Eager and Lazy evaluation.

David Bremner

March 21, 2023

Substitution Review

New ‘let1‘-form (like racket let)

```
lae : NUMBER
        | { + lae lae }
        | { - lae lae }
        | { * lae lae }
        | { / lae lae }
        | { let1 { ID lae } lae }
        | ID
```

Substitution

e[v/i] – To substitute an id. ‘i’ in an expr. ‘e’ with an expr. ‘v’, replace instances of ‘i’ free in ‘e’ with ‘v’.

Substitution Review
New 'let1'-form (like racket let)

```
lae : NUMBER
        | { + lae lae }
        | { - lae lae }
        | { * lae lae }
        | { / lae lae }
        | { let1 { ID lae } lae }
        | ID
```

Substitution
e[v/i] – To substitute an id. 'i' in an expr. 'e' with an expr. 'v', replace instances of 'i' free in 'e' with 'v'.

1. + Avoid writing expressions twice.
   + More expressive language (can express identity).
   + Avoid redundant redundancy.

‘Let1’ evaluation rules

Extended AE evaluation rules:

```
eval(...) = ... same as the AE rules ...
eval({let1 {x E1} E2}) = eval(E2[eval(E1)/x])
eval(ID) = error!
```

One thing we glossed over last time is the type error in this scheme. What’s wrong with this picture:

```
(has-type subst :
          (LAE Symbol LAE -> LAE ))
(has-type eval : (LAE -> Number))
```

We hacked around the type problem by wrapping the results of subst:

```
(define (interp expr)
  (type-case LAE expr
      ⋮
    [(Let1 bound-id named-expr bound-body)
     (interp (subst bound-body
                    bound-id
                    (Num (interp named-expr))))]))
```

Lazy Plait

What should this evaluate to?

lp

```
(let ([x (/ 1 0)])
  42)
```

- we can make it succeed by adding '#:lazy' to our language definition
- Lazy evaluation provides new kinds of modularity via *infinite sequences* or *streams* (related to *generators*).

Lazy Plait

What should this evaluate to?

```
(let ([x (/ 1 0)])
  42)
```

- we can make it succeed by adding '#:lazy' to our language definition
- Lazy evaluation provides new kinds of modularity via *infinite sequences* or *streams* (related to *generators*).

See the paper "Why Functional Programming Matters" by John Hughes.

Using Streams

streams

```
(define ones (stream-cons 1 ones))
(stream->list (stream-take ones 15))

(define (stream-add s1 s2)
  (for/stream ([a s1] [b s2]) (+ a b)))

(define twos (stream-add ones ones))
(stream->list (stream-take twos 15))

(define fibs (stream-cons 1
                (stream-cons 1
                  (stream-add fibs
                              (stream-rest fibs)))))
(stream->list (stream-take fibs 15))
```

Lazy vs Eager Evaluation

Two basic approaches to evaluation

- In lazy evaluation (without sharing), bindings are used for unevaluated code; computation is still duplicated.
- In eager evaluation bindings are used for values

What is our existing evaluator?

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

```
[(Let1 bound-id named-expr bound-body)
 (interp (subst bound-body bound-id
                (Num (interp named-expr)))))]
```

Making evaluation lazy

We can do the substitution purely syntactically:

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

Similarly in the code:

```
(define (interp expr)
  (type-case LAE expr
      ⋮
    [(Let1 bound-id named-expr bound-body)
     ;; no eval, wrapping
     (interp (subst bound-body bound-id named-expr))]
      ⋮ ))
```

Testing our lazy evaluator

11

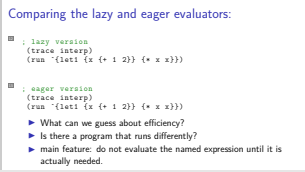
```
(test (run `5) 5)
(test (run `{+ 5 5}) 10)
(test (run `{let1 {x {+ 5 5}} {+ x x}}) 20)
(test (run `{let1 {x 5} {+ x x}}) 10)
(test (run `{let1 {x {+ 5 5}} {let1 {y {- x 3}}
                                    {+ y y}}}) 14)
(test (run `{let1 {x 5} {let1 {y {- x 3}} {+ y y}}}) 4)
(test (run `{let1 {x 5} {+ x {let1 {x 3} 10}}}) 15)
(test (run `{let1 {x 5} {+ x {let1 {x 3} x}}}) 8)
(test (run `{let1 {x 5} {+ x {let1 {y 3} x}}}) 10)
(test (run `{let1 {x 5} {let1 {y x} y}}) 5)
(test (run `{let1 {x 5} {let1 {x x} x}}) 5)
(test/exn (run `{let1 {x 1} y}) "free identifier")
```

Comparing the lazy and eager evaluators:

12
```
; lazy version
 (trace interp)
 (run `{let1 {x {+ 1 2}} {* x x}})
```

13
```
; eager version
 (trace interp)
 (run `{let1 {x {+ 1 2}} {* x x}})
```

- What can we guess about efficiency?
- Is there a program that runs differently?
- main feature: do not evaluate the named expression until it is actually needed.



1. We saw that our simple lazy evaluation strategy can cause duplicate work at runtime. Languages that rely heavily on lazy evaluation will typically introduce some sharing mechanism for unevaluated (or partially evaluated) expressions to avoid this.

Unused bound identifiers

this succeeds

14
```
(run `{let1 {x {/ 8 0}} 7})
```

Unfortunately so does this

15
```
(run `{let1 {x y} 7})
```

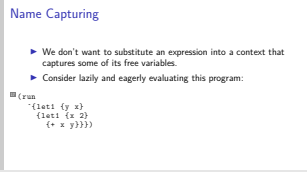
Why might we think about these as different?
Consider

16
```
(run `{let1 {y 0}
           {let1 {z y}
                 {let1 {x {/ 8 z}} 7}}})
```

Name Capturing

- We don't want to substitute an expression into a context that captures some of its free variables.
- Consider lazily and eagerly evaluating this program:

17
```
(run
  `{let1 {y x}
    {let1 {x 2}
      {+ x y}}})
```



1. Let1 the eager evaluator, name capture is not a problem because by the time we do the substitution, the named expression should not have free variables that need to be replaced.

Trade-offs between eager and lazy

- As long as the initial program is correct, both evaluation strategies produce the same results.
- If a program contains free variables, they might get captured in a naive lazy evaluator implementation
- It can be proved that when you evaluate an expression, if there is an error that can be avoided, lazy evaluation will always avoid it.

Trade-offs between eager and lazy

- As long as the initial program is correct, both evaluation strategies produce the same results.
- If a program contains free variables, they might get captured in a naive lazy evaluator implementation
- It can be proved that when you evaluate an expression, if there is an error that can be avoided, lazy evaluation will always avoid it.

1. This capturing is a bug
2. + On the other hand, lazy evaluators are usually slower than eager evaluator, so it's a trade-off.
3. Note that with lazy evaluation we say that an identifier is bound to an expression rather than a value. (Again, this is why the eager version needed to wrap 'eval's result in a 'Num' and this one doesn't.)

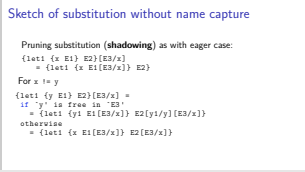
Sketch of substitution without name capture

Pruning substitution (**shadowing**) as with eager case:

```
{let1 {x E1} E2}[E3/x]
   = {let1 {x E1[E3/x]} E2}
```

For `x != y`

```
{let1 {y E1} E2}[E3/x] =
 if `y' is free in `E3'
   = {let1 {y1 E1[E3/x]} E2[y1/y][E3/x]}
 otherwise
   = {let1 {x E1[E3/x]} E2[E3/x]}
```



1. You can see that this is much more complicated, and probably not correct yet.

de Bruijn Indices

- Name capture is a problem that should be avoided.
- Note that the only thing we use names for are references.
- We don't really care what the name is:
- The only thing we care about is what variable points where.

```
{let1 {x 5} {+ x x}}
{let1 {y 5} {+ y y}}

(define (foo x) (list x x))
(define (foo y) (list y y))
```

de Bruijn Indices

- Name capture is a problem that should be avoided.
- Note that the only thing we use names for are references.
- We don't really care what the name is:
- The only thing we care about is what variable points where.

```
{let1 {x 5} {+ x x}}
{let1 {y 5} {+ y y}}

(define (foo x) (list x x))
(define (foo y) (list y y))
```

1. The sense in which these expressions are "obviously the same". is is called "alpha-equality").

Binding Structure

Which identifier references which expression is binding structure; we can visualize it in DrRacket

21

```
(let ([x 1])
  (let ([y x])
    (let ([x x])
      (+ x y))))
```

- Idea: if all we care about is where the arrows go, then simply get rid of the names
- Instead of referencing a binding through its name, just specify which of the surrounding scopes we want to refer to.

We can translate

```
{let1 {x 5} {let1 {y 6} {+ x y}}}
```

to use a new 'reference' syntax – "[N]" –

```
{let1 5 {let1 6 {+ [1] [0]}}}
```

Scope References

[0] is the value bound in the current scope, [1] is the value from the next one up etc...

> *To do this translation, we have to know the precise scope rules.*

More scope references

More complicated example:

```
{let1 {x 5} {+ x {let1 {y 6} {+ x y}}}}
```

Translates to:

```
{let1 5 {+ [0] {let1 6 {+ [1] [0]}}}}
```

Note that 'x' appears as a different reference based on where it appeared in the original code.

Even more scope references

Even more subtle:

```
{let1 {x 5} {let1 {y {+ x 1}} {+ x y}}}
```

translated to:

```
{let1 5 {let1 {+ [0] 1} {+ [1] [0]}}}
```

- What scope is the named expression of the inner `let1` in?

Even more scope references

Even more subtle:

```
{let1 {x 5} {let1 {y {+ x 1}} {+ x y}}}
```

translated to:

```
{let1 5 {let1 {+ [0] 1} {+ [1] [0]}}}
```

- What scope is the named expression of the inner `let1` in?

1. The inner ‘let1’ does not have its own named expression in its scope, so the named expression is immediately in the scope of the outer ‘let1’.

de Bruijn Indices

- Instead of referencing identifiers by their name, we use an index into the surrounding binding context.
- The major disadvantage, as can be seen in the above examples, is that the transformed code is not easy for humans to understand.

- Specifically, the same identifier is referenced using different numbers, which makes it hard to understand what some code is doing.
- Practically all compilers use de Bruijn indices for compiled code (think about stack pointers).