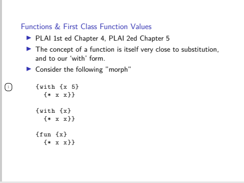
Functions & First Class Function Values

- PLAI 1st ed Chapter 4, PLAI 2ed Chapter 5
- The concept of a function is itself very close to substitution, and to our ‘with’ form.
- Consider the following ”morph”

```
{with {x 5}
  {* x x}}

{with {x}
  {* x x}}

{fun {x}
  {* x x}}
```

2019-02-02



Functions & First Class Function Values

- PLAI 1st ed Chapter 4, PLAI 2ed Chapter 5
- The concept of a function is itself very close to substitution, and to our ‘with’ form.
- Consider the following ”morph”

```
{with {x 5}
  {* x x}}

{with {x}
  {* x x}}

{fun {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.

- As with the ‘with’ expression, we also need a form to use these functions. We can use ‘call’ for this, so we want:

2
```
{call {fun {x} {* x x}}
      5}
```

to be the same as the original thing we started with

3
```
{with {x 5}
  {* x x}}
```

- So far, all we get is more verbose local bindings
- What we're really missing is a way to name these functions.

```
{with {sqr {fun {x} {* x x}}}
  {+ {call sqr 5}
     {call sqr 6}}}
```

- Here we say that ‘x’ is the **formal** parameter (or argument), and the ‘5’ and ‘6’ are **actual** parameters.

2019-02-02

- So far, all we get is more verbose local bindings
- What we're really missing is a way to name these functions.

4

```
{with {sqr {fun {x} {* x x}}}
  {+ {call sqr 5}
     {call sqr 6}}}
```

- Here we say that '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.

Implementing First Class Function Values

- PLAI Chapter 6 (uses some stuff from ch. 5, which we do later)

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: functions are values with all the rights of other values.

2019-02-02

Implementing First Class Function Values

- PLAI Chapter 6 (uses some stuff from ch. 5, which we do later)

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: functions are values with all the rights of other values.

1. They cannot be used or returned as values by other functions. This means that they cannot be stored in data values.
2. In particular, they can be supplied to other functions, returned from functions, stored in data structures, and new functions can be created at run-time.

- In machine-code, to compute an expression such as

5
```
(-b + sqrt(b^2 - 4*a*c)) / 2a
```

- you have to do something like this:

6
```
x = b * b
y = 4 * a
y = y * c
x = x - y
x = sqrt(x)
y = -b
x = y + x
y = 2 * a
s = x / y
```

- In "high-level" programming languages you can just write the original expression, with no names for these values.
- With first-class functions, complex expressions can have functions as intermediate values.

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

- **fun** generates a function, and we can choose to either bind it to a name, or not.
- This has a major effect on the "personality" and expressive power of a programming language as we will see.

Lambda is not the main issue

The following is working JavaScript code, that uses first class functions.

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

- Equivalent Racket code, no **explicit** `lambda`

9

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

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.

GCC includes extensions that allow internal function definitions, but it still does not have first class functions – trying to do the above is broken:

10

```
#include <stdio.h>
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

Now for the implementation – we call this new language FLANG. First, the BNF:

11
```
flang: NUMBER
         | { + flang flang }
         | { - flang flang }
         | { * flang flang }
         | { / flang flang }
         | { WITH { ID flang } flang }
         | ID
         | { FUN { ID } flang }
         | { CALL flang flang }
```

And the matching type definition:

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)]
  [With (id : Symbol) (named-expr : FLANG)
     (bound-body : FLANG)]
  [Fun  (param : Symbol) (body : FLANG)]
  [Call (fun : FLANG) (val : FLANG)]) ; first type!
```

The parser is pretty much cut-paste-and-munge from WAE

```
        ⋮
[(s-exp-match? `(fun (SYMBOL) ANY) sx)
   (let* ([args (sx-ref sx 1)]
          [id (s-exp->symbol (sx-ref args 0))]
          [body (parse-sx (sx-ref sx 2))])
    (Fun id body))]
[(s-exp-list? sx)
  (let* ([l (λ () (parse-sx (sx-ref sx 1)))]
         [r (λ () (parse-sx (sx-ref sx 2)))])
    (case (s-exp->symbol (sx-ref sx 0))
                ⋮
        [(call) (Call (l) (r))]
```

We also need to patch up the substitution function to deal with `fun` and `call`. Mostly it 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

{with {y E1} E2}[v/x] = {with {y E1[v/x]}
                              E2[v/x]}

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

The new part: call looks like arithmetic, fun looks like with.

15
```
{call E1 E2}[v/x]      = {call E1[v/x] E2[v/x]}
```

16
```
{fun {y} E}[v/x]       = {fun {y} E[v/x]}
{fun {x} E}[v/x]       = {fun {x} E}
```

And the matching code:

17

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

- 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

18
```
{fun {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.

- Use FLANG to represent values means that evaluating:

  ```
  (Add (Num 1) (Num 2))
  ```

  now yields

  ```
  (Num 3)
  ```

  and "(Num 5)" evaluates to "(Num 5)".
- In a similar way,

  ```
  (Fun 'x (Num 2))
  ```

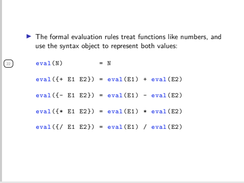
  ```
  evaluates to
  ```

  ```
  (Fun 'x (Num 2))
  ```

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

```
eval(N)          = N

eval({+ E1 E2}) = eval(E1) + eval(E2)

eval({- E1 E2}) = eval(E1) - eval(E2)

eval({* E1 E2}) = eval(E1) * eval(E2)

eval({/ E1 E2}) = eval(E1) / eval(E2)
```

2019-02-02



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

```
eval(N)         = N
eval({+ E1 E2}) = eval(E1) + eval(E2)
eval({- E1 E2}) = eval(E1) - eval(E2)
eval({* E1 E2}) = eval(E1) * eval(E2)
eval({/ E1 E2}) = eval(E1) / eval(E2)
```

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

```
eval(id)          = error!

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

eval(FUN)         = FUN ; assuming FUN
                        ; is a function expression

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

- Note that the `call` rule could be written using a translation to a ‘with’ expression:

```
24    eval({call E1 E2})
                  = if eval(E1) = {fun {x} Ef}
                     then
                       eval({with {x E2} Ef})
                    else
                       error!
```

- Symmetrically, we could specify ‘with’ using ‘call’ and ‘fun’:

```
25    eval({with {x E1} E2}) = eval({call
                                     {fun {x} E2}
                                     E1})
```

- we now have two kinds of values, so we need to check the arithmetic operation's arguments too:

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

The FLANG evaluator looks like:

```
[(Num n) expr]  ; <- change here
      :
[(With bound-id named-expr bound-body)
 (eval (subst bound-body bound-id
             (eval named-expr)))] ; <- no `(Num
                 ...)'
      :
[(Fun bound-id bound-body) expr]  ; <- like `Num'
[(Call fun arg-expr)
      (type-case FLANG fun
         [(Fun bound-id bound-body) ; like `With`
              (eval (subst bound-body bound-id
                         (eval arg-expr)))]
         [else (error 'eval "not a function")])]))
```

The ‘arith-op’ function is in charge of

- checking that the input values are FLANG numbers,
- translating them to plain numbers,
- performing the Racket operation, then re-wrapping the result in a ‘Num’.

```
(define (arith-op op expr1 expr2)
  (local
      [(define (Num->number e)
         (type-case FLANG e
           [(Num n) n]
           [else (error 'arith-op "expects a
               number")]))]
    (Num (op (Num->number expr1)
             (Num->number expr2)))))
```

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

29

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

It’s alive?

```
(test (run `{call {fun {x} {+ x 1}} 4})
      5)
(test (run `{with {add3 {fun {x} {+ x 3}}}
             {call add3 1}})
      4)
(test (run `{with {add3 {fun {x} {+ x 3}}}
             {with {add1 {fun {x} {+ x 1}}}
               {with {x 3}
                 {call add1 {call add3 x}}}}})
      7)
```

Oops, I an evaluation.

```
(test (run `{with {identity {fun {x} x}}
                {with {foo {fun {x} {+ x 1}}}
                      {call {call identity foo}
                         123}}})
    124)
(test (run `{call
           {call {fun {x} {call x 1}}
                 {fun {x} {fun {y} {+ x y}}}}
           123}
         )
    124)
```

Fixing call

- We just worked through an implementation of first class functions in WAE. But it had a few failing tests.
- It seems like our `call` implementation needs work

```
(run `{with {identity {fun {x} x}}
        {with {foo {fun {x} {+ x 1}}}
          {call {call identity foo} 123}}})
(run `{call {call {fun {x} {call x 1}}
                  {fun {x} {fun {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 fun arg-expr)
      (let [(funV (eval fun))]
        (type-case FLANG funV
          [(Fun bound-id bound-body)
               ; just like `with'
               (eval (subst bound-body
                            bound-id
                            (eval arg-expr)))]
          [else (error 'eval "expected
             function")]))])
```

Now our tests pass

```
(trace eval)
(test (run `{with {identity {fun {x} x}}
              {with {foo {fun {x} {+ x 1}}}
                {call {call identity foo} 123}}})
      124)
(test (run `{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x
                           y}}}}
                  123})
      124)
```