Experiments with an Abstract Machine

December 14, 2016

The aim is to build an abstract machine which follows the semantics of OCaml, but which 1) can operate step-by-step 2) keeps around enough information to be able to recompute a source-code representation of the currently-executing portion of the program 3) and its place in the rest of the source code. In Leroy "Functional Programming Languages: Part II: Abstract Machines", a basic arithmetic example is provided, followed by an SECD machine (with tail-call elimintion), and then OCaml's actual bytecode, the Zinc Abstract Machine.

The arithmetic example is implemented like this. Here is the type for programs:

```
type op = Add | Sub | Mul | Div

type prog =
  Int of int
| Op of prog * op * prog
| Underline of prog
```

And here is the type for bytecode instructions:

```
type instr =
  IConst of int
| IOp of op
```

Compilation is very simple, consisting of just conversion to Reverse Polish:

```
let rec compile = function
  Int i -> [IConst i]
| Op (a1, op, a2) -> compile a1 @ compile a2 @ [IOp op]
```

Evaluation is simple, too. The program is run instruction-by-instruction, keeping a stack on to which operands are placed. An operator takes two operands off the stack, and pushes its result. At the end, the answer is left on the stack and may be returned:

```
let calc_op a b = function
  Add -> a + b | Sub -> a - b
| Mul -> a * b | Div -> a / b

let rec run s = function
  [] -> hd s
| IConst i::r -> run (Int i::s) r
| IOp op::r ->
    match s with
      Int n2::Int n1::s' ->
        run (Int (calc_op n1 n2 op)::s') r
```

Running step-by-step is then simple: just process one instruction of the bytecode as above, returning the remaining ones and the new state of the stack. Now, how can we reconstruct the program source code, given the current state of the program and the stack? Here is the uncompilation function. We take the remainder of the program and the current stack. When we find an IOp instruction, we take two things off the stack and build an Op node. This is the conversion back to infix.

```
let rec uncompile s = function
  [] -> hd s
| IConst i::r ->
    uncompile (Int i::s) r
| IOp op::r ->
    match s with
      a::b::s’ -> uncompile (Op (b, op, a)::s’) r
```

We only print out steps where something important happens. For arithmetic, this means an op.

How do we highlight the current redex? It’s easy because the first piece of work to be done is always at the top of the program, since the program is executed in linear order.

Example

```
$ ./arith -e "1 + 2 * (3 + 4)" -show-unimportant
1 + 2 * (3 + 4)
1 + 2 * (3 + 4)
1 + 2 * (3 + 4)
1 + 2 * (3 + 4)
1 + 2 * (3 + 4)
1 + 2 * 7
1 + 14
15

$ ./arith -e "1 + 2 * (3 + 4)"
1 + 2 * (3 + 4)
1 + 2 * 7
1 + 14
15

$ ./arith -e "1 + 2 * (3 + 4)" -debug
IOp +; IOp *; IOp + || 4; 3; 2; 1
1 + 2 * (3 + 4)
IOp *; IOp + || 7; 2; 1
1 + 2 * 7
IOp + || 14; 1
1 + 14
15
```

Speed

Calculating $1 + 2 * (3 + 4)$ one million times, this bytecode interpreter is 64x faster than our syntactic interpreter when not printing anything out, and 5x faster when printing out all the steps.

Next Steps

Note that, because each step reduces the size of the expression (and does not generate anything new), 2) and 3) in the first paragraph above are the same thing. Soon, they will not be, and we will need to find some way to keep enough information around.