CS4613 Lecture 6: Substitution and Functions

David Bremner

March 21, 2023

Functions & First Class Function Values

- PLAI "Evaluating Functions"
- The concept of a function is itself very close to substitution, and to our 'let1' form.

- Consider the following "morph"

```
{let1 {x 5}
  {* x x}}

{let1 x  ; potential syntax
  {* x x}} ;for anonymous function

{lam x
  {* x x}}
```

Functions & First Class Function Values
- PLAI "Evaluating Functions"
- The concept of a function is itself very close to substitution, and to our 'let1' form.
- Consider the following "morph"

```
{let1 {x 5}
  {* x x}}

{let1 x  ; potential syntax
  {* x x}} ;for anonymous function

{lam x
  {* x x}}
```

1. Now that we have a form for local bindings, which forced us to deal with proper substitutions and everything that is related, we can get to functions.

Functions and binding

We need a form to use these functions. We want

```
{{lam x {* x x}}
      5}
```

to be the same as the original thing we started with

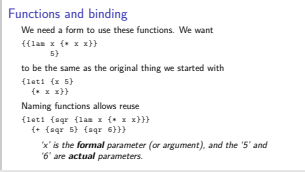
```
{let1 {x 5}
  {* x x}}
```

Naming functions allows reuse

```
{let1 {sqr {lam x {* x x}}}
  {+ {sqr 5} {sqr 6}}}
```

*'x' is the **formal** parameter (or argument), and the '5' and '6' are **actual** parameters.*

1. Note that this way of binding functions depends strongly on our functions being values.



First class functions

Three basic approaches to functions/procedures

First order functions are not values. E.g. Java methods.

Higher order functions can receive and return other functions as values. This is what you get in C.

First class can be stored in data structures. Can be defined where / when other values can be, using values in current scope.

Functions as expressions

In machine-code, to compute an expression such as

```
(-b + sqrt(b^2 - 4*a*c)) / 2a
```

You have to do something like this:

```
x = b * b
y = 4 * a
⋮
s = x / y
```

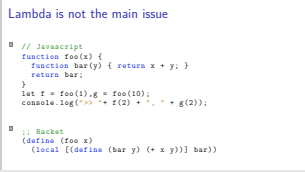
With first-class functions, complex expressions can have functions as intermediate values.

```
(map (λ (x) (+ x 3))
     (list 1 2 3))
```

Lambda is not the main issue

8
```
// Javascript
function foo(x) {
  function bar(y) { return x + y; }
  return bar;
}
let f = foo(1),g = foo(10);
console.log(">> "+ f(2) + ", " + g(2));
```

9
```
;; Racket
(define (foo x)
  (local [(define (bar y) (+ x y))] bar))
```



In both cases the returned function is not anonymous, but it's not really named either. the 'bar' name is bound only inside the body of 'foo', and outside of it that name is irrelevant.

C local functions

GCC allows local function definitions as an extension, but they are not first class. Why not?

```
10 typedef int(*int2int)(int);
   int2int foo(int x) {
     int bar(int y) { return x + y; }
     return bar;
   }
   int main() {
     int2int f = foo(1);
     int2int g = foo(10);
     printf(">> %d, %d\n", f(2), g(2));
   }
```

The FLANG Language

Concrete Syntax

```
flang: NUMBER
           | { "+" flang flang }
           | { "-" flang flang }
           | { "*" flang flang }
           | { "/" flang flang }
           | { "let1" { ID flang } flang }
           | ID
           | { "lam"  ID flang }
           | { flang flang } ;; Call
```

FLANG Abstract Syntax

12
```
(define-type FLANG
  [Num  (val : Number)]
  [Add  (l : FLANG) (r : FLANG)]
  [Sub  (l : FLANG) (r : FLANG)]
  [Mul  (l : FLANG) (r : FLANG)]
  [Div  (l : FLANG) (r : FLANG)]
  [Id   (name : Symbol)]
  [Let1 (id : Symbol)
        (named-expr : FLANG)
        (bound-body : FLANG)]
  [Lam  (param : Symbol)
        (body : FLANG)]
  [Call (lam : FLANG)
        (val : FLANG)]) ; first type!
```

FLANG parser

Parsing `Lam` requires an identifier in position 2

```
[(s-exp-match? `(lam SYMBOL ANY) sx)
 (let* ([id (s-exp->symbol (sx-ref sx 1))]
        [body (parse-sx (sx-ref sx 2))])
   (Lam id body))]
```

We also need a surface syntax for calling/applying a function. We follow the book (and Racket) and just make that the default:

```
[(s-exp-match? `(ANY ANY) sx)
 (Call (parse-sx (sx-ref sx 0))
       (parse-sx (sx-ref sx 1)))]
```

FLANG substitution I/II

Mostly substitution is the same:

```
N[v/x]                    = N
{+ E1 E2}[v/x]            = {+ E1[v/x] E2[v/x]}
;;   -, *, /...
y[v/x]                    = y
x[v/x]                    = v
{let1 {y E1} E2}[v/x]     = {let1 {y E1[v/x]} E2[v/x]}
{let1 {x E1} E2}[v/x]     = {let1 {x E1[v/x]} E2}
```

FLANG Substitution, the new parts

`call` looks like arithmetic

```
{E1 E2}[v/x]      = {E1[v/x] E2[v/x]}
```

`lam` looks like `let1`.

```
{lam y E}[v/x]       = {lam y E[v/x]}
{lam x E}[v/x]       = {lam x E}
```

New substitution cases in Racket

```
(type-case FLANG expr
                ⋮
  [(Call l r)
     (Call (subst l from to) (subst r from to))]
  [(Lam bound-id bound-body)
   (if (eq? bound-id from)
     expr
     (Lam bound-id (subst bound-body from to)))]))
```

Representing values in FLANG

- we need to decide on how to represent values in FLANG.
- Before, we had only numbers and we used (Racket) numbers to represent them.
- What should be the result of evaluating

```
{lam x {+ x 1}}
```

- We need some way, e.g. type variants to distinguish between functions and numbers.
- In fact we already have a type which would work, namely FLANG
- We could also define a result type, which might be a bit cleaner, but the main issues are the same.

Using FLANG to represent values

Numbers are wrapped

```
(test (eval (Add (Num 1) (Num 2))) (Num 3))
```

and evaluate to themselves

```
(test (interp (Num 5)) (Num 5))
```

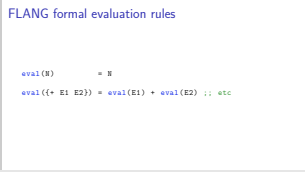
as do functions

```
(interp (test (Lam 'x (Num 2))
              (Lam 'x (Num 2))))
```

FLANG formal evaluation rules

```
eval(N)          = N

eval({+ E1 E2}) = eval(E1) + eval(E2) ;; etc
```



1. 'call' will be very similar to 'let1' – the only difference is that its arguments are ordered a little differently, being retrieved from the function that is applied and the argument.

FLANG formal evaluation rules (part 2)

The formal evaluation rules treat functions like numbers, and use the syntax object to represent both values:

```
eval(id)          = error!

eval({let1 {x E1} E2}) = eval(E2[eval(E1)/x])

eval(LAM)         = LAM ; assuming LAM is a function expr

eval({E1 E2}) = if eval(E1) == {lam x Ef} then
                        eval(Ef[eval(E2)/x])
                else error!
```

Call and Let1

Note that the `call` rule could be written using a translation to a 'let1' expression:

```
eval({E1 E2})
                  = if eval(E1) = {lam x Ef} then
                        eval({let1 {x E2} Ef})
                    else
                        error!
```

Symmetrically, we could specify 'let1' using 'call' and 'fun':

```
eval({let1 {x E1} E2}) =
eval({{lam x E2} E1})
```

Dynamic typing

Need to check arithmetic operation's arguments:

```
eval({+ E1 E2}) = eval(E1) + eval(E2)
                    if eval(E1) and eval(E2) are
                       numbers
                    otherwise error!
```

28

```
(define (arith-op op expr1 expr2)
  (let ([unwrap
         (λ (e)
           (type-case FLANG e
             [(Num n) n]
             [else (error 'arith-op "NaN")]))])
    (Num (op (unwrap expr1)
             (unwrap expr2)))))
```

Flang evaluator

```
[(Num n) expr]  ; <- change here
      ⋮
[(Let1 bound-id named-expr bound-body)
 (interp (subst bound-body bound-id
                (interp named-expr)))] ; <- no `(Num ...)'
[(Lam bound-id bound-body) expr] ;cf. Num

[(Call lam arg-expr)
 (type-case FLANG lam
   [(Lam bound-id bound-body) ;cf. Let1
    (interp (subst bound-body bound-id
                   (interp arg-expr)))]
   [else (error 'eval "non-function")])]))
```

Returning numbers

We can also make things a little easier to use if we make ‘run’ convert the result to a number:

```
(define (run sx)
  (let ([result (interp (parse-sx sx))])
    (type-case FLANG result
      [(Num n) n]
      [else (error 'run "returned a non-number")])))
```

It's alive?

30

```
(test (run `{{lam x {+ x 1}} 4})
      5)
(test (run `{let1 {add3 {lam x {+ x 3}}}
             {add3 1}})
      4)
(test (run `{let1 {add3 {lam x {+ x 3}}}
             {let1 {add1 {lam x {+ x 1}}}
               {let1 {x 3}
                 {add1 {add3 x}}}}})
      7)
```

Oops, I an evaluation.

31

```
(test (run `{let1 {identity {lam x x}}
                  {let1 {foo {lam x {+ x 1}}}
                        {{identity foo} 123}}})
      124)
(test (run `{{{lam x {x 1}}
                {lam x {lam y {+ x y}}}} 123})
      124)
```

Fixing call

It seems like our `call` implementation needs work

32

```
(trace interp)
(run `{let1 {identity {lam x x}}
        {let1 {foo {lam x {+ x 1}}}
          {{identity foo} 123}}})
(run `{{{lam x {x 1}}
                  {lam x {lam y {+ x y}}}}
            123})
```

So we have to reduce the expression in the function position ***before*** *we can tell if it is a function.*

```
[(Call lam arg-expr)
      (let [(funV (interp lam))]
        (type-case FLANG funV
          [(Lam bound-id bound-body)
               ; just like `let1'
               (interp (subst bound-body
                              bound-id
                              (interp arg-expr)))]
          [else (error 'eval "expected function")]))]))
```

Now our tests pass

34

```
(trace interp)
(test (run `{let1 {identity {lam x x}}
              {let1 {foo {lam x {+ x 1}}}
                {{identity foo} 123}}})
      124)

(test (run
       `{{{lam x {x 1}} {lam x {lam y {+ x y}}}} 123})
      124)
```