

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Spring 2004

Recitation 23
Analysis

Static Analysis

Since some sections of code will be evaluated repeatedly, performance can be improved by doing some work before beginning to evaluate, such that each evaluation takes less time. The `analyze` evaluator does this by computing how to evaluate an expression and saving it, such that at evaluation time, it doesn't need to re-figure it out each time the expression is evaluated. It saves the work in a procedure by returning a lambda that takes in an environment. Some bits of `analyze`:

```
(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))

(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env)
      (define-variable! var (vproc env) env)
      'ok)))

(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application (fproc env)
                           (map (lambda (aproc) (aproc env))
                                aprocs)))))

(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
        (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                            args
                            (procedure-environment proc))))
        (else
         (error "Unknown procedure type -- EXECUTE-APPLICATION"
                proc))))
```

1. Assuming that `analyze` is extended to include `((let? exp) (analyze-let exp))`, implement `analyze-let`:

```
(define (analyze-let exp)
```

Syntactic Analysis

In the project, the question on desugaring `or` mentions that desugaring `or` into a `let` can produce an accidental capture. This would happen when a subexpression of the `or` uses the same variable as is chosen for the `let`. One method to avoid this is to dynamically pick the variable to be one that isn't used.

With respect to an expression, a **free variable** is one whose value is not found in a frame created by the expression. A **bound variable** is one whose value *is* found in a frame created by the expression. For example, in the expression `(lambda (x) (+ x y))`, `y` is a free variable¹ and `x` is a bound variable. We would say that the `(lambda (x))` *captures* the variable `x`, as `x` is free in the body of the lambda: `(+ x y)`.

2. What are the free and bound variables in the following expression:

```
((lambda (x y z)
  (if z
    (+ x y)
    7))
 4 val x)
```

In order to avoid the variable capture issue mentioned above, we just need to pick a variable that isn't free in the subexpressions. So, let's write some code to detect free variables. First, a data structure:

```
(define (empty-free)          ; builds an empty free list
  (list))

(define (single-free var)    ; builds a free list of one variable
  (list var))
```

¹technically, `+` is also a free variable

```
(define (bind-var var freelist) ; removes a variable from the list
  (delq var freelist))          ; (it's been bound/captured)

(define (free-var var freelist) ; adds a variable to the list
  (if (not (memq var freelist)) ; avoiding adding duplicates
      (cons var freelist)
      freelist))

(define (merge-freelists f1 f2) ; if variable is free in either list
  (fold-right free-var f1 f2)) ; it's free in the result
```

3. We'll need some code to deal with a list of expressions. If a variable is free in any of the expressions, it's free in the whole set, so write `free-in-list`:

```
(define (free-in-list exps free)
```

4. Write `free-in`, which takes an expression and returns a freelist of the variable that are free in that expression.

```
(define (free-in exp)
  (cond ((self-evaluating? exp)

        ((variable? exp)

         ((lambda? exp)

          ((if? exp)
           (free-in-list (cdr exp) (empty-free)))
          ((definition? exp)
           (bind-var (define-variable exp)
                     (free-in (define-value exp))))
          ((assignment? exp)
           (free-in-list (cdr exp) (empty-free)))
          ((let? exp)
           (free-in (let->application exp)))
          ((application? exp)
           (free-in-list exp (empty-free)))
          (else
           (error "free-in unknown exp" exp))))))
```

Now that we've got `free-in`, we can write `unused-variable`, which picks a variable name that isn't on the free list. Well, the concatenation of `unused` with all the variables on the free list is guaranteed not to be used, so:

```
(define (unused-variable freelist)
  (fold-right symbol-append 'unused freelist))

(unused-variable (free-in '((lambda (x y) (+ x y)) 4 val)))
;Value: +valunused
```

Halting Theorem

`Halts(M,I)`

- returns `true` if machine `M` halts on input `I`
- returns `false` if machine `M` does not halt on input `I`

Machine `H`:

```
H(M)
  if halts(M,M)
    run forever
  else
    halt
```

What happens if `H` is run on itself: `(H H)`