Lazy Evaluation of Function Arguments

```
let f = proc(x)0
 in (f +(1,+(2,+(3,+(4,+(5,6))))))
```

The computed 21 is never used.

What if we were *lazy* about computing function arguments (in case they aren't used)?

Lazy Evaluation of Function Arguments

One way to laziness:

```
let f = proc(xthunk)0
 in (f proc()+(1,+(2,+(3,+(4,+(5,6))))))
```

```
let f = proc(xthunk)-((xthunk), 7)
 in (f proc()+(1,+(2,+(3,+(4,+(5,6))))))
```

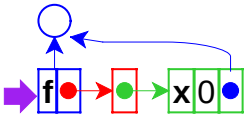
By using **proc** to delay evaluation, we can avoid unnecessary computation.

How about making the language compute function arguments lazily in *all* applications?

Evaluation with Lazy Arguments

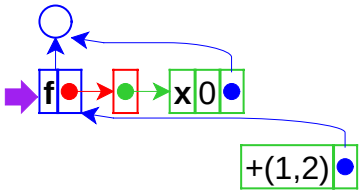
```
let f = proc(x)0
 in (f +(1,2))
```

Evaluation with Lazy Arguments




```
let f = proc(x)0
 in (f +(1,2))
```

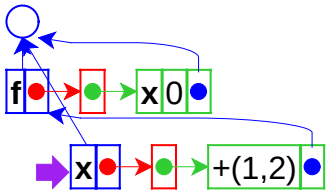
Evaluation with Lazy Arguments



Application creates a new kind of green box, with two slots: a ***thunk***

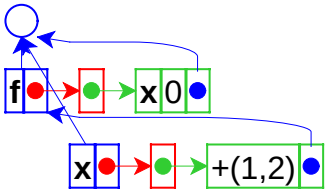
let f = **proc**(**x**)0
in (**f** +(1,2))

Evaluation with Lazy Arguments



let f = **proc**(**x**)0
in (**f** +(1,2))

Evaluation with Lazy Arguments



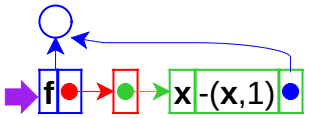
The result is 0

let f = **proc**(**x**)0
in (**f** +(1,2))

Evaluation with Lazy Arguments

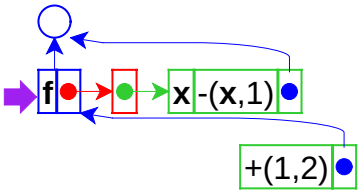
let f = **proc**(**x**)-(**x**,1)
in (**f** +(1,2))

Evaluation with Lazy Arguments



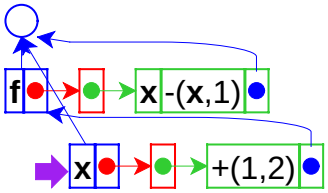
let f = **proc**(**x**)-(**x**,1)
in (**f** +(1,2))

Evaluation with Lazy Arguments



let f = **proc**(**x**)-(**x**,1)
in (**f** +(1,2))

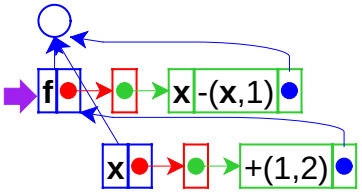
Evaluation with Lazy Arguments



lookup of **x**...

let f = **proc**(**x**)-(x,1)
in (**f** +(1,2))

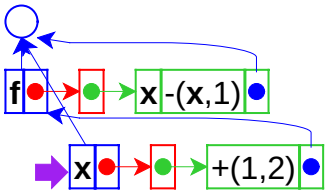
Evaluation with Lazy Arguments



... forces evaluation of the thunk

let f = **proc**(**x**)-(**x**,1)
in (**f** +(1,2))

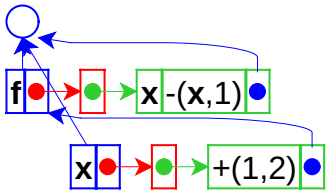
Evaluation with Lazy Arguments



so 3 is the value of **x**

```
let f = proc(x)-(x,1)
  in (f +(1,2))
```

Evaluation with Lazy Arguments



The result is 2

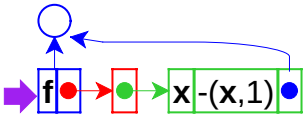
```
let f = proc(x)-(x,1)
  in (f +(1,2))
```

Evaluation with Lazy Arguments

Lazy expression that needs its environment...

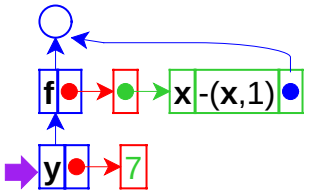
```
let f = proc(x)-(x,1)
  in let y = 7
     in (f +(1,y))
```

Evaluation with Lazy Arguments



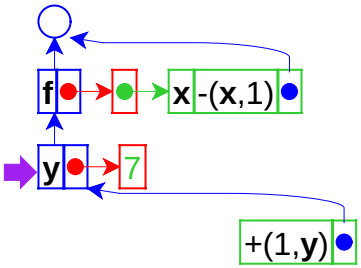

```
let f = proc(x)-(x,1)
  in let y = 7
     in (f +(1,y))
```

Evaluation with Lazy Arguments



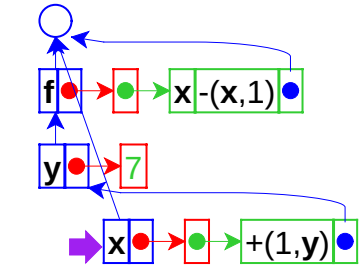

```
let f = proc(x)-(x,1)
  in let y = 7
     in (f +(1,y))
```

Evaluation with Lazy Arguments




```
let f = proc(x)-(x,1)
  in let y = 7
     in (f +(1,y))
```

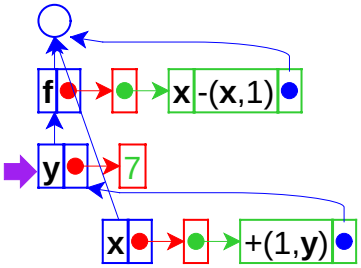
Evaluation with Lazy Arguments



Evaluation of **x** forces the thunk...

```
let f = proc(x)-(x,1)
  in let y = 7
     in (f +(1,y))
```

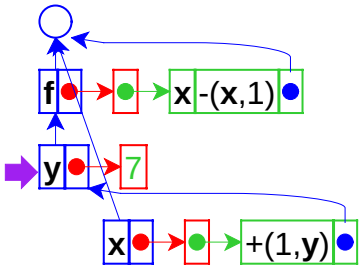
Evaluation with Lazy Arguments



Triggering evaluation with the *thunk*'s enviornment, not the current one

```
let f = proc(x)-(x,1)
  in let y = 7
     in (f +(1,y))
```

Evaluation with Lazy Arguments



(The result will be 7)

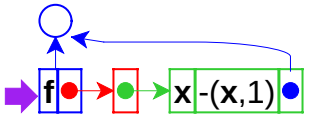
```
let f = proc(x)-(x,1)
  in let y = 7
     in (f +(1,y))
```

Evaluation with Lazy Arguments

What if the right-hand side for **y** is an expression, instead of a value?

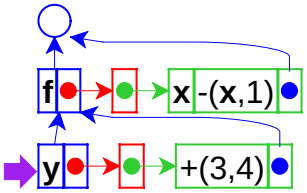
```
let f = proc(x)-(x,1)
  in let y = +(3,4)
     in (f +(1,y))
```

Evaluation with Lazy Arguments




```
let f = proc(x)-(x,1)
  in let y = +(3,4)
     in (f +(1,y))
```

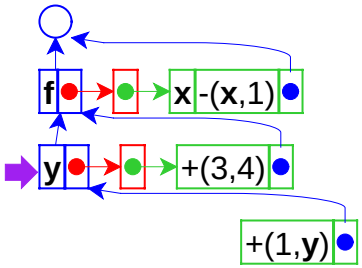
Evaluation with Lazy Arguments



Added thunk for the value of **y**

```
let f = proc(x)-(x,1)
  in let y = +(3,4)
     in (f +(1,y))
```

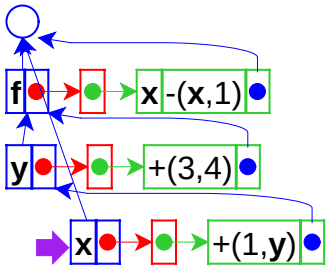
Evaluation with Lazy Arguments



Another thunk for the argument of **f**

```
let f = proc(x)-(x,1)
  in let y = +(3,4)
     in (f +(1,y))
```

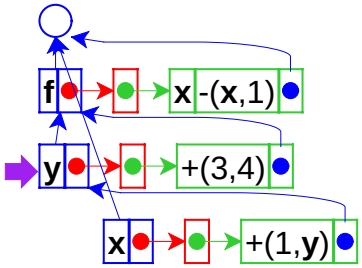
Evaluation with Lazy Arguments



Evaluation of **x** forces a thunk...

```
let f = proc(x)-(x,1)
  in let y = +(3,4)
     in (f +(1,y))
```

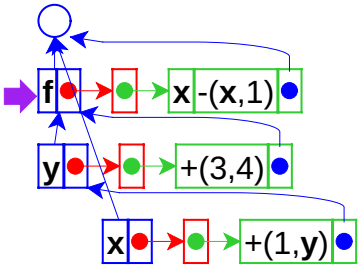
Evaluation with Lazy Arguments



which, in turn, forces another thunk...

```
let f = proc(x)-(x,1)
  in let y = +(3,4)
     in (f +(1,y))
```

Evaluation with Lazy Arguments



and so on (to get 7)

```
let f = proc(x)-(x,1)
  in let y = +(3,4)
     in (f +(1,y))
```

Implementing Lazy Evaluation

Interpreter changes:

- Change **`eval-fun-rands`** to create thunks
- Change variable lookup to force thunk evaluation

(Implement in DrScheme)

Call-by-Name and Call-by-Need

The lazy strategy we just implemented is ***call-by-name***

- Advantage: unneeded arguments are not computed
- Disadvantage: needed arguments may be computed many times

let f = **proc**(**x**)+(**x**,+(**x**,**x**))
in (**f** +(1,+(2,+(3,+(4,+(5,6))))))

Best of both worlds: ***call-by-need***

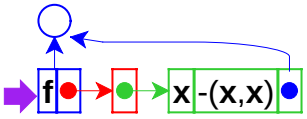
- Evaluates each lazy expression once, then remembers the result

Evaluation with Lazy Arguments

Start as before...

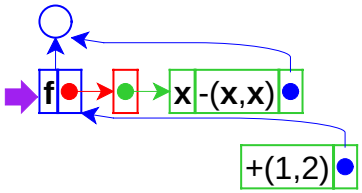
let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

Evaluation with Lazy Arguments



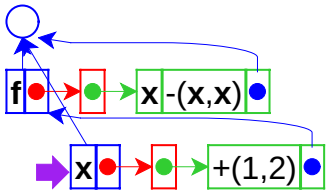
let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

Evaluation with Lazy Arguments



let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

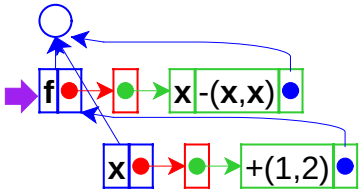
Evaluation with Lazy Arguments



lookup of **x**...

let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

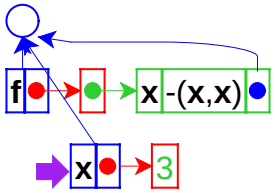
Evaluation with Lazy Arguments



... forces evaluation of the thunk to get 3

let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

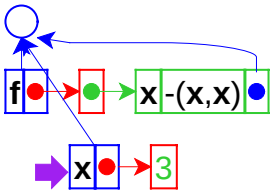
Evaluation with Lazy Arguments



so change **x** to 3 --- which is the essence of call-by-need

let f = **proc**(**x**)-(**x**,**x**)
in (**f** +(1,2))

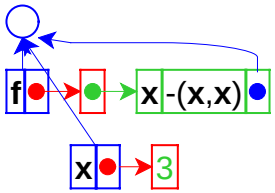
Evaluation with Lazy Arguments



lookup of **x** again gets 3

let f = **proc**(**x**)-(**x**,**x**)
  in (**f** +(1,2))

Evaluation with Lazy Arguments



(The result is 0)

let f = **proc**(**x**)-(**x**,1)
  in (**f** +(1,2))

Implementing Call-by-Need

Interpreter changes:

- Change variable lookup to replace thunks in locations with their values

(Implement in DrScheme)

Calling Convention Terminology

- Call-by-name and call-by-need = ***lazy*** evaluation
- Call-by-value = ***eager*** evaluation

Call-by-reference can augment either

Popular Calling-Convention Choices

- Most languages are call-by-value
  - C, C++, Pascal, Scheme, Java, ML, Smalltalk...
- Some provide call-by-reference
  - C++, Pascal
- A few are call-by-need
  - Haskell
- Practically no languages are call-by-name

Popularity of Laziness

Why don't more languages provide lazy evaluation?

- Disadvantage: evaluation order is not obvious

```
let x = 0  f = ...
 in let y = set x=1
        z = set x=2
    in { (f y z) ; x }
```

Popularity of Laziness

Why do some languages provide lazy evaluation?

- Evaluation order does not matter if the language has no **set** form
- Such languages are called ***purely functional***
  - Note: call-by-reference is meaningless in a purely functional language
- A language with **set** can be called ***imperative***

Laziness and Eagerness

Even in a purely functional language, lazy and eager evaluation can produce different results

```
let f = proc(x)0
 in (f [loop forever])
```

- Eager answer: none
- Lazy answer: 0