CS450 - Structure of Higher Level Languages

Continuation Passing

November 9, 2020

Continuation Style Programming

- A function written in **continuation-passing** style takes an extra argument: an explicit "continuation", i.e. a function of one argument that receives the result of the expression.
- When the Continuation Style function has computed its result value, it "returns" it by calling the continuation function with this value as the argument.
- The calling function is therefore required to supply a procedure to be invoked with the "return" value.
- The key is to remember that
  1. every function takes an extra argument, its continuation
  2. every argument in a function call must be either a variable or a lambda expression (not a more complex expression).
- This basically turns expressions "inside-out" because the innermost parts of the expression must be evaluated first, so we explicate the order of evaluation as well as the control flow.

A Simple Example

- Given the following function:
  `(+ 4 (+ 1 2))`
- the result of `(+ 1 2)` will be added to 4. The addition to 4 is that expression's continuation.
- We can also represent this concept of continuation as:
  `(lambda (v) (+ 4 v))`
- We can make continuations explicit by using the built-in procedure `call-with-current-continuation` or `call/cc`
- `(call/cc expr)` does the following:
  1. Captures the current continuation.
  2. Constructs a function `C` that takes one argument, and applies the current continuation with that argument value.
  3. Passes this function as an argument to expr – i.e., it invokes `(expr C)`.
  4. Returns the result of evaluating `(expr C)`, unless `expr` calls `C`, in which case the value that is passed to `C` is returned.

https://courses.cs.washington.edu/courses/cse341/04wi/lectures/15-scheme-continuations.html

The call/cc Procedure

```
(+ 4 (call/cc
    (lambda (cont) (cont (+ 1 2)))))
```

- The continuation is what receives the result of the expression (i.e. `(+ 4 ...)`
- The result of evaluating `(+ 1 2)` is passed directly to the continuation.
- This is pretty much like writing:

```
((lambda (cont) (cont (+ 1 2)))
    (lambda (v) (+ 4 v))
```

https://courses.cs.washington.edu/courses/cse341/04wi/lectures/15-scheme-continuations.html

The begin.scm Example

```
(begin
  (display 3)
  (call/cc
   (lambda (xyz)
     (display 4)
     (xyz 11) ; Argument is ignored.
     (display 5)
     )) ; End call/cc
  (display 6)
  )
```

The continuation is:

```
(lambda (val)
  (display 6)
  (exit)) ; This explains why 5 is never displayed!
```

The `begin.scm` Example

- The "exit" is a non-standard exit procedure that escapes from the current evaluation.
- This can't be written in Scheme.
- So a continuation, even though it acts as a procedure in the sense that it takes one argument which it evaluates, is different from a procedure in that a call to it does not return.
- So, what happens is this (after `(display 3)`):

```
((lambda (xyz)
(display 4)
(xyz 11) ; Argument is ignored.
(display 5))
(lambda (val) (display 6))) ; with the implicit exit
```

The begin.scm Example

What about now?

```
(begin
  (display 3)
  (call/cc
   (lambda (xyz)
     (display 4)
     (xyz 11)
     (display 5)
     ))
  )
```

- The continuation `xyz` is bound to the print part of the read-eval-print loop of the Scheme interpreter:

  `(lambda(val) (print val))`
- Therefore, the 11 is printed by the interpreter and nothing further is printed.

The begin.scm Example

What gets printed is therefore equivalent to:

```
(begin
  (display 3)
   ((lambda (xyz)
     (display 4)
     (xyz 11)
     (display 5)
     ) (lambda (val) (print val) (exit)))) ; Implicit exit
```

Or something like:

```
(begin
  (display 3)
   ((lambda (lambda (val) (print val) exit)
     (display 4)
     ((lambda (val) (print val) (exit))11)
     (display 5)
     )))
```

The begin.scm Example

What about now?

```
(begin
  (display 3)
  (display (call/cc  ;; Note the display
   (lambda (xyz)
     (display 4)
     (xyz 11)
     (display 5)
     )))
  (display 6))
```

Here what is passed is

```
(lambda (val)
  (display val)
  (display 6)
  (exit))
```

The `save_continuation` Example

- This is a simple example to show how `call-with-current-continuation` can be used to implement exception handling.
- The example is so trivial that it could be done more simply; the point is just to show how `call/cc` can be used.
- We will define a procedure (`main_loop`) which when invoked asks the user to input a number different from 0.
- If the user inputs the number 0 or anything other than a number, a message is generated explaining what the user did wrong, and the loop starts over.
- If the input is acceptable, the procedure echoes it and quits.
- This can be used as a basis for HW7, part 2. I highly recommend it.

The save_continuation Example

```
; Initial definition, to be overwritten
(define target '())

;;; Define a procedure which needs to escape.
;;; Use the target to tell
;;; it where to escape to.
(define (f x)
  (cond ((= x 0)
         (display "0 entered; try again.")
         (newline)
         (target x)) ;;; the argument x will be ignored.
        (else
         (display "Success: ")
         (display x)
         (newline))
        )  )
```

The save_continuation Example

```
(define (main_loop)
  (call/cc
   (lambda(here)
     (set! target here))) ; End of call/cc
  (display "Type a number different from 0:  ")
  (let ((n (read)))
    ;; First check to make sure that a number was entered.
    (if (not (number? n))
        (begin
          (display n)
          (display " is not a number; try again.")
          (newline)
          (target n)   ;;; the argument n will be ignored.
          )
        )
     ;; OK; a number was entered.
     ;;; Now call f to do the rest
    (f n)
    ) )
```

The save_continuation Example

What is the continuation here? It should be something like this:

```
(lambda (val)
  (display "Type a number different from 0:  ")
 (let ((n (read)))

   ;; First check to make sure that a number was entered.
   (if (not (number? n))
        (begin
        (display n)
        (display " is not a number; try again.")
          (newline)
          (target n)   ;;; the argument n will be ignored.
          ) )

;; OK; a number was entered.
;; Now call f to do the rest of the processing.
     (f n) )
  (exit) ;; well, escape to the top level!
  )
```

Modifying the Meta-Circular Evaluator

- In Scheme, and in s450, all arguments are passed by value. Other languages use other semantics:
  - Dynamic rather than static scoping. A variable's value is looked up in the runtime environment.
  - Call by-reference: C++ and Java have this option.
- Additionally, we want to implement delayed (lazy) evaluation that allows us to support infinite streams.
- HW7, part 1, requires you to implement all that.
- A word of warning: You will get both parts at the same time, and you have two weeks for both. However, part 1 is **A lot** more time consuming than part 2.
- As a matter of fact, once you understand the concept of continuation passing, part 2 can be implemented in a very short time.

Delayed Arguments

- A delayed argument is packed into an object named "thunk".
- It needs to include the expression, the environment and an indication that this is a thunk.
- For example, a list of "I'm a thunk", exp, env.
- Remember that currently, all arguments are evaluated before application.
- The code needs to be changed so delayed arguments are "intercepted" before they are evaluated.
- There are several ways to do it. Do whatever works.

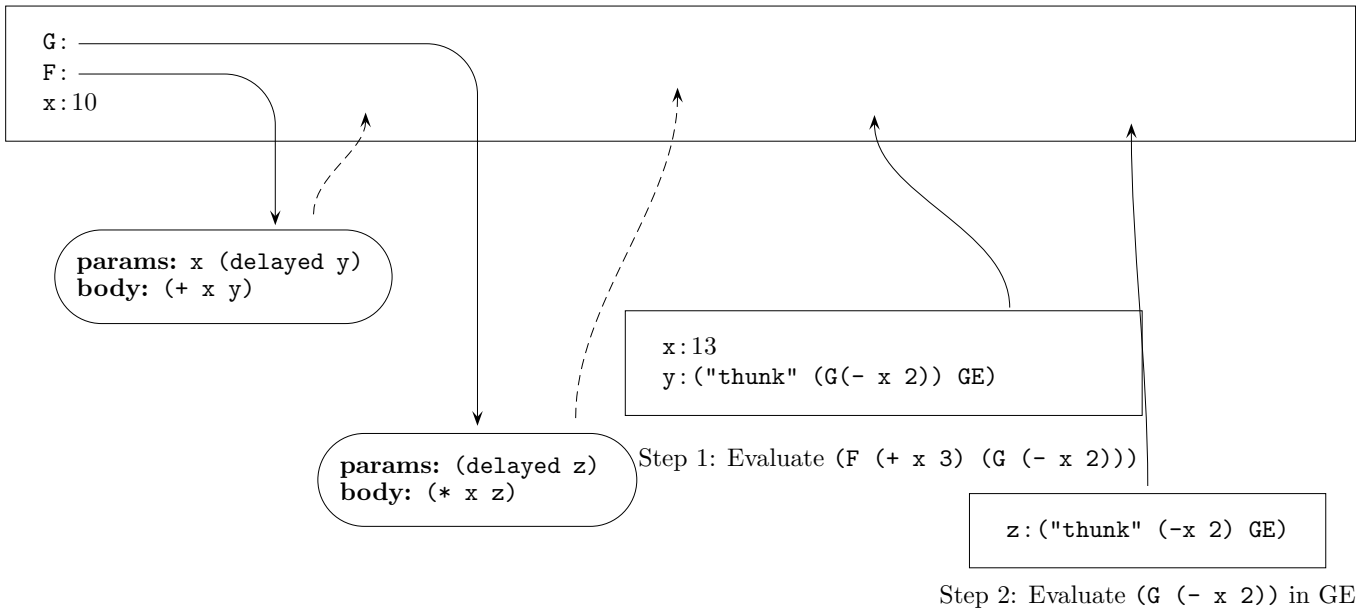
Delayed Arguments Example

Given the following piece of code:

```
(define x 10)
(define (F x (delayed y))
            (+ x y))
(define (G (delayed z))
            (* x z))
(F (+ x 3) (G (- x 2)))
```

Let us see what happens during evaluation.

Delayed Arguments Example



Next steps:

3. Force first thunk, G evaluates to `(*10 ("thunk" (- x 2) GE))`
4. Force this thunk, evaluate `(- x 2)` in GE (global environment).
5. Evaluate to 80, return to evaluating F.
6. Evaluate `(+ 13 80)` to get 93.

Delayed Arguments Example

- Now you have the tools to implement streams.
- Implement `cons-stream` as a special form.
- Implement other stream operations accordingly.
- You can use the thunk mechanism for it.
- There is no need to use "delayed" variables explicitly since it's a special form.
- I recommend using infinite streams for testing.

Dynamic Arguments

- The evaluation here is done as any other applicative order variable (that is, before application).
- The argument is evaluated in the dynamic environment of the program at run-time, rather than in the (static, lexical) environment associated with the procedure object.
- At any one time there is only one active dynamic environment.
- I suggest that you keep a global variable, perhaps `the-dynamic-environment`. This environment is a list of frames, just like the static environment.
- You need to manage this variable like a stack.
- Each time a function is invoked, you push a frame onto `the-dynamic-environment` (using `cons`).
- Make sure you understand how `xtend-environment` works.

Dynamic Arguments

- The frame that you push onto `the-dynamic-environment` is really the same frame that is added to the static environment.
- The difference is that you are not adding to the procedure's environment (i.e., the environment that the procedure object was originally defined in), but instead you are adding to the dynamic environment. (That's why it's a stack.)
- Each time a function terminates, you have to restore `the-dynamic-environment` to its previous state.
- Thus, after the function call is complete, you pop that frame off of `the-dynamic-environment`.

Dynamic Arguments

- The simplest way to add the frame is to do it when you are adding a frame to the static environment, i.e., in `xtend-environment`.
- But be careful: the return value of `xtend-environment` has to be the static environment, not the dynamic environment.
- Where does the dynamic environment get restored to its previous value after the procedure call?
- One obvious place is in `xapply`. But again you have to be careful: the return value of `xapply` is the return value of the procedure.
- Make sure you don't throw that value away when you restore the dynamic environment.
- You may have another idea of where to restore the dynamic environment. Whatever you do, document it carefully.

Dynamic Argument Example

```
(define f (lambda(x)(lambda(y)(cons (g x) y))))
(define g (lambda((dynamic z))(cons z 4)))
(define h (f 2))
(define x 1)
```

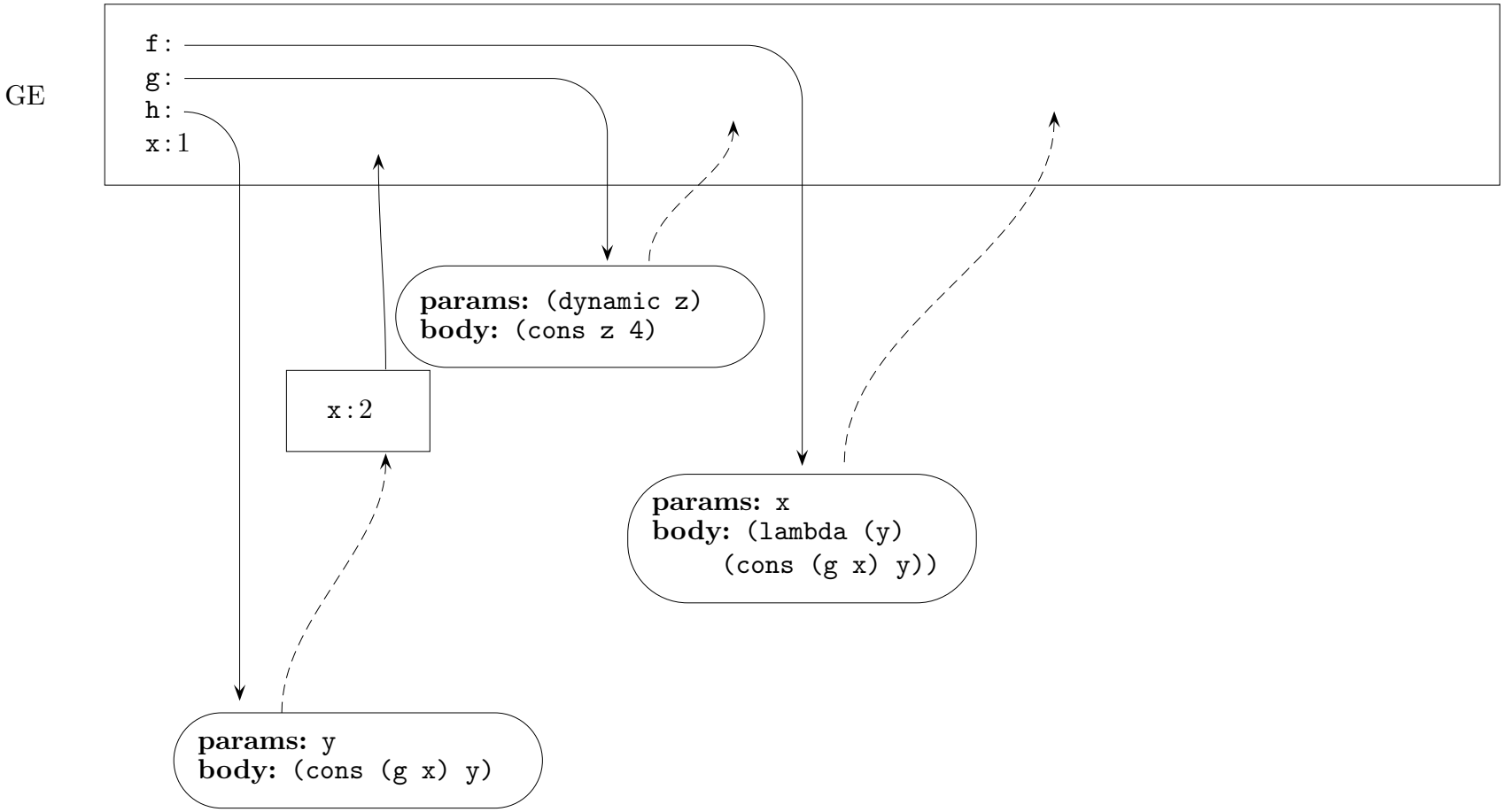
then

```
s450==> (h 5)
((1 . 4) . 5)
```

Let's wrap our heads around it...

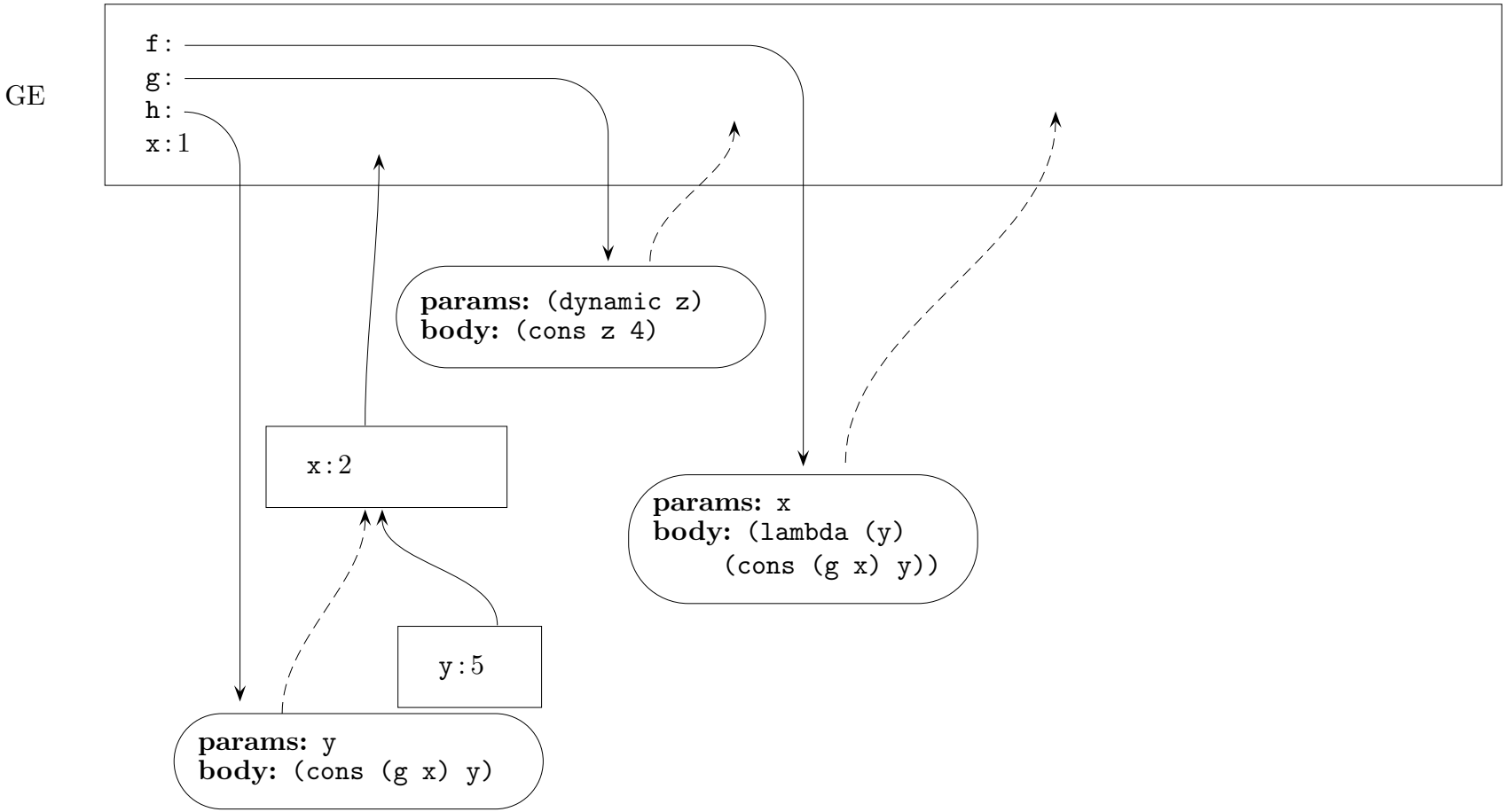
Dynamic Argument Example

Global Environment before call to `(h 5)`



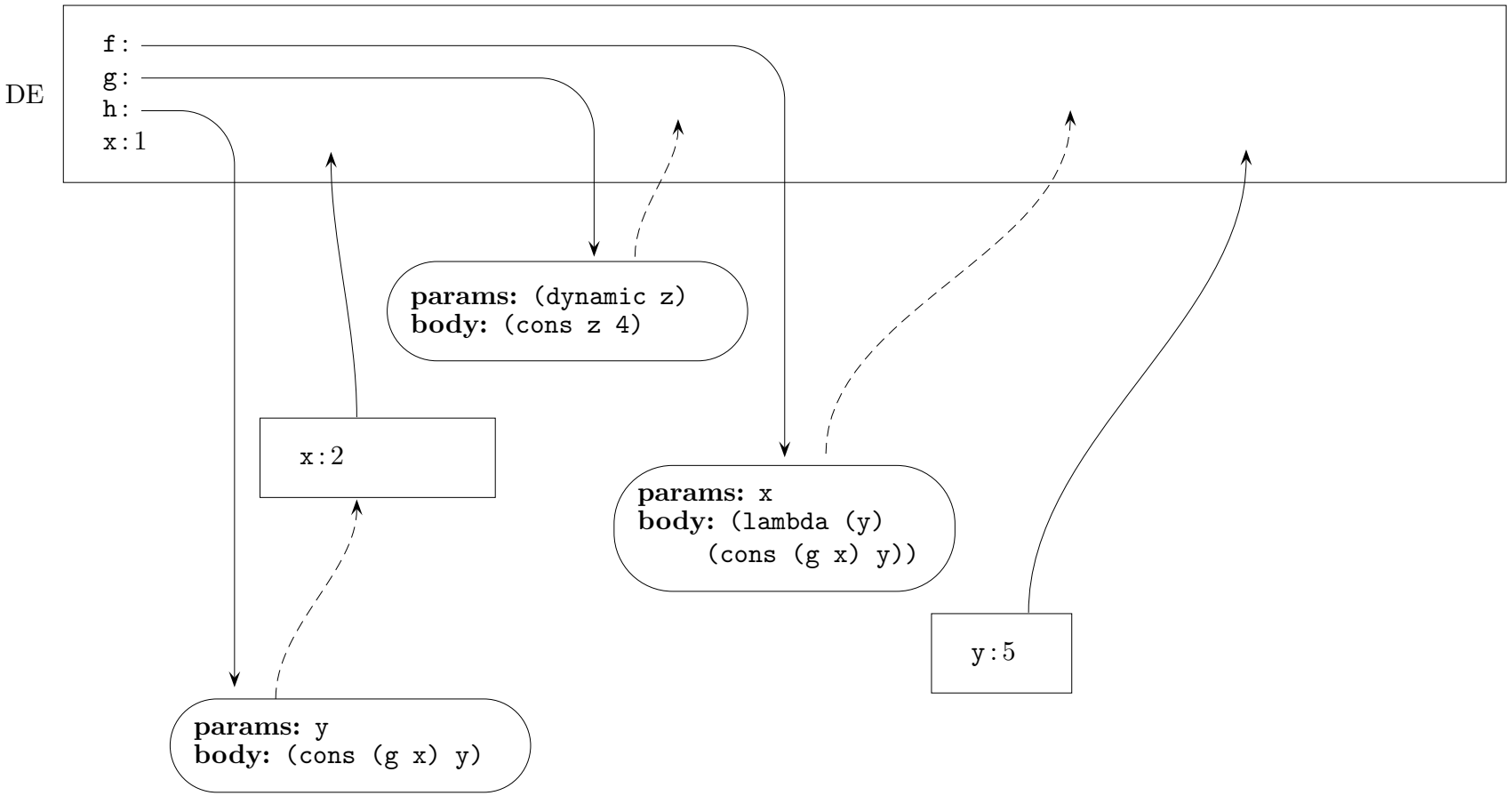
Dynamic Argument Example

Global Environment after call to `(h 5)`



Dynamic Argument Example

Dynamic Environment after call to `(h 5)`



Dynamic Argument Example

- z (g's formal parameter) is being evaluated in the dynamic environment.
- g's body is `(cons z 4)`, so `(g x)` takes its value from the environment g points to, which is the same as the global environment.
- Remember that after h is executed, the dynamic environment is restored to its previous state, in this case becomes the same as GE.
- Hence, z is bound to x which evaluates to 1.
- The body of g evaluates to `(1 . 4)` and returns to h.
- h them applies the second `cons` to get `((1 . 4) . 5)`.
- **Question:** What would we get if g's argument weren't dynamic?

Reference Arguments

- The reference formal argument `w` represents a reference to the actual argument.
- There is an important difference between reference arguments and the other kinds of arguments:
- When a lambda expression is applied, the actual arguments for the other kind of arguments can be arbitrary expressions, but (in our implementation) an actual reference argument must be a symbol with a value in the environment in which the lambda expression is evaluated.
- You can use ideas from HW6, when implementing the `defined?` special form.

Reference Arguments

- Actually, much of the implementation of reference arguments follows that of delayed arguments.
- When a lambda expression is invoked, you need to create a thunk-like object (give it a different name, though, like "reference") that holds the actual argument (which must be a symbol) and the environment in which it is found.
- During evaluation, if you encounter a formal argument that is bound to such a "reference", then you `xeval` the actual argument in the saved environment.
- This is the same thing as what you do when you force a thunk.

Difference Between Reference and Delayed Arguments

- When you make a change to a delayed argument, you change what that argument is bound to.
- Setting the argument equal to 4, for instance, means that it is no longer bound to a thunk.
- When you make a change to a reference argument, you do not change what the argument is bound to, but you change the value of the symbol it is bound to.
- That is, you are actually reaching into the environment carried along with the reference argument and changing the value bound to the referenced symbol.
- The difference between reference and delayed arguments appears in their different behavior under `set!`. For example,

```
((lambda ((reference x)(delayed y))
         (set! x 3)(set! y 4)) a b)
```

changes the value of a, but not the value of b.

Reference Argument Example

```
(define u 3)
(define x 10)
(define t 2)
(define g (lambda ((reference x))(f x x) t))
(define f (lambda ((reference x)(reference y))
            (set! x 5) y))
```

then

```
s450==> (f u u)
5
S450==> (f x x)
5
S450==>(g t)
5
```

Another Reference Argument Example

```
s450==> (define f (lambda (x (reference y))
                  (display (cons x y))(newline)
                           (set! x 10)
                          (set! y 20)
                  (cons x y)))
f

s450==> (define a 1)
a

s450==> (define b 2)
b
```

Another Reference Argument Example (cont.)

```
s450==> (f a b)
(1 . 2)  ; from the display statement in f
(10 . 20) ; pair returned by f,
; displayed at end of r-e-p loop

s450==>a
1  ; actual argument a (passed by value) is unchanged

s450==>b
20  ; actual argument b (passed by ref.) has new value

s450==> (f a 2)
error: lambda expression -
    actual argument for reference formal argument y must
    be a defined symbol
reset and all that stuff ...
```