Closures & Continuations

Overview

- Controlling the Control
  - Closures
  - Continuation

Bindings Revisited

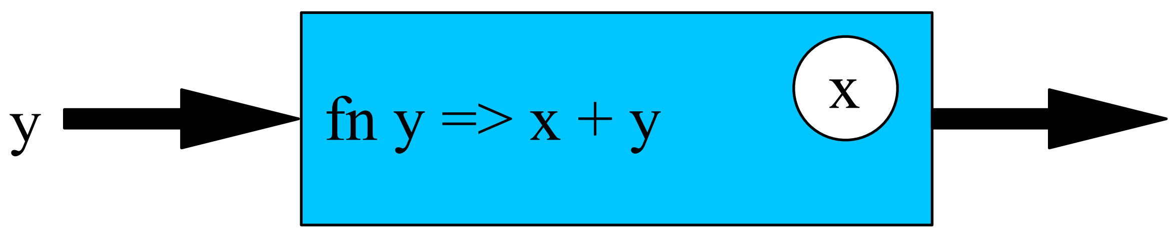
- Consider the following fragment

```
fun foo f = f 10;

val special = let val x = 100
              in fn y => x + y
              end;

foo special;
```

- What is going on here ?

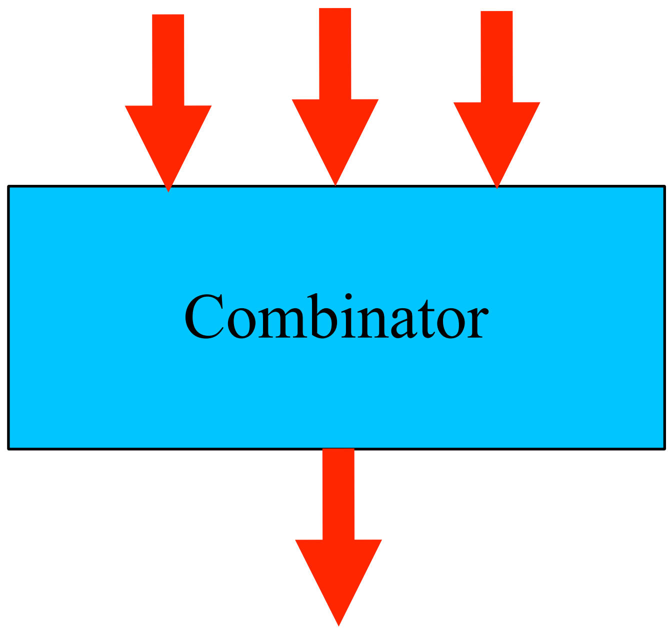


Environment Semantics

- The SML environment is….
  - A "Stack" of bindings
- Each time you state "val x = …"
  - You push a new binding on the stack
- Each time you state "let val x = … in … end
  - You *temporarily* push a binding on the stack
  - That is popped when reaching the "end"
- Each time you refer to a binding by name
  - You hold onto the matching environment
  - The environment sticks around as long as something refers to it!

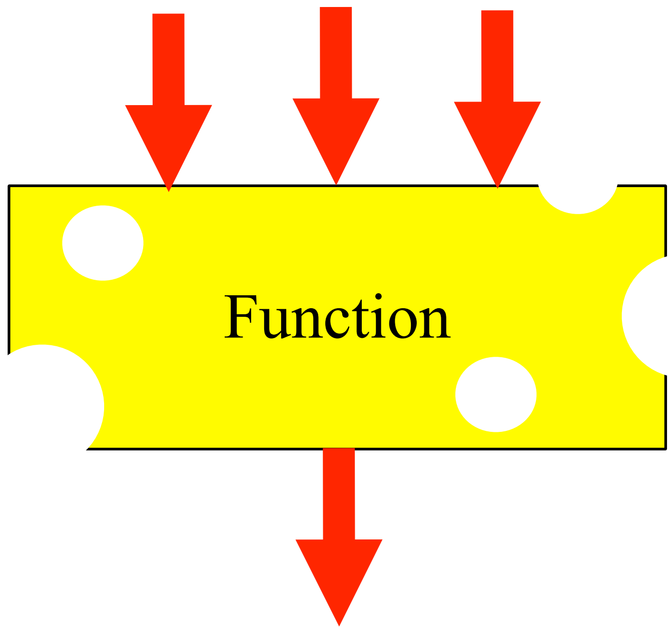
Combinators

- What are combinators ?



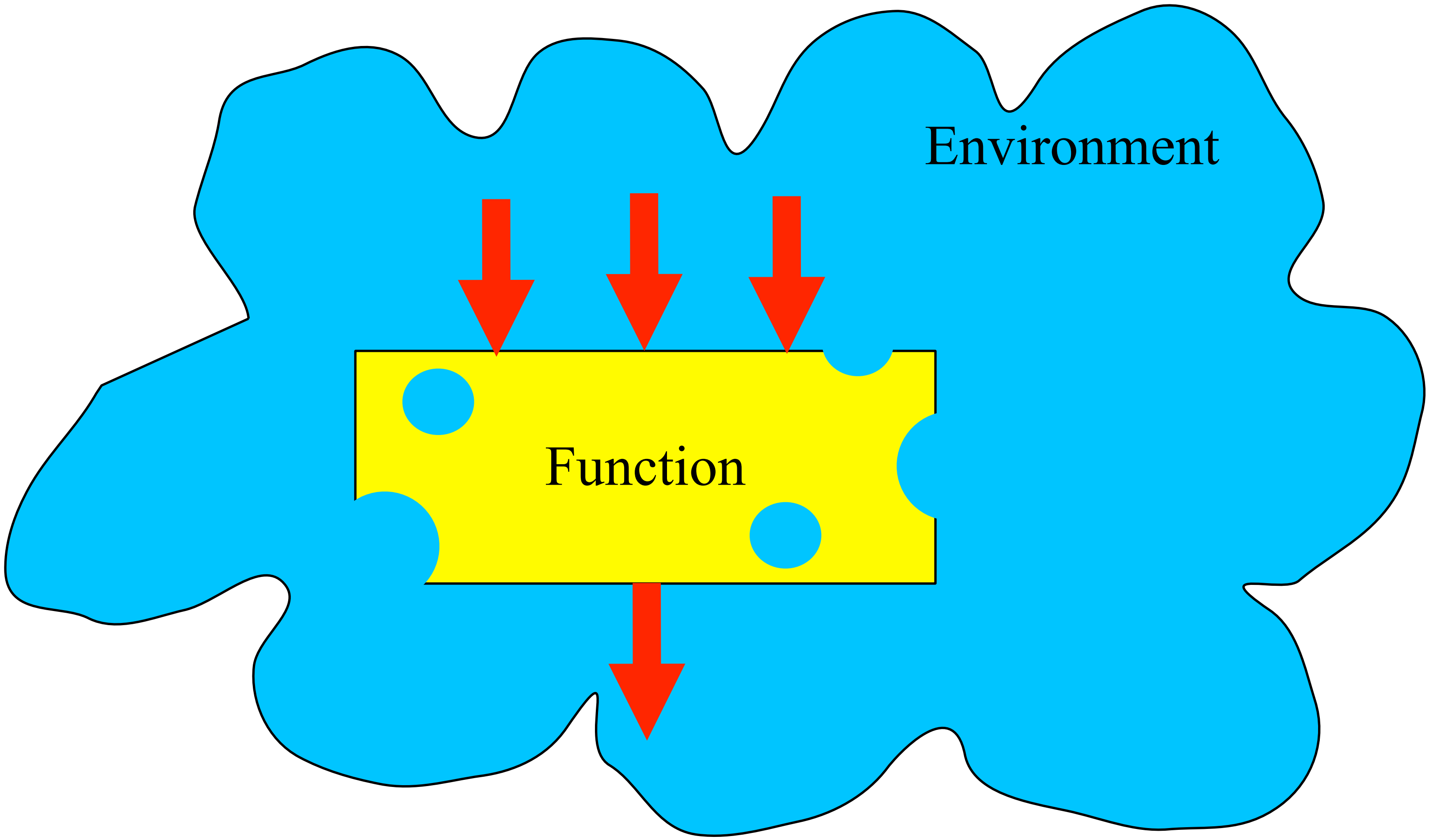
Plain Old Functions

- Any Function
  - What is the difference ?



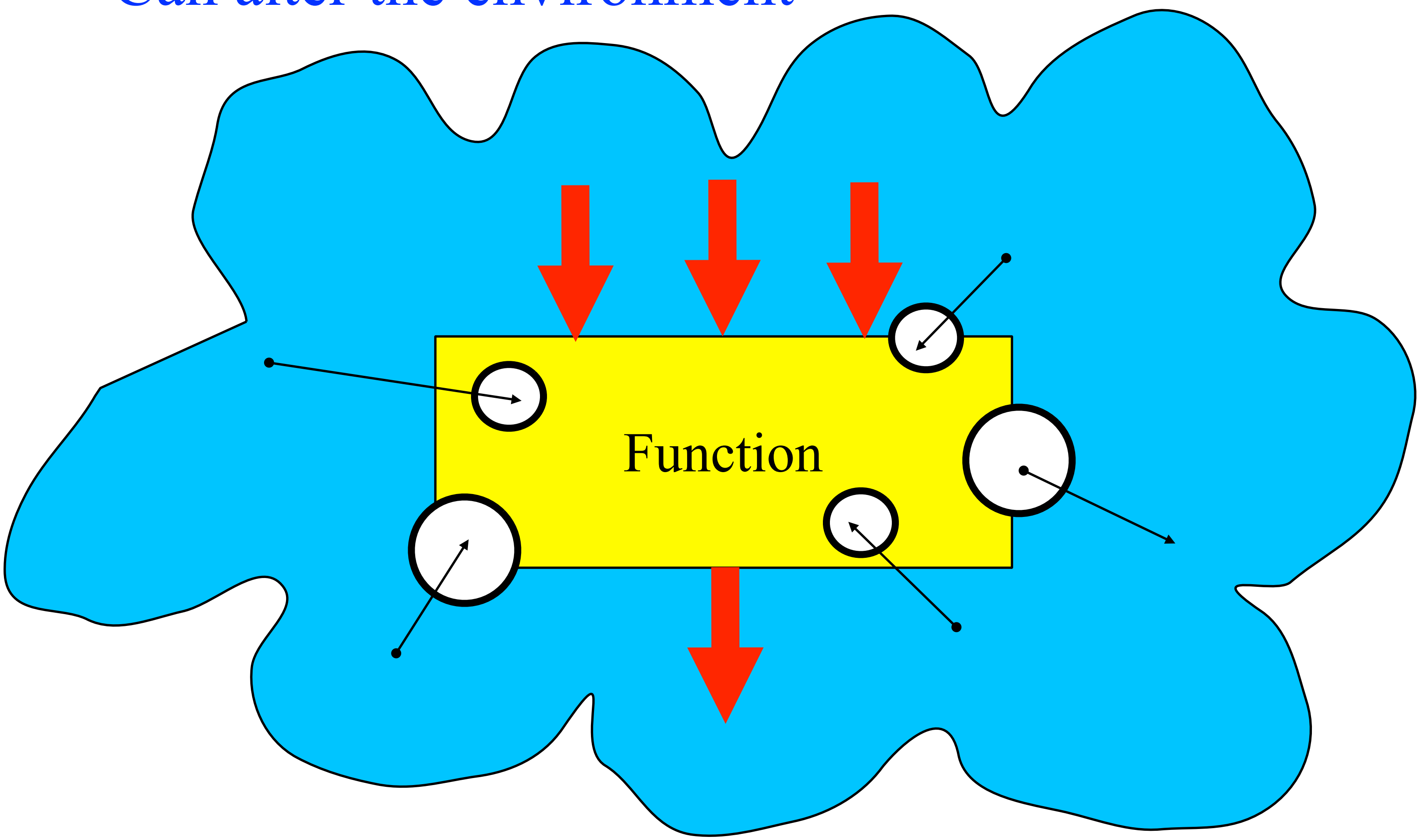
Environment

- The environment Provides
  - The “filling”



Alternative View

- Functions
  - Depend on the environment
  - Can alter the environment

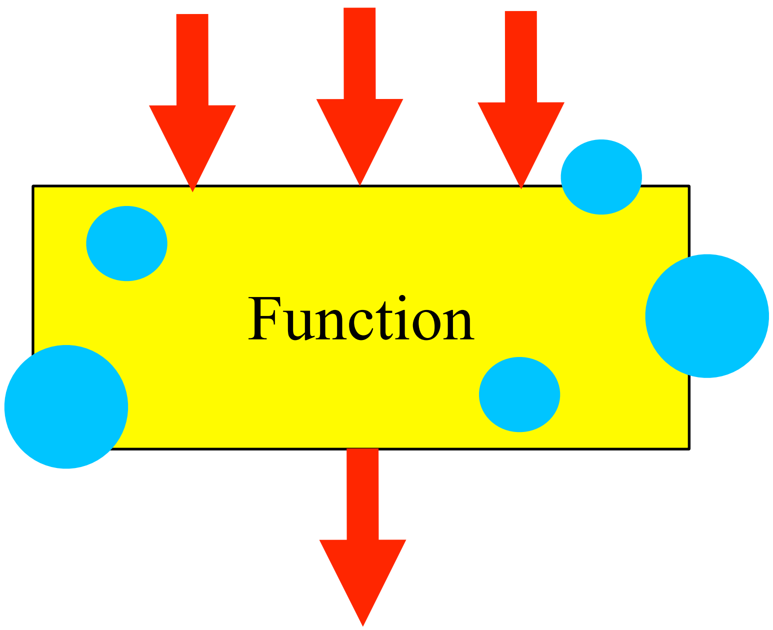


Combinators

- Can be composed
  - Easily
  - Just feed the output of one
  - Into the input of others!
- Can be dealt with
  - Independently
  - Without any reference to any context
- What about “plain functions” ?

Closures

- Function in context
  - A way to deal with a function
    - Package
      - The function
      - Its context
    - Deal with “the package” as a whole



What About ML...

- What is ML really doing ?
  - ML **never** let you use functions with “holes”
  - ML **always** provides closures
- Corollary
  - In ML, **every** “function” is actually a “combinator”

Example

- A direct example
  - What is the context ?

```
val x = 10
fun add n = x + n

val y1 = add 1
val z1 = add 2
val x = 20
val y2 = add 1
val z2 = add 2
```

Output ?

Example

- A more subtle example
  - What is the context ?
  - What is so subtle about it ?

```
val f = let val x = 20
            fun add n = x + n
        in add
        end

val y = f 10
```

Output ?

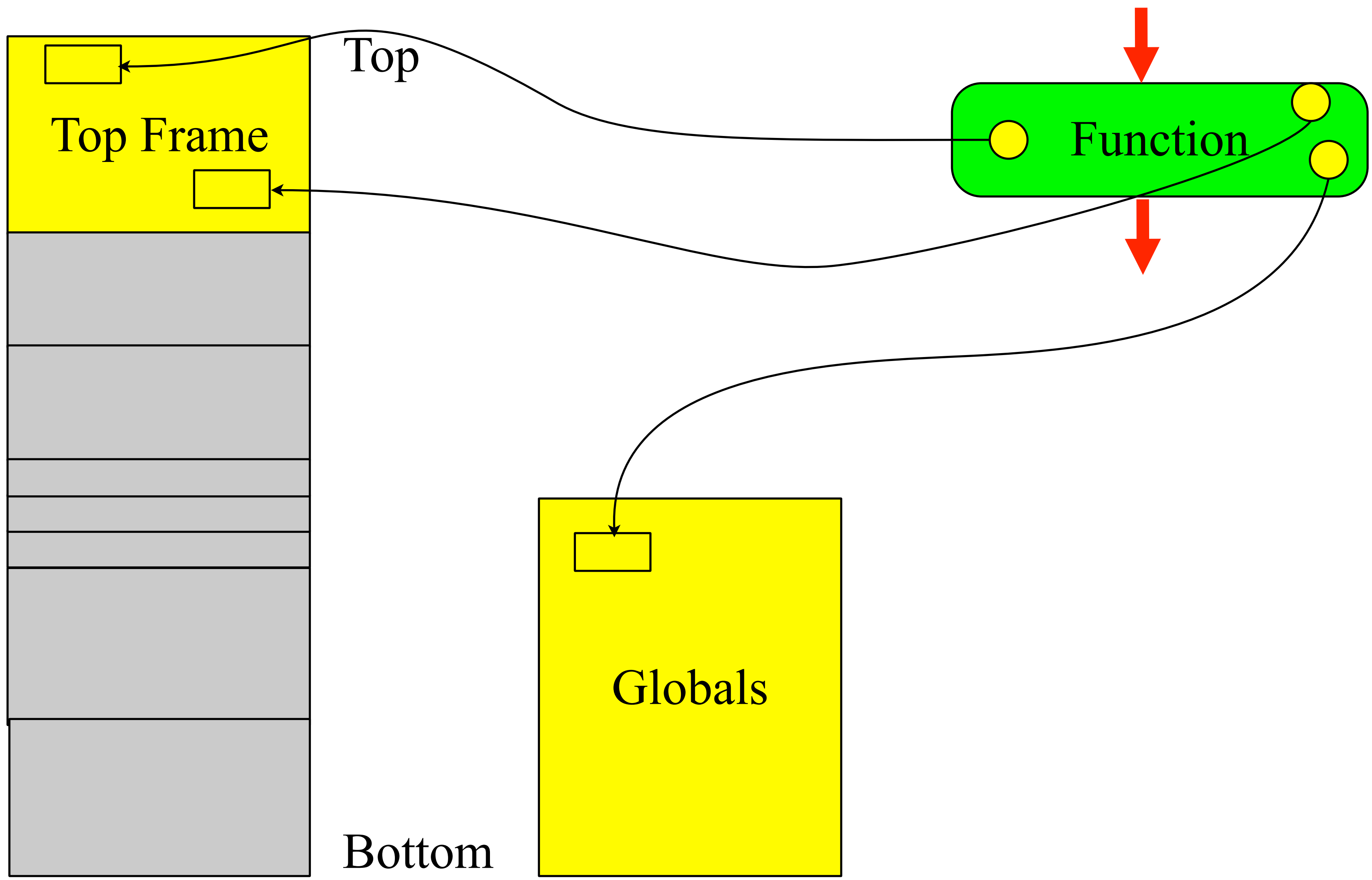
Closures & Context

- Which context should be used ?
  - The context current at closure creation time
- What constitutes the context ?
  - Must take into account
    - Model
      - Flat
      - Nested
    - Scoping
      - Static
      - Dynamic

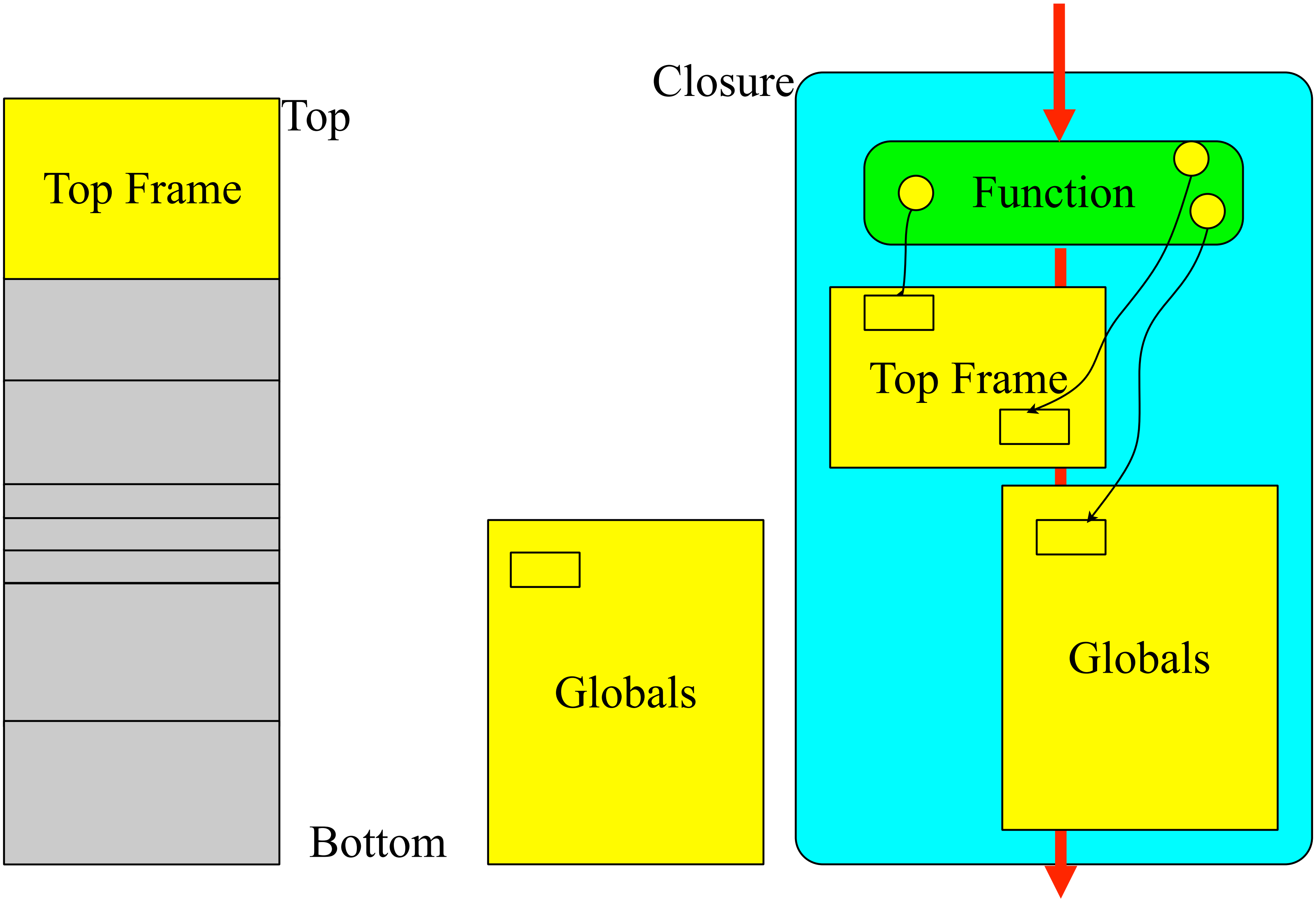
Flat Model / Static Scoping

- Issue
  - What is visible inside the function ?
- What must be saved...
  - In a stack-based implementation ?
  - In a functional implementation ?

Stack-Based



Stack-Based



Top Frame
Top
Bottom
Globals
Closure
Function
Top Frame
Globals

Implication

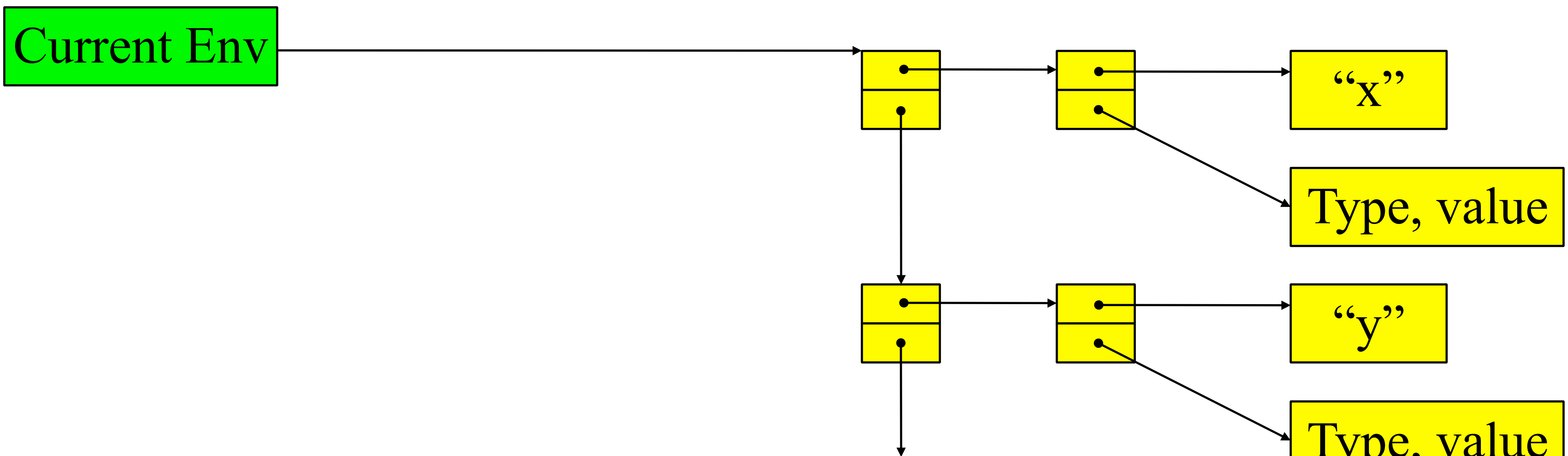
- We must be able to
  - Manipulate the stack
    - Copy frames from the stack
      - Easy to see...
    - Copy frames back to the stack
      - Why ?
  - What about the globals ?
- Questions
  - How expensive is a closure creation ?
  - Can we improve ?

Functional Context

- Do we really need a stack ?
  - Of course not!
- Stacks are good for
  - Languages with destructive operations
    - e.g., assignments
  - Efficient update
  - variable lifetime = variable scope
- With functional languages
  - No destructive operation
  - Variable lifetime can exceed scope
- We can take advantage of that to do away with the stack!

Environments

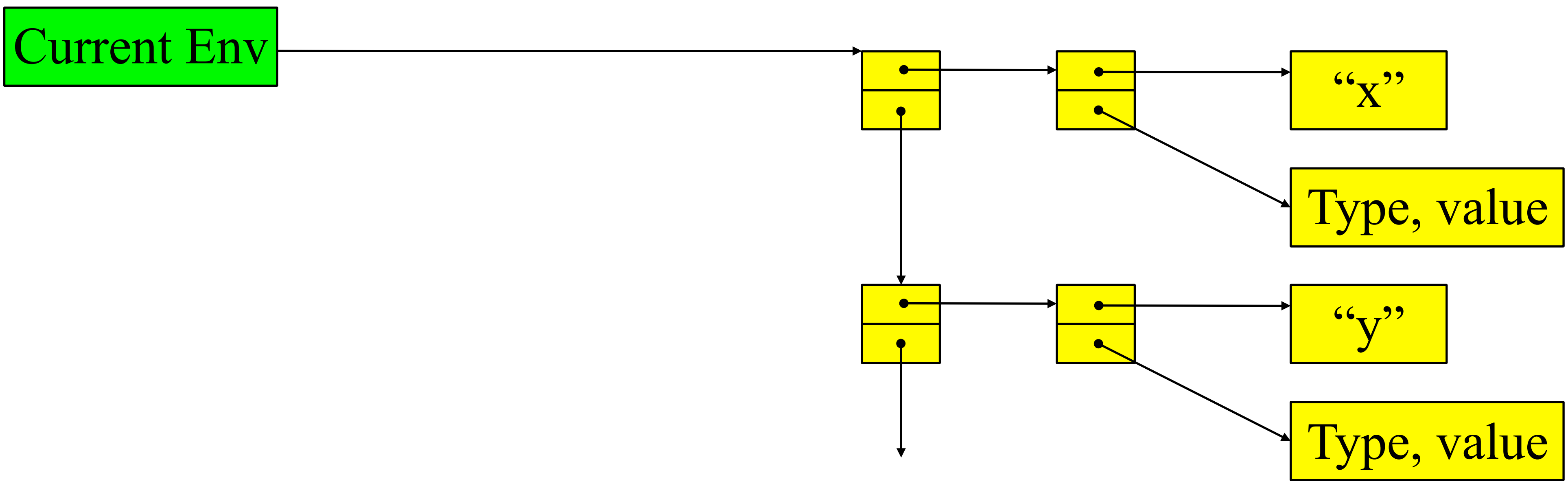
- Implementation with either
  - Association list
  - Central reference table
- Association list



Binding a Name

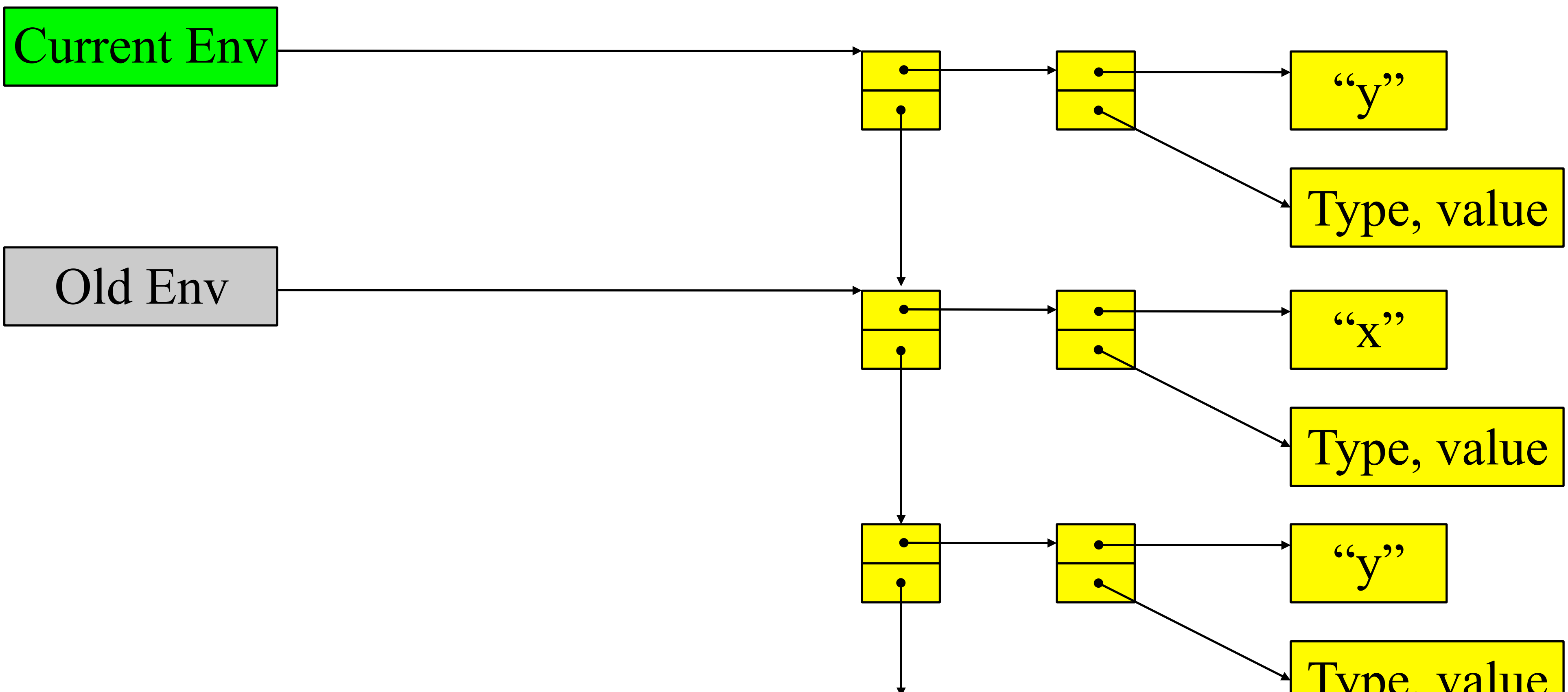
- What It does

```
...
let y = <some expr>
in  <some other expr>
end
...
```



Binding a Name

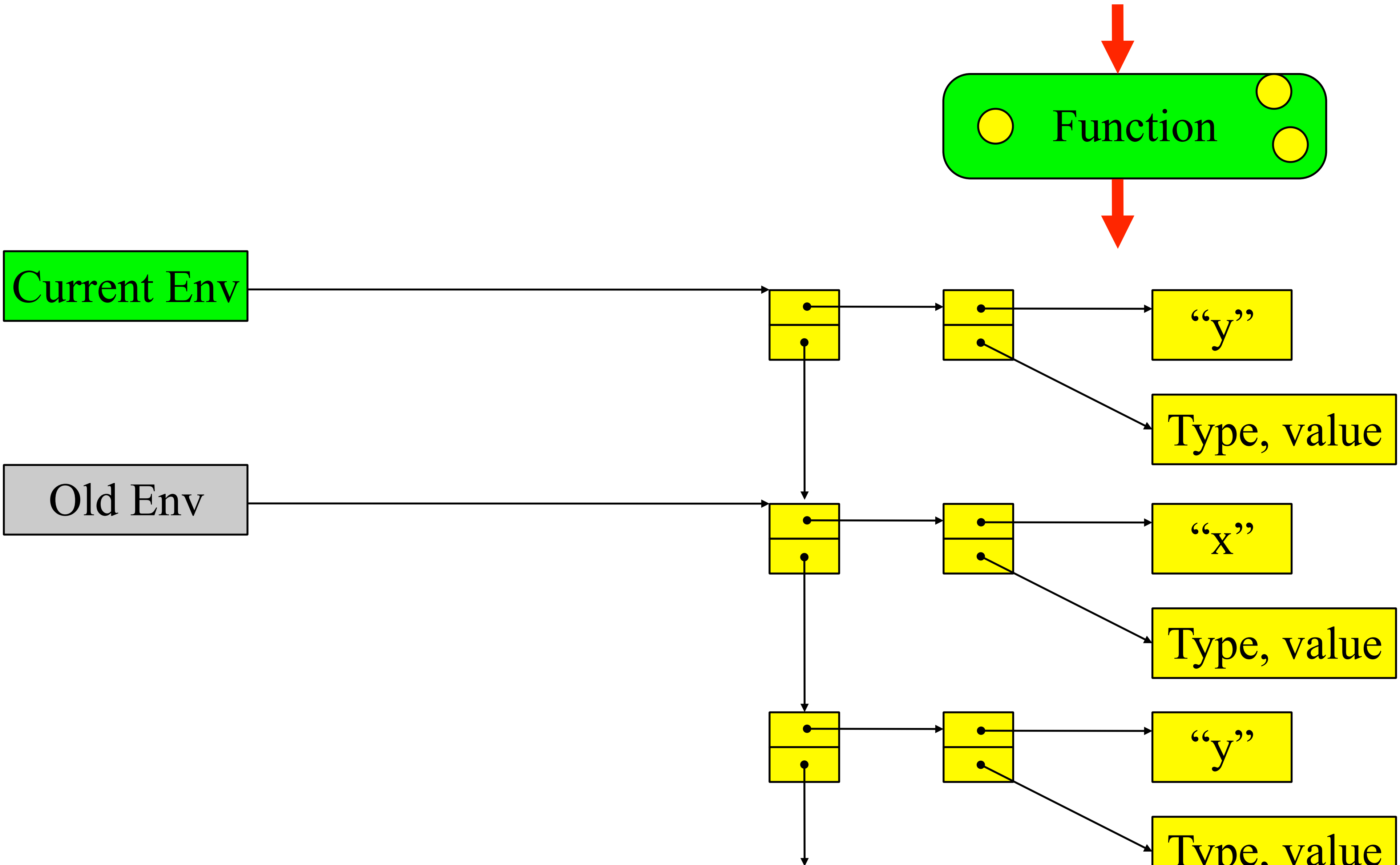
- What It does
  - When we leave the let statement: restore the oldEnv
  - When we need to lookup a value, scan the env.



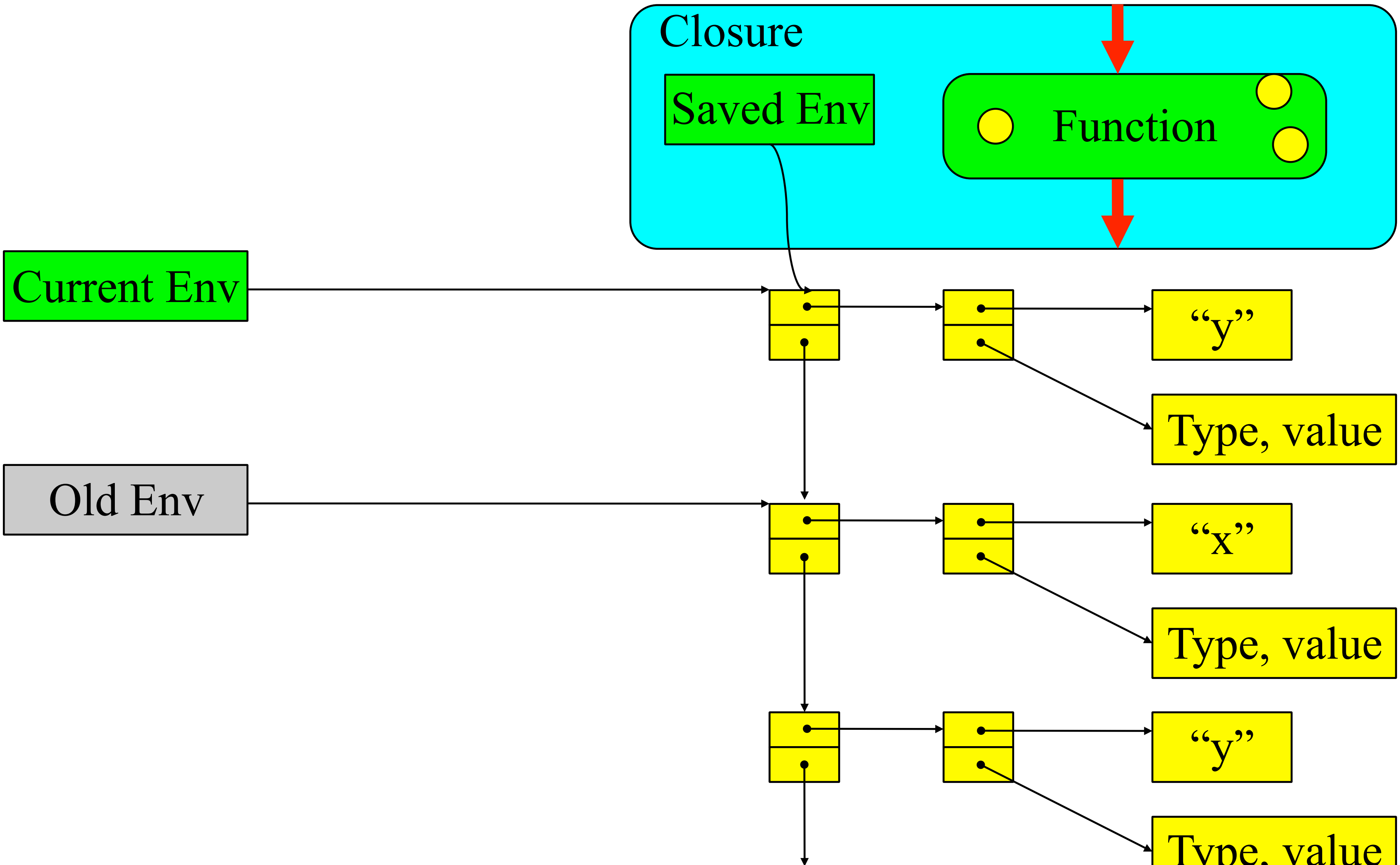
Creating a Closure

- What must be done exactly ?

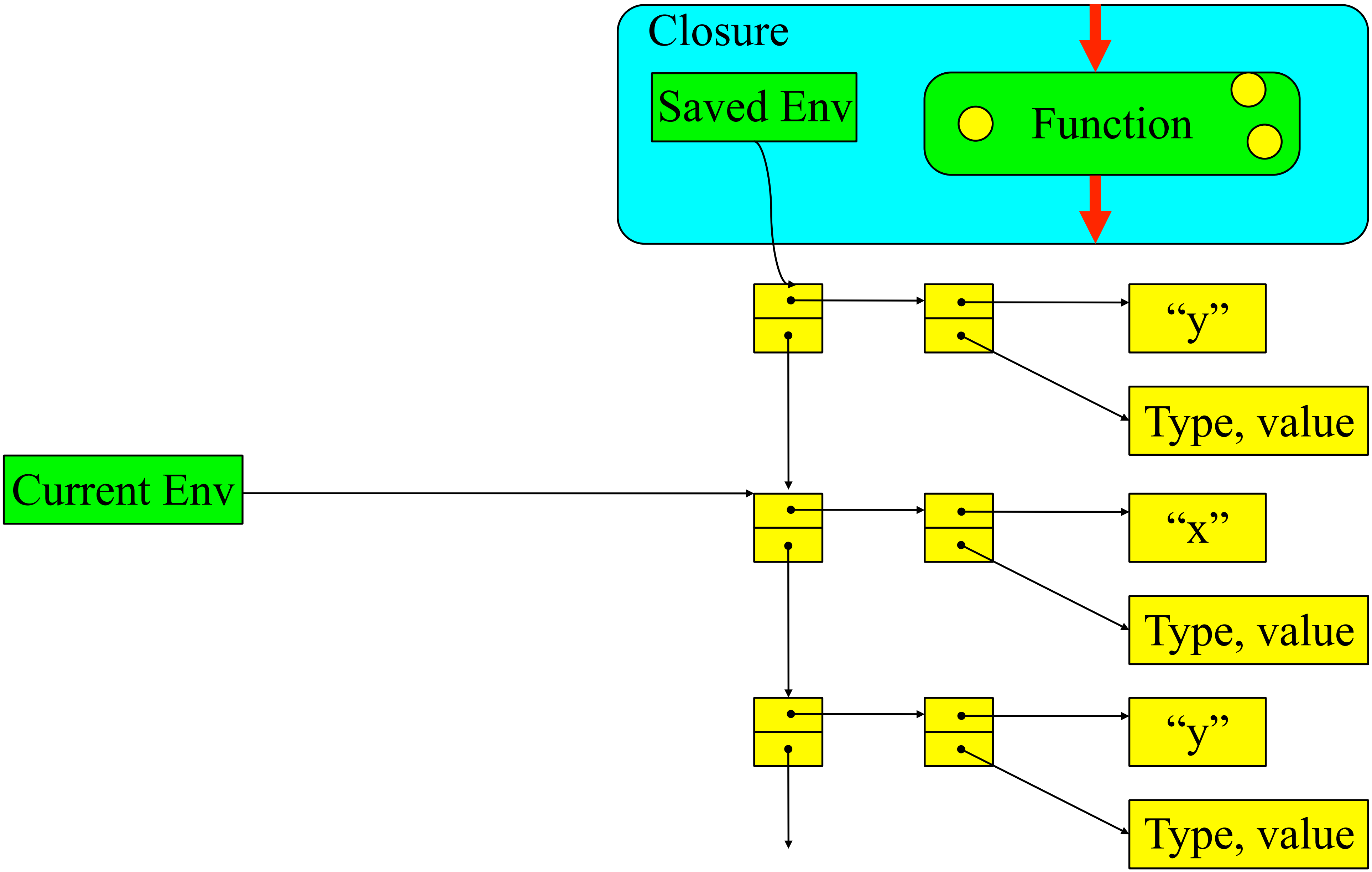
The Function to Close



Closing


Closure
Saved Env
Function
Current Env
Old Env
"y"
Type, value
"x"
Type, value
"y"
Type, value

Leaving the Let



Association List

- Any downside ?

Applications

- Are closures really *that* useful ?
  - Their value is in the eye of the beholder
  - With functional languages
    - You use closures without even noticing!
- Applications
  - Reactive Programming, event handlers, GUI
  - Flexible control flow
    - Delayed evaluation of statements/blocks
    - Execution in context
    - Lazy computation
    - Parallel computation (body of loops! — GCD —)

Closures in Mainstream

- Closures (or close approximation to closures)
  - In Java
    - Event Handlers. Anonymous classes. Native in JDK1.8
  - In C#
    - Anonymous blocks
    - Syntactic sugar. Rewritten as Java anonymous classes
  - In Python
    - Lambdas: Python SML

```
lambda x : x + 1
```

```
fn x => x + 1
```

  - In C++
    - Lambdas in standard! (C++0x11)

```
[=](int x) -> int { return x + 1;}
```

Summary

- Closures
  - Allow lazy/delayed evaluation
  - Modulate the control flow
  - Extend classic functions naturally
  - Doable in functional, imperative, O.O. Languages
    - Language must provide the support though!
  - Moving (increasingly fast) in the mainstream
    - Java,C++,C#,Python,Rust,Nim,Go,Ruby,….
  - Limitation
    - Deal only with a single function
    - No major alteration to control-flow

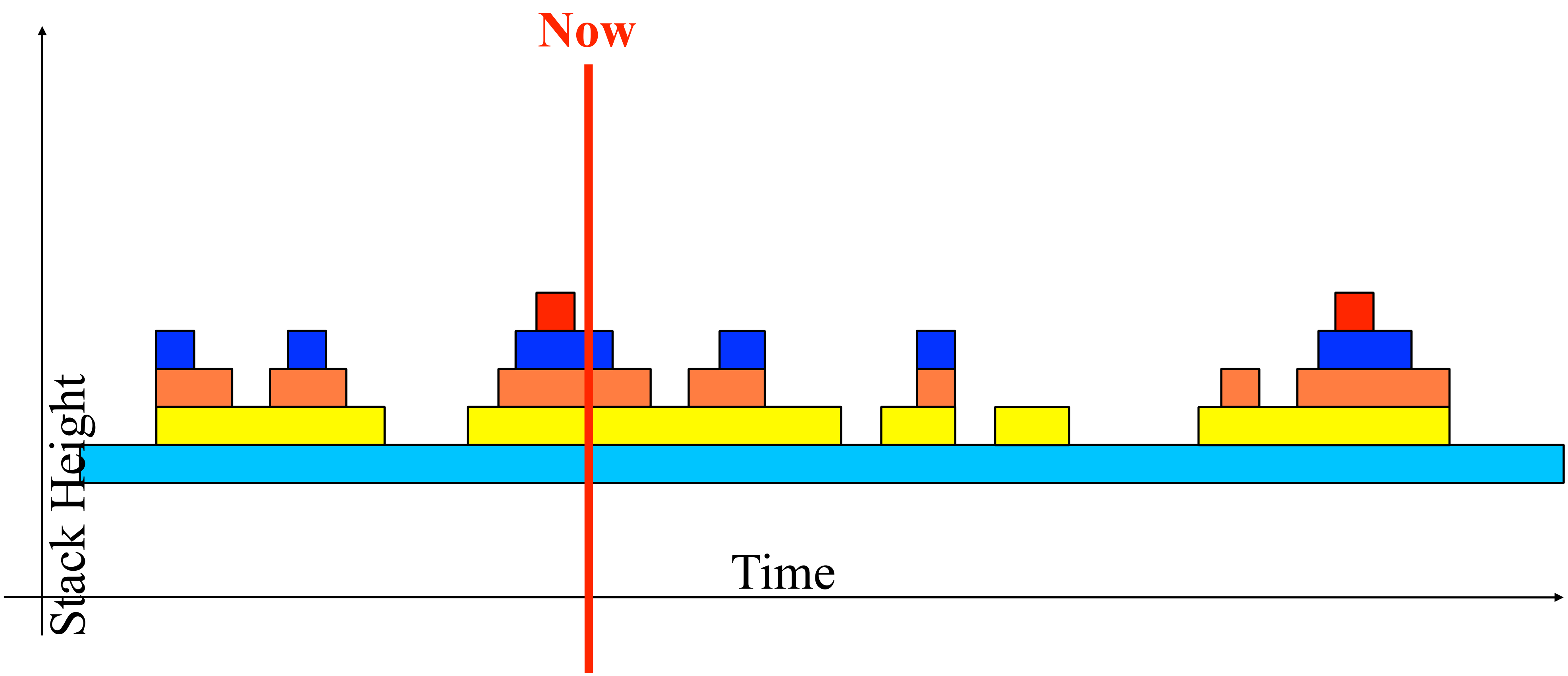
Overview

Continuations

- Definition
- Application
  - Compilation
  - First Class Objects
- Examples

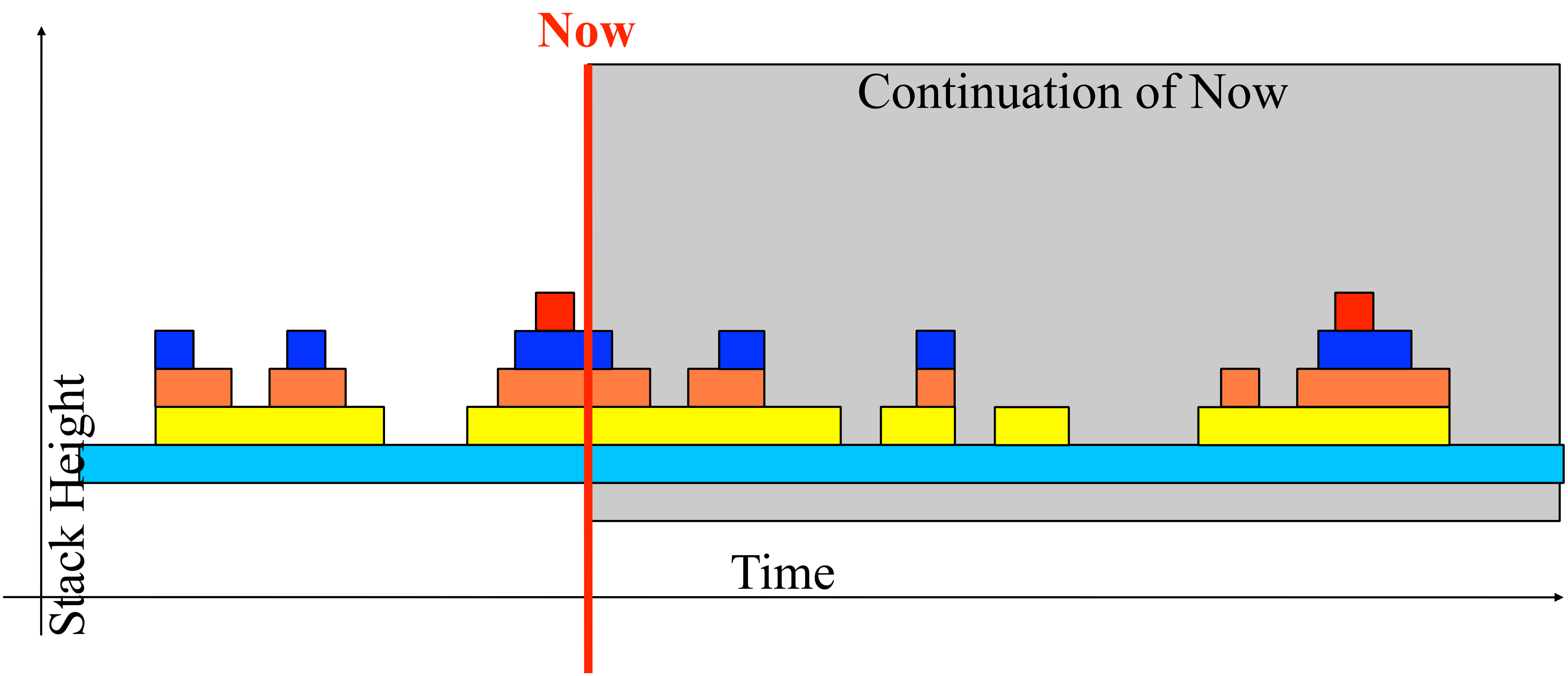
Definition

- What is a continuation ?
  - Informally
    - What is left to do from *now* to complete the computation.



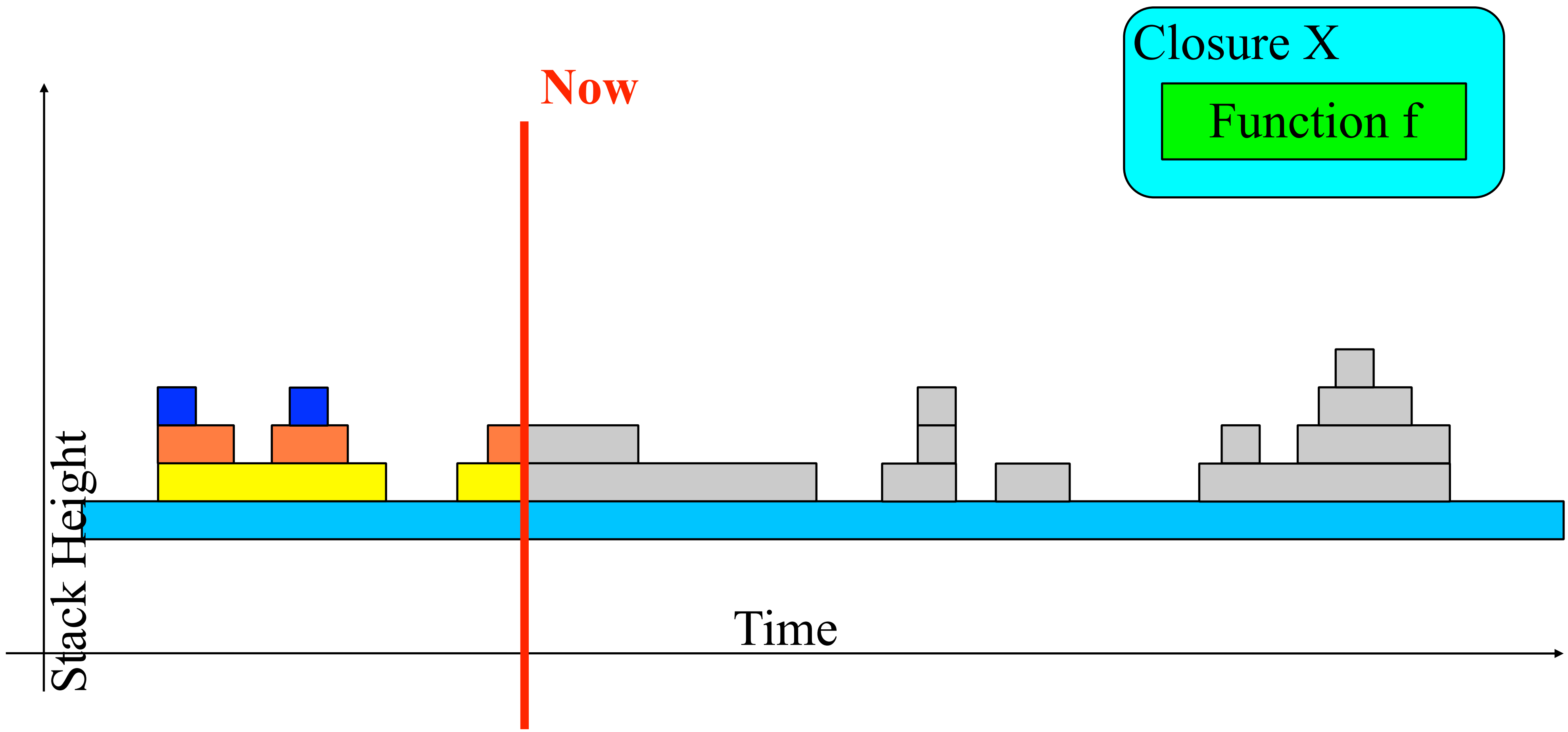
Definition

- What is a continuation ?
  - Informally
    - What is left to do from *now* to complete the computation.



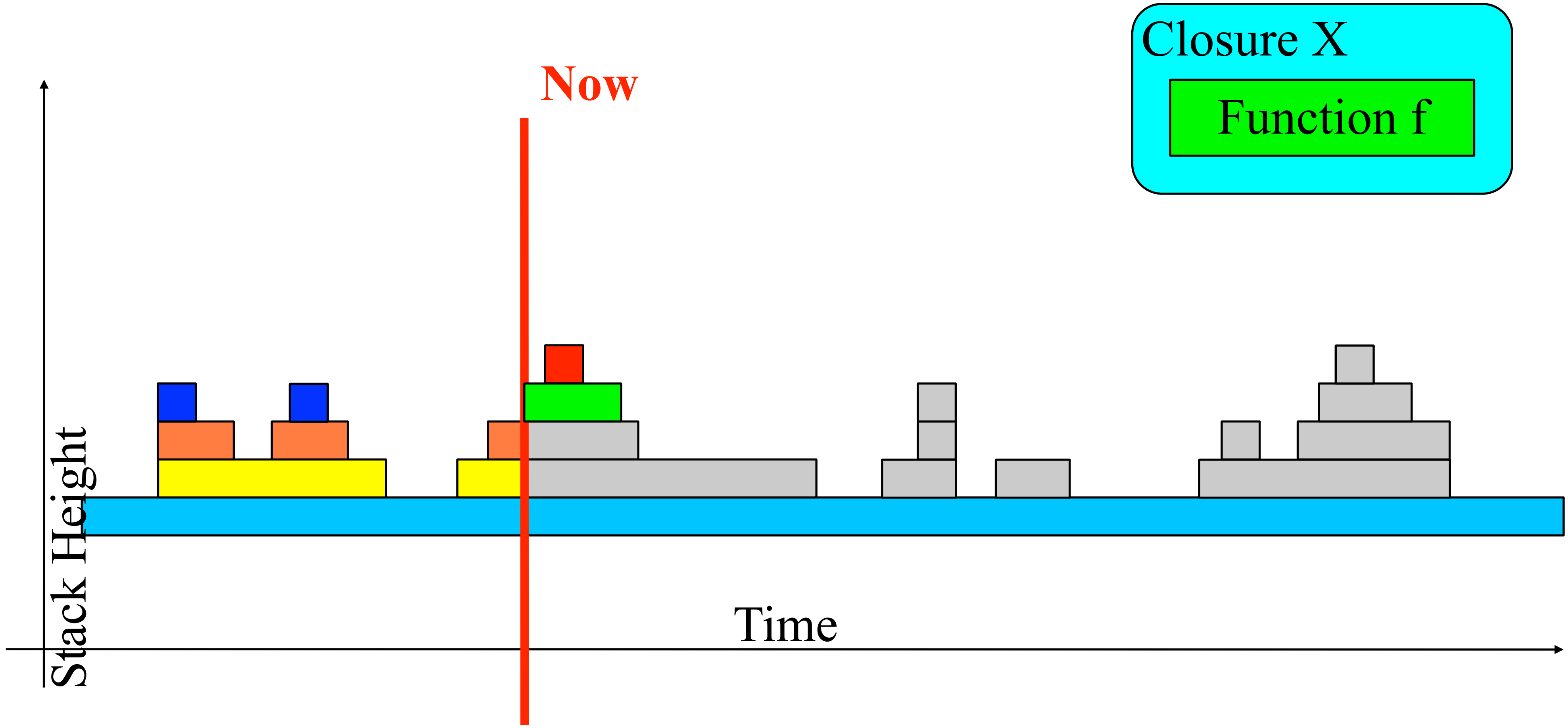
Continuation vs. Closure

- Continuation is
  - The future of the computation.
- What is the difference with a closure ?



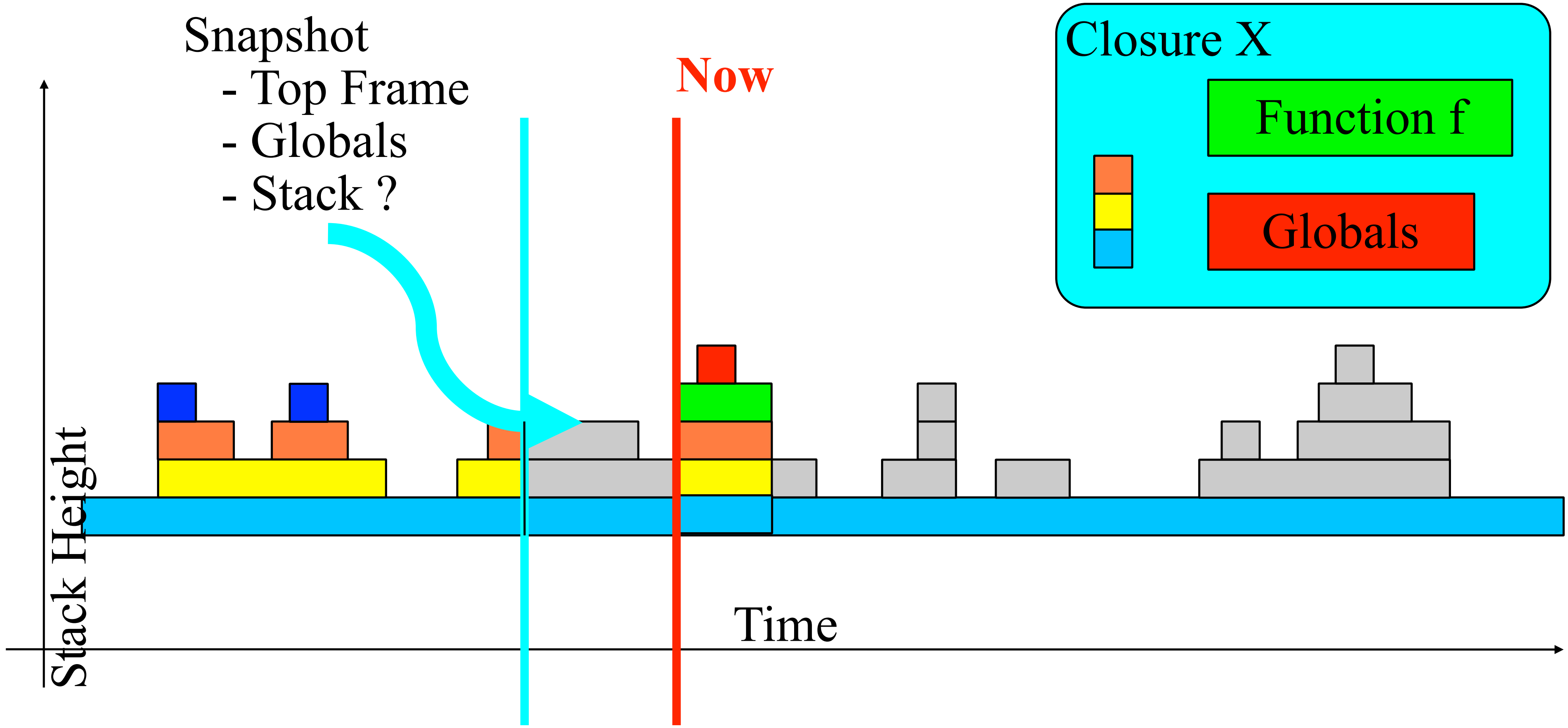
Continuation vs. Closure

- Picture of
  - Calling function f right now
    - Function f executes in the current context



Continuation vs. Closure

- Picture of
  - Creating closure X right now
    - Closure X capture the current environment : Snapshot!



Difference

- A Continuation captures
  - The whole future
- A closure captures
  - A snapshot of the context
  - Allow us to execute a function in context
  - It does not manipulate the future.

Content of a Continuation

- What must be saved exactly ?
- Answer
  - Everything necessary to resume execution
    - Stack
    - Processor state
      - Stack pointers
      - Registers
      - PI-counter
    - Runtime state
      - Additional info such as
        - Pending events

Application

- What can we do with continuations ?

- Two Application areas
  - Compilation technique
    - Many control structure can be described in terms of continuations
    - A method to rewrite programs and eliminate the stack altogether
  - First class objects to
    - Manipulate your future
    - Control the control-flow

Control Primitives

- Consider a few primitives
  - Non-local exit [aborts]
  - Goto
  - setjmp/longjmp
  - Exceptions

Non-local Exit

- Purpose
  - Abort a computation when an error is detected
- The continuation view
  - The program has two possible future
    - Carrying on with whatever it is doing
    - Stopping
  - The abort primitive
    - Replaces the default continuation [carry on] by
      - The continuation that stops.
- Bottom line
  - We must be able to override the current continuation

Goto

- Purpose
  - Transfer control from the current point to a labeled instruction already seen

```
Lbl : x = x + 1
      if B
      then
          <body>
          jump Lbl
      endif
      <goon>
```

- The continuation view
  - When seeing a label
    - Save the current continuation into a table indexed by Lbl.
  - When seeing a jump
    - Lookup the continuation table at lbl and transfer control to that continuation
  - Jump could be non local (from a nested function call)
- Any catch ?

setjmp/longjmp

- Purpose
  - C functions that achieve non-local jumps
  - Setjmp creates a “label”
  - Longjmp jumps to the specified “label”
- Continuation view
  - Same as goto really.
  - Again
    - Only save the control
    - Only save the stack pointer
- Catch?
  - The stack can only shrink.

Exceptions

- Purpose
  - Transfer control non-locally on error detection
  - Try – Catch construction to specify
    - The guarded block, the exception handler
  - Throw construction to trigger an exception
- Required data structure
  - A table indexed by exception types
    - As many entries as types of exceptions
    - Content of an entry
      - Closure
      - Continuation

Exceptions

- Continuation view
  - Entering a try-catch
    - Capture a closure for the exception handler
    - Capture the current continuation
    - Save the content of exception table at the exception offset
    - Save (Closure,Continuation) in the table (at correct offset)
  - When throwing
    - Execute the closure handler
    - Transfer control to saved continuation
  - When leaving a try-catch block (normally)
    - Restore the table entry for the exception type to its old value
      - Old Handler
      - Old Continuation

Program Rewrites

- Continuation can be helpful to rewrite programs
- Objective of rewrite
  - Eliminate the stack
  - Make programs tail-recursive
  - Allow optimizations
- Technique name
  - Continuation Passing Style or CPS
- More on this a little later....

Overview

- Definition
- Application
  - Compilation
  - First Class Objects
- Examples

First Class Object

- Continuation are useful inside the compiler
- Continuation are also useful for end-user
  - It gives control over the control-flow of the program
- Primitives
  - Two key abstraction
    - call/cc
      - Captures the current continuation
    - throw
      - Transfer control to a specified continuation
- From now on
  - Study continuation in functional language (ML)
  - Continuations are functions
  - Continuation take one argument

Basic Example

- Motivation
  - A starting example

```
fun bar x = print ("bar:"  ^ (Int.toString x) ^ "\n")
fun zoo x = print ("zoo:"  ^ (Int.toString x) ^ "\n")

fun test a b  = a orelse b
fun foo a b x = if test a b
                   then bar x
                   else zoo x

val z = foo true false 5
```

Output => “bar: 5”

Basic Example

- Motivation
  - Get familiar with CPS

```
fun test a b  = a orelse b
fun foo a b x = if test a b
                   then bar x
                   else zoo x
val z = foo true false 5

fun test' a b k = k (a orelse b)
fun foo' a b x  = let fun f b = if b
                                then bar x
                                else zoo x
                     in test' a b f
                  end

val z = foo' true false 5
```

Output => “bar: 5”

Basic Example With call/cc

- Motivation
  - Get familiar with call/cc

```
fun test a b  = a orelse b
fun foo a b x = if test a b
                   then bar x
                   else zoo x
val z = foo true false 5

val callcc = SMLofNJ.Cont.callcc
val throw  = SMLofNJ.Cont.throw
type 'a cont  = 'a SMLofNJ.Cont.cont

fun test'' a b  = (callcc (fn k => throw k (a orelse b)))
fun foo'' a b x =  if test'' a b
                   then bar x
                   else zoo x
val z = foo'' true false 5
```

What we did...

- It doesn't look like much....
- But
  - We captured the continuation and bound it to k
    - Done with callcc
  - We can now manipulate k
    - Save it
    - Wrap it in a closure
    - Alter its argument
    - Call it
  - We finally called k with the original argument
    - Done with throw

Saving & Using Continuations

- How can we save a continuation...
  - Can we bind it to a local variable ?

References

- ML offers destructive variables
  - So we finally see assignments
- ML destructive variables are references
  - A reference is a pointer to the real object
  - Ref keyword to create a reference
  - := operator to modify the content of a reference
  - ! to lookup the content of a reference.

```
val int x = 10;

val int ref x = ref 10;

x := 20;

val z = 2 * !x
```

Reference and Continuation

- If we want to save a continuation....
  - Store it in a reference!
- A catch
  - A reference must always be initialized
  - What should we store at declaration time ?
- Solution
  - Wrap the continuation in a data type with 2 alternatives
    - Empty
    - A real continuation

Reference and Continuations

```
val callcc = SMLofNJ.Cont.callcc
val throw  = SMLofNJ.Cont.throw
type 'a cont  = 'a SMLofNJ.Cont.cont
exception Bad of string
datatype 'a SCont = Null
                  | Cont of 'a cont

fun Sthrow (Null)     v = raise Bad("No continuation")
  | Sthrow (Cont a)   v = throw a v

val myc : int SCont ref = ref Null
```

The variable myc holds a reference to a datastructure that could be null, or a continuation. The continuation stored in myc takes an integer as input.

Second Example

- Factorial... revisited

```
exception Bad of string
datatype 'a SCont = Null
                  | Cont of 'a cont
fun Sthrow (Null)     v = raise Bad("No continuation")
  | Sthrow (Cont a)   v = throw a v
val myc : int SCont ref = ref Null

fun fact 0 = (callcc (fn k => (myc := Cont k;1)))
  | fact n = let val _   = print("Going down (n= " ^
                                 (Int.toString n)  ^ ")\n")
             in n * fact(n-1)
             end;

let val y = fact 5
in
    print("y = " ^ (Int.toString y) ^ "\n");
    Sthrow (!myc) 1
end
```