CS4613 Lecture 4: Binding and Substitution

David Bremner

March 21, 2023

Repeated Code

Repetition is not just extra computation

```
{* {+ 4 2} {+ 4 2}}
```

it leads to bugs

```
{* {+ 4 2} {+ 4 1}}
```

and obscures meaning

```
with x = {+ 4 2},
  {* x x}
```

Syntax to be negotiated, this is binding,
introducing an identifier.

Repeated Code
Repetition is not just extra computation
```
{+ {+ 4 2} {+ 4 2}}
```
it leads to bugs
```
{+ {+ 4 2} {+ 4 1}}
```
and obscures meaning
```
with x = {+ 4 2},
  {+ x x}
```
Syntax to be negotiated, this is binding, introducing an identifier.

1. These are often called "variables", but we will try to avoid this name: what if the identifier does not change (vary)?

Binding and substitution

Substitute away bindings

```
{let1 {x {+ 4 2}}
  {* x x}}
```

Reduce to: `{* 6 6}` by substituting 6 for 'x' in the body sub-expression of 'let1'.

A little more complicated example

```
{let1 {x {+ 4 2}}
  {let1 {y {* x x}}
    {+ y y}}}
```

A little more complicated example:

```
{let1 {x {+ 4 2}}
  {let1 {y {* x x}}
    {+ y y}}}
```

We can do a series of substitutions

```
= {let1 {x 6} {let1 {y {* x x}}  [add]
                    {+ y y}}}

= {let1 {y {* 6 6}} {+ y y}}     [subst]
= {let1 {y 36} {+ y y}}          [mul]
= {+ 36 36}                      [subst]
= 72                             [add]
```

Adding Bindings to AE: The LAE Language

- PLAI (1st ed) Chapter 3. PLAI3 "Evaluating Local Binding"
- To add `let1` to our language, we start with the BNF. We now call our language 'LAE' (Let1+AE):

- Two new grammar rules: one for introducing an identifier, and one for using it.

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

└Adding Bindings to AE: The LAE Language

Adding Bindings to AE: The LAE Language
- PLAI (1st ed) Chapter 3. PLAI3 "Evaluating Local Binding"
- To add `let1` to our language, we start with the BNF. We now call our language 'LAE' (`Let1+AE`):
- Two new grammar rules: one for introducing an identifier, and one for using it.

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

1. This is common in many language specifications, for example 'define-type' introduces a new type, and it comes let1 'type-case that allows us to destruct its instances.

Extended abstract syntax

We use Symbols for IDs (performance/tradition)

```
(define-type LAE
  [Num  (val : Number)]
  [Add  (l : LAE) (r : LAE)]
  [Sub  (l : LAE) (r : LAE)]
  [Mul  (l : LAE) (r : LAE)]
  [Div  (l : LAE) (r : LAE)]
  [Id   (name : Symbol)]
  [Let1 (name : Symbol)
        (val : LAE)
        (expr : LAE)])
```

Two new forms to parse

```
(cond
  ⋮
  [(s-exp-symbol? sx) (Id (s-exp->symbol sx))]
  [(s-exp-match? `(let1 (SYMBOL ANY) ANY) sx)
   (let* ([def (sx-ref sx 1)]
          [id (s-exp->symbol (sx-ref def 0))]
          [val (parse-sx (sx-ref def 1))]
          [expr (parse-sx (sx-ref sx 2))])
     (Let1 id val expr))]
```

Implementing 'let1' Evaluation

To evaluate:

```
{let1 {id LAE1} LAE2}
```

Evaluate 'LAE2' with 'id' substituted by 'LAE1'.

```
eval( {let1 {id LAE1} LAE2} ) =
        eval( subst(LAE2,id,LAE1) )
```

There is a more common syntax for substitution

```
eval( {let1 {id LAE1} LAE2} ) = eval( LAE2[LAE1/id] )
```

Now all we need is an exact definition of substitution.

Implementing 'let1' Evaluation

To evaluate:
{let1 {id LAE1} LAE2}

Evaluate 'LAE2' with 'id' substituted by 'LAE1'.
eval({let1 {id LAE1} LAE2}) =
eval(subst(LAE2,id,LAE1))

There is a more common syntax for substitution
eval({let1 {id LAE1} LAE2}) = eval(LAE2[LAE1/id])
Now all we need is an exact definition of substitution.

1. Note that substitution is not the same as evaluation, only part of the evaluation process. In the previous examples, when we evaluated the expression we did substitutions as well as the usual arithmetic operations that were already part of the AE evaluator. In this last definition there is still a missing evaluation step, see if you can find it.

Naive Substitution

e[v/i] := replace all occurances of 'i' in 'e' by 'v'.

Simple cases work

```
{let1 {x 5} {+ x x}}   -->  {+ 5 5}
{let1 {x 5} {+ 10 4}}  -->  {+ 10 4}
```

Nonsense results in other cases

```
{let1 {x 5} {+ x {let1 {x 3} 10}}}
          -->  {+ 5 {let1 {5 3} 10}} ???
```

Clearly there are different kinds of occurrences of identifiers.

Binding, Free, and Bound identifiers

- **Binding** Instance: names the identifier in a new binding.
- In our BNF syntax, binding instances are only the ID position of the 'let1' form.
- **Scope** region of program text in which instances of the identifier refer to the value bound in the binding instance.
- **Bound** Instance (or **Bound** Occurrence): contained within the scope of a binding instance of its name.
- **Free** Instance (or Free Occurrence): not contained in any binding instance of its name

Binding, Free, and Bound identifiers

- **Binding** Instance: names the identifier in a new binding.
- In our BNF syntax, binding instances are only the ID position of the 'let1' form.
- **Scope** region of program text in which instances of the identifier refer to the value bound in the binding instance.
- **Bound** Instance (or **Bound** Occurrence): contained within the scope of a binding instance of its name.
- **Free** Instance (or Free Occurrence): not contained in any binding instance of its name

1. Note that this definition of **scope** actually relies on a definition of substitution, because that is what is used to specify how identifiers refer to values.

Substitution, take 2

Goals

- don't substitute for binding instances
- handle shadowing

Definition (Substitution v2)

`e[v/i]` := replace all instances of 'i' that are free in 'e' with 'v'.

Racket substitution definition

```
; returns expr[to/from].
; leaves no free occurences of `to'
(define (subst expr from to)
  (type-case LAE expr
    :
    [(Add l r) (Add (subst l from to)
                    (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(Let1 bound-id named-expr bound-body)
     (if (eq? bound-id from)
       expr            ; <-- don't go in!
       (Let1 bound-id
             named-expr
             (subst bound-body from to)))]))
```

Let's test a few cases.

22

```
(test (subst
       ;; {+ x {let1 {x 3} x}
       (Add (Id 'x) (Let1 'x (Num 3) (Id 'x)))
       'x (Num 5))
      ;; {+ 5 {let1 {x 3} x}}
      (Add (Num 5) (Let1 'x (Num 3) (Id 'x))))
(test (subst
       ;; {+ x {let1 {y 3} x}}
       (Add (Id 'x) (Let1 'y (Num 3) (Id 'x)))
       'x (Num 5))
      ;; {+ 5 {let1 {y 3} 5}}
      (Add (Num 5) (Let1 'y (Num 3) (Num 5))))
```

- Before we find more bugs, we need to see how substitution is used in evaluation.

evaluating ‘let1’

To evaluate:

```
{let1 {x E1} E2}
```

- evaluate ‘E1’ to get a value ‘V1’
- substitute the identifier ‘x’ let1 the expression ‘V1’ in ‘E2’,
- evaluate the new expression.

In other words:

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

Evaluating identifiers?

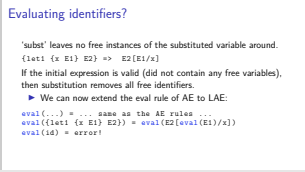
'subst' leaves no free instances of the substituted variable around.

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

If the initial expression is valid (did not contain any free variables), then substitution removes all free identifiers.

- We can now extend the eval rule of AE to LAE:

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

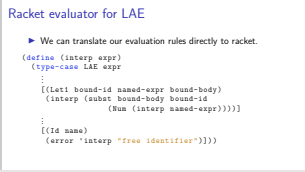


1. If you're paying close attention, you might catch a potential problem in this definition: we're substituting 'eval(E1)' for 'x' in 'E2' – an operation that requires an LAE expression, but 'eval(E1)' is a number. (Look at the type of the 'eval' definition we had for AE, then look at the above definition of 'subst'.) This seems like being overly pedantic, but we it will require some resolution when we get to the code.

Racket evaluator for LAE

- We can translate our evaluation rules directly to racket.

```
(define (interp expr)
  (type-case LAE expr
    ⋮
    [(Let1 bound-id named-expr bound-body)
     (interp (subst bound-body bound-id
                    (Num (interp named-expr))))]
    ⋮
    [(Id name)
     (error 'interp "free identifier")]))
```



1. We will (try to) follow the book and call this function `interp` from here on
2. Note the 'Num' expression in the marked line: evaluating the named expression gives us back a number – we need to convert this number into a syntax to be able to use it let1 'subst'. The solution is to use 'Num' to convert the resulting number into a numeral (the syntax of a number). It's not an elegant solution, but it will do for now.

Testing the evaluator


```
(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} {+ 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/exn (run `{let1 {x 1} y}) "free")
```

What about these cases?

29

```
(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} {let1 {y x} y}}) 5)
(test (run `{let1 {x 5} {let1 {x x} x}}) 5)
```

A missing substitution

In expressions like:

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

- We forgot to substitute 'x' in `{let1 {y x} ...}`
- We need to do the recursive subsititution in both the `let1`'s named-expression as well as its body.

Updated substitution function

```
;; expr[to/from]
(define (subst expr from to)
  (type-case LAE expr
    ⋮
    [(Let1 bound-id named-expr bound-body)
     (if (eq? bound-id from)
         expr
         (Let1
          bound-id
          (subst named-expr from to) ; new
          (subst bound-body from to)))]))
```

Another missing substitution

We still have a problem:

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

This should evaluate to 5, not fail with an error.

- When we subst. the outer 'x', we don't go inside the inner 'let1' (same name) – but we **do** need to go into its named expr.
- We need to subst. in the named expr. even if the ident is the **same** one we substituting.
- cf. `let` vs. `let*`

Updated substitution function v2

```
;; expr[to/from]
(define (subst expr from to)
  (type-case LAE expr
    ⋮
    [(Id name)
     (if (eq? name from) to expr)]
    [(Let1 bound-id named-expr bound-body)
     (Let1
      bound-id
      (subst named-expr from to)
      (if (eq? bound-id from)
          bound-body
          (subst bound-body from to)))]))
```

```
(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")
```