UCSD CSE131 F19 – Substitution, Variables, Single-Stepping, and Functions

October 22, 2019

Describing Evaluation

One of the things we often need to do is describe how a programming language feature ought to work without writing out precisely how it compiles. This is important, because we may want to implement the language in multiple ways, or for multiple processor types, and need to specify what the language should do independent of a particular platform.

We've worked with a few different strategies for doing this, but two major ones. First, we've appealed to step-by-step algebraic substituion by example, and second, we've written out descriptions in precise English. For example, in Copperhead, we describe `set` expressions with:

> `set` expressions update the value of a variable. The behavior of a `set` expression is to evaluate its subexpression, then set the value of the named variable to the result of that subexpression. The result of the entire `set` expression is the new value, and its effect is to make future accesses of that variable get the updated value.

There's a lot going on there, and some of it is subtle. In fact, it's not clear that this description and the algebraic substitution idea are compatible! Let's look at a simple example:

```
(let (x 5) (set x 10) x)
```

In our algebraic model, we might have said this takes a step by replacing occurrences of x with 5:

```
(set 5 10) 5
```

This is nonsense, for two reasons. First, it doesn't make any sense to set the value of 5 to the value of 10 – what does that mean? Second, this program ought to evaluate to 10, but if we just evaluate the sequence of expressions in order, we'd get 5. The step-by-step rule seems to have been incorrect in substituting the value of x when it could be updated by a set before being used. In fact, this substitution idea seems to be at odds with the English description which refers to "future accesses of that variable". If we substitute away all the variable accesses as soon as we define the variable, we can't talk about accessing it at different points in time.

It's possible to reconcile these two views, and there are a few options. One that we're going to use involves leaving some context around during the step-by-step evaluation so we can accurately process updates to variables. The idea we'll use is to keep the `let` bindings surrounding its body as they evaluate step-by-step, and refer back to the set of bindings for access and update. So we could think of the above program as taking these steps:

    `(let (x 5) (set x 10) x)`

We focus on the set expression while leaving the let in place

$\rightarrow$ `(let (x 5)` $\boxed{\texttt{(set x 10)}}$ `x)}`

The set expression changes the immediately-surrounding let's x binding

$\rightarrow$ `(let (x` $\boxed{\texttt{10}}$`}) 10 x)}`

The variable expression looks up the x binding in the immediately-surrounding let

$\rightarrow$ `(let (x` $\boxed{\texttt{10}}$`}) 10` $\boxed{\texttt{10}}$`)}`

The body evaluates to the last expression's value

$\rightarrow$ `(let (x 10}) 10)`

When a let expression has just one value in its body, that's the result of the whole expression, and we're done with this x

$\rightarrow$ `10`

This is an interpretation that uses some context—the let expression's bindings—to track information about variables' values during evaluation. This is a way to write down in text what you may have written down in other contexts as a stack frame with a box mapping variables to their values. This idea composes well with some related ideas for loops. For example, consider this program, that sums the numbers from 1 to 9:

```
    (let (x 1)
      (let (sum 0)
        (while (< x 10)
          (set sum (+ sum x))
          (set x (+ x 1)))
        sum))
```

The idea here is that a while loop steps to an if that either runs one iteration and then loops, or evaluates to false

```
→ (let (x 1)
      (let (sum 0)
        (if (< x 10)
            (
              (set sum (+ sum x))
              (set x (+ x 1))
              (while (< x 10)
                (set sum (+ sum x))
                (set x (+ x 1))
              )
            )
            false)
        sum))
```

Look up current value of x and compare

```
→ (let (x 1)
      (let (sum 0)
        (if \boxed{true}
            (
              (set sum (+ sum x))
              (set x (+ x 1))
              (while (< x 10)
                (set sum (+ sum x))
                (set x (+ x 1))
              )
            )
            false)
        sum))
```

Reduce if to its then case

```
→ (let (x 1)
      (let (sum 0)
        (set sum (+ sum x))
        (set x (+ x 1))
        (while (< x 10)
          (set sum (+ sum x))
          (set x (+ x 1))
        )
        sum))
```

Look up sum and x

```
→ (let (x 1)
      (let (sum 0)
        (set sum (+ \boxed{0} \boxed{1}))
        (set x (+ x 1))
        (while (< x 10)
          (set sum (+ sum x))
          (set x (+ x 1))
        )
        sum))
```

Update the sum variable

```
→ (let (x 1)
    (let (sum \boxed{1})
      \boxed{1}
      (set x (+ x 1))
      (while (< x 10)
        (set sum (+ sum x))
        (set x (+ x 1))
      )
      sum))
```
Look up then update x
```
→ (let (x \boxed{2})
    (let (sum 1)
      1
      \boxed{2}
      (while (< x 10)
        (set sum (+ sum x))
        (set x (+ x 1))
      )
      sum))
```
Process the while loop again
```
→ (let (x 2)
    (let (sum 1)
      1
      2
      (if (< x 10)
          (
            (set sum (+ sum x))
            (set x (+ x 1))
            (while (< x 10)
              (set sum (+ sum x))
              (set x (+ x 1))
            )
          )
          false)
      sum))
```
and so on. The final state will look something like this:
```
→ (let (x 10)
    (let (sum 45)
      1 2 3 3 6 4
      10 5 15 6 21 7
      28 8 36 9 45 10
      false
      45))
```

It's helpful to be aware that we can write programs out with variable update in this way, because it means we don't have to throw out all the work we did understanding substitution. We simply need to enhance our model to accommodate using let as a binding context that stores the current value of a variable.

Of course, this is still just one way to describe evaluation. The English descriptions are still valuable, and sketching examples with pictures of stack frames is also valuable, and talking about the specific generated instructions and memory layout is also valuable! We will continue to use whichever one is appropriate and helpful for understanding a particular topic, with the knowledge that we can shift between them as necessary.

This also harmonizes our understanding of programs that rely on loops and variable update with the algebraic stepping approach. This is useful because it gives us some chances to see a more unified framework that explains both the function-and-recursion focused style of OCaml and Haskell as well as the loop-and-variable style of C, Python, and Java.

Describing Function Evaluation

Many languages have functions, and the reasons for having them are several (we could list more beyond these):

1. Encapsulating a computation into a limited scope and span of code
2. Avoiding code repetition
3. Implementing recursive algorithms
4. Describing an operation that should happen at a future time (when the function is applied), potentially multiple times

The essence of functions is twofold:

1. The separation of a definition of an expression from its use
2. The parameterization of an expression with names whose values are chosen when the function is applied

This motivates a design for functions that has two pieces of syntax – one for defining functions and one for using them.[1]

$$\begin{array}{lcl} e & := & \ldots\text{other expressions}\ldots \mid \texttt{(}f\ \ e\texttt{)} \\ d & := & \texttt{(def (}f\ \ x\texttt{)}\ \ e\texttt{)} \end{array}$$

Or, in Ocaml:

```
type expr =
  | EApp of string * expr

type def =
  | DFun of string * string * expr

type prog = def list * expr
```

Note that these functions have only a *single* argument. The descriptions here do generalize to multi-argument functions with some work.

So a function definition for the absolute value function is:

```
(def (abs x)
  (if (< x 0) (* -1 x) x))
```

and an application of it is:

```
(abs 9)
```

It's relatively straightforward to agree that it should evaluate to `9`, and that `(abs -3)` should evaluate to `3`. We could describe this in a few ways. One is to use an English sentence like:

> A function application evaluates by first evaluating its argument, then finding the definition that matches the given function name, and evaluating the definition's body with the parameter name set to the value of the argument. The result of the body's evaluation is the result of the entire function application.

It's also interesting to think about how to define it in terms of algebraic stepping. There are a few ways we could do this, but one that uses some of what we discussed in the last section is helpful here. A function application works by stepping to a `let` expression:

[1]We omit types in this description for simplicity, since the types by design don't affect how the program *runs*!

```
   (abs 9)
→ (let (x 9) (if (< x 0) (* -1 x) x))
→ (let (x 9) (if false (* -1 x) x))
→ (let (x 9) x)
→ 9
```

That is, when we have a function application like (f e_a), and a definition with a matching name (def f (x:τ_1):τ_2 e_b), it evaluates by taking a step to (let (x e_a) e_b).

This is a helpful description because it naturally generalizes to functions that call other functions, including recursive ones, and including functions that might use `set` on their arguments.

```
(def (sum x)
  (if (< x 1) 0 (+ x (sum (- x 1)))))

   (sum 3)
→ (let (x 3) (if (< x 1) 0 (+ x (sum (- x 1)))))
→ (let (x 3) (if false 0 (+ x (sum (- x 1)))))
→ (let (x 3) (+ x (sum (- x 1))))
→ (let (x 3) (+ 3 (sum 2)))
→ (let (x 3) (+ 3 (let (x 2) (if (< x 1) 0 (+ x (sum (- x 1)))))))
→ (let (x 3) (+ 3 (let (x 2) (if false 0 (+ x (sum (- x 1)))))))
→ (let (x 3) (+ 3 (let (x 2) (+ x (sum (- x 1))))))
→ (let (x 3) (+ 3 (let (x 2) (+ 2 (let (x 1) (if (< x 1) 0 (+ x (sum (- x 1)))))))))
→ ...
→ (let (x 3) (+ 3 (let (x 2) (+ 2 (let (x 1) (+ 1 (let (x 0) 0)))))))
→ ...
→ 6
```

Note that in this example, it's crucial that variable lookup finds the instance of x that is the closest to the use – there is space for one x created per function call with the `let`. This matches the behavior of `let` that we've used since the beginning of the course.

This description also accounts for a different version of sum that uses a loop:

```
(def (sum x)
  (let (sum 0)
    (while (> x 0)
      (set sum (+ sum x))
      (set x (- x 1)))
    sum))

(sum 3)
→
(let (x 3)
  (let (sum 0)
    (while (> x 0)
      (set sum (+ sum x))
      (set x (- x 1)))
    sum))
→ ...
```

This continues with a similar pattern to the earlier sum example.

Compiling Function Evaluation

Of course, our eventual goal in this course is to turn source programs into executables by way of x86-64 assembly. The above discussion will help us in several ways.

First, we assume a few extra steps as setup – first we parse all the definitions into a list of `def` along with a final expression as an `expr`. We compile the final expression which is the entry point for the program, and pass the definitions list to `e_to_is` so we can use definition information during compilation.

```
type prog = def list * expr
let parse_def (sexp : Sexp.t) : def = ... parse ...
let parse_program (sexps : Sexp.t list) : prog = ... parse ...

let rec e_to_is (e : expr) (si : int) (env : tenv) (defs : def list) : string list =
... other cases as before ...
| EApp(f, arg) ->

let compile (program : Sexp.t list) : string =
  let (defs, body) = parse_program program in
  let instrs = e_to_is body 1 [] defs in
```

A First Try

The step-by-step rule we gave above suggests a really simple first cut at compiling functions – just use let! So we could try something like this:

```
let rec e_to_is (e : expr) (si : int) (env : tenv) (defs : def list) : string list =
... other cases as before ...
| EApp(f, arg) ->
  match find_def f def_env with
    | None -> failwith "No such function definition"
    | Some(DFun(name, arg_name, arg_typ, ret_typ, body)) ->
      compile_expr (ELet(arg_name, arg, body))
```

We can try it on a simple program as input:

```
(def (f x)
  (+ x 2))
(f 10)
```

If we compile and run this, we get the right answer!

Other examples work, as well, including calling the function twice, as in `(+ (f 10) (f 3))`, or using `(f (f 7))` as the main expression.

Even more complex examples work, like calling a function from within another function:

```
(def (g y)
  (+ 5 y))
(def (f x)
  (+ x (g x)))
(f 42)
```

Awesome! This will extend well to functions that contain `while` loops and variable assignment. Clearly, we are done and can ship the compiler to our users, confident that they will now be able to write and use functions!

Wait a minute. This is weird.

We “implemented” functions, but somehow managed to avoid ever using the `call` and `ret` instructions that seem purposefully built for function call and return.

We “implemented” functions, but never talked about the stack pointer or where the code for a function goes.

This violates a lot of intuition we probably have from looking at compiled C programs.

Let’s try just one more function:

```
(def (sum n)
  (if (< n 1) 0 (+ n (sum (+ n -1)))))
(sum 3)
```

```
> make sum.run
./compile sum.int > sum.s
Fatal error: exception Stack overflow
make: *** [sum.s] Error 2
rm sum.s
```

That’s a stack overflow *in the compiler* – the assembly didn’t even get generated. When we compile the `EApp("sum", ENum(3))` expression, we look up the definition of `sum`, which has a use of `sum` in its body. Then we recursively call `e_to_is` with a new let expression containing that call. When that use of `sum` is visited by the compiler, we generate a let with the body of `sum` again, and the cycle repeats. Eventually we reach Ocaml’s maximum stack depth and the compiler crashes.

Since in general we can’t know how many recursive calls will happen, especially in a case like `sum` where there is an `if` separating the base case and the recursive case, we need to do something more sophisticated. It’s worth pointing out that for functions that aren’t recursive (or never call a recursive function) the strategy of copying the expression actually produces working programs! It’s an optimization strategy called **inlining** that trades larger generated code for decreasing the number of function calls that happen.

However, for recursive calls, we have to do something different, and this motivates compiling definitions separate from calls, and the idea of a **calling convention**. We still want to get the kind of behavior we saw with let above, but have to approach the implementation differently.

A simple such **calling convention** is implemented in `https://ucsd-cse131-f19.github.io/lectures/10-22-lec8/compile.ml` and demonstrated in `https://ucsd-cse131-f19.github.io/lectures/10-22-lec8/calling.pdf`.