problem from load time to run time, where it belongs. But the UFT is not a production compiler, and if you want to pass these forms through so they fail at load time, that's OK with me.

(10) *Tie up any remaining loose ends.* At this point, your `codegen.sml` should be complete and should handle all cases. The only use of `Impossible` or any other uncaught exception should be in a case where the registers in a call are not numbered consecutively—which indicates a bug in your UFT.

I. Make sure every syntactic form has a well-defined translation in each of the three contexts (destinies).

Test your code generator

(11) *Lightweight testing.* Get your system to pass three test cases.

J. *Factorial.* I've hand-translated a factorial function into K-normal form. I've also translated the unit test `(check-expect (factorial 4) 24)`.

```
(let ([$r0 (lambda ($r1)
         (let* ([$r2 0]
                [$r2 (= $r1 $r2)])
               (if $r2
                   1
                  (let* ([$r2 factorial]
                         [$r3 1]
                         [$r3 (- $r1 $r3)]
                         [$r2 ($r2 $r3)])
                      (* $r1 $r2)))))])
  (set factorial $r0))

(begin
  (let* ([$r0 factorial] [$r1 4] [$r0 ($r0 $r1)]) (check
factorial))
   (let* ([$r0 24]) (expect $r0 '24)))
```

Copy this code into file `fact.kn` and confirm that the test passes with both my interpreter and your UFT/SVM:

```
$ vscheme < fact.kn
The only test passed.
$ uft kn-vo fact.kn | svm
The only test passed.
```

K. *Tail calls.* Translate your `overtail.vs` file from module 7 into K-normal form as file `overtail.kn`. Confirm that it translates and runs without overflow.

L. *Tail calls with parameters.* Translate the `times-plus` test from module 7 into K-normal form as file `tailm.kn`. (You can either translate forward from the `tailm.scm` file or translate backward from assembly code `tailm.vs`, both in the module 7 tests handout.) Confirm that the test translates, loads, runs, and passes.

What and how to submit

(12) On Monday, *submit the homework.* In the `src/uft` directory you'll find a file `SUBMIT.08`. That file needs

[2]Primitive + has just one argument; it expects two

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 hw08 src
```

or if you keep an additional `tests` directory,

```
provide cs106 hw08 src tests
```

(13) 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 reflection08 REFLECTION
```

Learning outcomes

Outcomes available for points

Learning outcomes available for project points:

1. *Tail position.* You can identify what expressions occur in tail position.
2. *Calling conventions.* You understand how a call can kill registers.
3. *Undefined behavior.* You have an idea how a translator can exploit undefined behavior.
4. *Translation in tail position.* You understand the point of having special rules for translation expressions that appear in tail position.
5. *Totality of code generation.* You know how to ensure that your translator generates code for any input—as long as there are no bugs in the compiler, code generation always succeeds.
6. *Course design and compiler design.* You can comment on the experience of splitting code generation across two modules.

This module does not include any new opportunities for depth points.

How to claim the project points

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

1. To claim this point, identify every line of code in `codegen.sml` that *pattern matches* on a K-normal form expression of the form `if x then e₁ else e₂`. At **each** such line, analyze the roles of subexpressions `e₁` and `e₂`, and answer these two questions:
   - At that line, are `e₁` and `e₂` *always* in tail position, *never* in tail position, or potentially *some of each*?
   - How do you know?
2. To claim this point, point to your code for ~~part B~~ part C of step (6) (`forEffect applied to a function-call form`). In a sentence or two, explain which register you use as the destination register and why you chose that register.
3. To claim this point, identify the lines of your code that handle the case where an effectful primitive is used to set a register (part G in step (9)). Explain what behavior you chose and why.
4. Suppose that `toReturn'` does not use any special rules and simply delegates to `toReg'` as follows:

   ```
   fun toReturn e = toReg returnReg e  o  S (A.return returnReg)
   ```

   where `returnReg` can be any register that won't be overwritten by `toReg`. To claim this point,
   - Say what changes in the generated code.
   - Say what Scheme program you would write to observe the effect of the change at run time.
5. To claim this point, identify one thing you changed in step (10) to ensure that your code generator is total. If all you had to do was add calls, refer back to the code you submitted for module 6 and note that the only uncovered cases were for calls.
6. This term we split code generation over two modules (6 and 8), with the implementation of call instructions (module 7) in between. I'm considering alternatives:

   The course as it is:

   5. K-normal form
   6. Code generation without calls or returns
   7. Function calls in the SVM
   8. Code generation for calls and returns

   Do SVM calls first:

   5. Function calls in the SVM
   6. K-normal form
   7. Code generation, including calls and returns
   8. Depth points (or other use of a free week)

Do UFT calls first:

5. K-normal form
6. Code generation, including calls and returns
7. Function calls in the SVM
8. Depth points (or other use of a free week)

To claim this point, you must say which of the potential alternatives might improve on your current experience and why. And if you prefer either of the alternatives that free up a week, you must say where in the semester that free week should be placed—and why.

You must explain your reasoning in enough depth that I get some insight into your experience.