

# Call by Reference

```
public void swap(ref int x, ref int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

```
int a = 100;  
int b = 200;
```

```
Console.WriteLine("Before swap, value of a : {0}", a);  
Console.WriteLine("Before swap, value of b : {0}", b);
```

```
n.swap(ref a, ref b);
```

```
Console.WriteLine("After swap, value of a : {0}", a);  
Console.WriteLine("After swap, value of b : {0}", b);
```

# Passing Reference Types by Value

```
static void Change(int[] pArray) {  
    pArray[0] = 888; // This change affects the original element.  
    pArray = new int[5] {-3, -1, -2, -3, -4}; // This change is local.  
    System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);  
}  
  
static void Main() {  
    int[] arr = {1, 4, 5};  
    System.Console.WriteLine("Inside Main, before calling, the first element is: {0}", arr[0]);  
  
    Change(arr);  
    System.Console.WriteLine("Inside Main, after calling, the first element is: {0}", arr[0]);  
}
```

/\* Output:

Inside Main, before calling the method, the first element is: 1

Inside the method, the first element is: -3

Inside Main, after calling the method, the first element is: 888

# Passing Reference Types by Reference

```
static void Change(int[] pArray) {  
    // Both of the following changes will affect the original variables:  
    pArray[0] = 888;  
    pArray = new int[5] {-3, -1, -2, -3, -4};  
    System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);  
}  
  
static void Main() {  
    int[] arr = {1, 4, 5};  
    System.Console.WriteLine("Inside Main, before calling, the first element is: {0}", arr[0]);  
  
    Change(ref arr);  
    System.Console.WriteLine("Inside Main, after calling, the first element is: {0}", arr[0]);  
}
```

/\* Output:

Inside Main, before calling the method, the first element is: 1

Inside the method, the first element is: -3

Inside Main, after calling the method, the first element is: -3

# Concepts of Programming Languages

## Tail-Calls and Continuations

Prof. Dr. Guido Salvaneschi

# Outline

- Recursion, Iteration and Tail Call Optimization
- Short Journey in Continuation Passing Style (CPS)
- Continuations as First-Class Values in the Language
- Implementing Continuations

# Recursive Factorial Function

```
def factorial1(n: Int): Int = {  
  if (n == 1) 1  
  else n * factorial1(n - 1)  
}  
factorial1(6)
```

```
6 * factorial1(5)  
6 * 5 * factorial1(4)  
6 * 5 * 4 * factorial1(3)  
6 * 5 * 4 * 3 * factorial1(2)  
6 * 5 * 4 * 3 * 2 * factorial1(1)  
6 * 5 * 4 * 3 * 2 * 1  
6 * 5 * 4 * 3 * 2  
6 * 5 * 4 * 6  
6 * 5 * 24  
6 * 120  
720
```

Stack frame of  
function callers are  
stored and restored

# A More Iterative Factorial Function

```
def factIter(n: Int, result: Int): Int = {  
  if (n <= 1) result  
  else factIter(n - 1, result * n)  
}  
def factorial2(n: Int): Int = factIter(n, 1)  
factorial2(6)
```

```
factIter(6, 1)  
factIter(5, 6)  
factIter(4, 30)  
factIter(3, 120)  
factIter(2, 360)  
factIter(1, 720)
```

```
return factIter(6, 1)  
return (return factIter(5, 6))  
return (return (return factIter(4, 30))  
return (return (return (return factIter(3, 120))  
return (return (return (return (return factIter(2, 360))  
return (return (return (return (return (return factIter(1, 720))  
return (return (return (return 720)  
return (return (return 720))  
return (return 720)  
return 720  
720
```

# A More Iterative Factorial Function

- Nothing interesting happens after function returns
- Actually, **no need to store and restore stack frame** of callers
- How can we detect and support such optimization?

```
factIter(6, 1)
factIter(5, 6)
factIter(4, 30)
factIter(3, 120)
factIter(2, 360)
factIter(1, 720)
```

```
return factIter(6, 1)
return (return factIter(5, 6))
return (return (return factIter(4, 30)))
return (return (return (return factIter(3, 120))))
return (return (return (return (return factIter(2, 360)))))
return (return (return (return (return (return factIter(1, 720)))))
return (return (return (return 720)))
return (return (return 720))
return 720
720
```



# Comparison

- **factorial1** :
  - + Code structure reflects mathematical definition
  - Final result depends on caller, hence we need a stack
- **factorial2**:
  - + Final result only depends on callee
  - More complex, less elegant definition (accidental complexity)

# Comparison

```
def factorial1(n: Int): Int = {  
  if (n == 1) 1  
  else n * factorial1(n - 1)  
}
```

Needs to return to caller; it needs the result to complete

```
def factIter(n: Int, result: Int): Int = {  
  if (n <= 1) result  
  else factIter(n - 1, result * n)  
}  
def factorial2(n: Int): Int = factIter(n, 1)
```

Captures the rest of computation and everything needed to complete it. No need to return. **Tail Call**

# Tail Call Informally

A call from a function  $f$  to a function  $g$  is a **tail call** if:

In the control graph of  $f$ , the call of  $g$  is a leaf.

That is,  $f$  simply returns the result of calling  $g$ .

Special case: **tail recursion**

A tail call from a function to itself.

# Tail Call Optimization (TCO)

A call from a function  $f$  to a function  $g$  is a **tail call** if:

In the control graph of  $f$ , the call of  $g$  is a leaf.

That is,  $f$  simply returns the result of calling  $g$ .

Special case: **tail recursion**

A tail call from a function to itself.

**Tail-call optimization:**

$g$  sends its result directly to whoever is expecting  $f$ 's result

→ calling  $g$  does not require any stack manipulation.

# Tail Call Optimization (TCO)

## Tail-call optimization (TCO):

**g** sends its result directly to whoever is expecting **f**'s result  
→ calling **g** does not require any stack manipulation.

- Tail calls do not return to the caller.
- No stack frames need to be stored/restored for tail calls.
- Compilers apply TCO to replace calls in a tail position with **jump** statements → save stack space

# Consequences of TCO

**Function calls become inexpensive,  
same performance as inlined code**

**Recursive functions run on  
constant amount of stack space**

**→ Can use functions and recursion without  
stack-space or run-time penalty**

# Consequences of TCO

- **Built-in looping constructs not necessary anymore.**
  - Whatever was previously written using a built-in loop can be written using tail recursion.
  - Compiler will convert the calls into gotos achieving the same efficiency as the loop version.
- Having custom loop constructs be as efficient as build-in loops is especially interesting for new data structures.

# Support for TCO in Programming Languages

- TCO is traditionally supported in functional languages like Scheme and ML.
- But, any other language can have TCO as well!
- Yet, mainstream languages, such as Java, do not support TCO (this is almost true also for C#).
- Follow the web links for more details.
  - <http://debasishg.blogspot.com/2006/03/non-java-languages-on-jvm.html>
  - <http://www.infoworld.com/print/21437>
  - <http://lamp.epfl.ch/~odersky/papers/babel01.html>



# Support for TCO in Programming Languages

- Some compilers support only tail recursion optimization.
- Tail recursion is an important special case, but not the only useful one.

# Have the Cake and Eat it Too?

- Functions with tail calls are more efficient
  - but can be more complex to write (recall e.g., the two definitions of factorial).
  - Can we have both clarity of definitions and efficiency?
- “Traditional code” can be transformed automatically into a form where **every call is a tail call!**
  - Such a transformation is said to bring a function into **Continuation Passing Style (CPS)**.

# Outline

- Recursion, Iteration and Tail Call Optimization
- Short Journey in Continuation Passing Style (CPS)
- Continuations as First-Class Values in the Language
- Implementing Continuations

# Continuation Passing Style (CPS)

CPS as another semantics of function calls ...

# CPS as a Function Call Semantics

function  $f$  calls function  $g$  ...

## Traditional semantics

- $f$  passes actual arguments to  $g$
- $f$  waits for  $g$  to return (eventually with a result)
- $f$  resumes its computation eventually using the result of  $g$

## CPS Semantics

- In addition to arguments,  $f$  passes to  $g$  a function of one parameter, to which  $g$  should pass its result.  
This is the **continuation**.
- The **continuation** captures the **rest-of-computation** at the point of making the call.
- Called function does not return, but calls the continuation.

# Factorial in CPS

- Let's try to derive the CPS version of **factorial** ...

```
def factorial1(n: Int): Int = {  
  if (n == 1) 1  
  else n * factorial1(n - 1)  
}
```

- First, top-level factorial is called from somewhere with some continuation **k** ...

```
def factCPS(n: Int, k: Int => Nothing) {  
  ???  
}
```

# Factorial in CPS

Questions to answer:

- What should `factCPS` do in the case `n = 1`?
- What's the rest of computation at the point of the recursive call?

```
def factCPS(n: Int, k: Int => Nothing) {  
    ???  
}
```

# Factorial in CPS

- What should `factCPS` do in the case `n = 1`?
- What's the rest of computation at this point?

Instead of returning, pass the result to the continuation:

```
def factCPS(n: Int, k: Int => Nothing) {  
  if (n == 1) k(1)  
  else ???  
}
```



# Factorial in CPS

What's the rest of computation at the recursive call?

```
def factCPS(n: Int, k: Int => Nothing) {  
  if (n == 1) k(1)  
  else factCPS(???)  
}
```

Rest-of-computation informally: multiply the result of the recursive call, say **restResult**, with **n** and return the result of this multiplication (i.e, pass to **k**)

```
restResult => k(n * restResult)
```

# Factorial in CPS

If **n** is not **1**, **factCPS** simply calls itself, passing the new continuation as a parameter.

```
def factCPS(n: Int, k: Int => Nothing) {  
  if (n == 0) k(1)  
  else factCPS(n - 1, { restResult => k(n * restResult) })  
}
```

# Systematic Transformation to CPS

- Any program can **automatically transformed** into a behaviorally equivalent version in **CPS**.
- Such transformation is called **CPS transformation**.
- See PLAI for details.

# Outline

- Recursion, Iteration and Tail Call Optimization
- Short Journey in Continuation Passing Style (CPS)
- Continuations as First-Class Values in the Language
- Implementing Continuations

# CPS Has its Problems ...

1. Requires access to the entire program: Sometimes we need to transform functions that are in a library, e.g., `map`.
2. Inhibits built-in compiler optimizations by replacing the stack with explicit representation of continuations
3. Assumes that the language has TCO to avoid the creation of needless stack frames.

Many languages, e.g., Java, C, do this anyway; the program consumes memory unnecessarily.

# First-Class Continuations

- Some PLs provide an operation that reifies and returns the **“rest of the computation”** at any program point.
- Examples: Scheme; Smalltalk; Ruby, Standard ML, Scala.

**In contrast to CPS:** Program does not need to be transformed a priori. Reifications of rest-of-computation can be done on-the-fly.

# First-Class Continuations

- In functional languages (including Scala)
  - “rest of the computation” is represented as a function of one argument called a **continuation**.
  - Computation can be resumed by applying it to a value.
- In Smalltalk (OO language),
  - The “rest of computation” is represented as an object of type **Context**.
  - Computation resumed by calling method **resume** on that object.

# Continuations in Scala

- Scala provides the **shift** operator to capture the continuation at the point of its evaluation.
- Scala's continuations capture the rest of the execution up to a region explicitly specified by the programmer via **reset**
- The **shift** function captures the remaining computation in a **reset** block and passes it to a closure provided by the user.

```
reset {  
  shift { k: (Int => Int) => k(3) } + 5  
}
```

What is the value of this program?



# Escapers

What's the continuation  
result of the computation-  
lambda at shift?

What's the result of the  
expression?

```
reset {  
  1 + shift { k: (Int => Int) => k(3) }  
}
```

Is this the  
continuation?

```
(v: Int) => 1 + v
```

# Escapers

```
1 + k(3)
= 1 + ((v: Int) => 1 + v)(3)
= 1 + (1 + 3)
= 5
```

```
k = (v: Int) => 1 + v
```

- **Problem:**
  - We apply the “rest-of-computation” twice.
  - The computation should halt after applying **k**.

# Escapers

- We will symbolically use the notation  $\Rightarrow \uparrow$  to represent a lambda that halts computation after execution of its body  $\rightarrow$  ESCAPERS.
- Then continuations are bound to ESCAPERS.

```
reset {  
  1 + shift { k: (Int => Int) => k(3) }  
}
```

```
k = (v: Int) =>! 1 + v
```

- Let's try to substitute again

```
1 + k(3)  
= 1 + ((v: Int) =>! 1 + v)(3)  
= ((v: Int) =>! 1 + v)(3)  
= 1 + 3  
= 4
```

# Some More Examples

What will be printed on the console?

```
def foo() = shift{k: (Int => Int) => k(7) + 1}  
println(reset{2 * foo()})
```

```
println(reset {shift { k: (Int=>Int) => k(k(k(7))) } + 1 } * 2)
```

```
println(reset {shift { k: (Int=>Int) => k(k(k(7))); "done" } + 1})
```

```
def foo() = { 1 + shift{k: (Int=>Int) => k(k(k(7)))}}  
def bar() = {foo() * 2 }  
def baz() = { reset{bar() + 10} }  
println(baz())
```

# Continuations in Real Web Programming

- Continuation-based Web servers have gained popularity:
  - Seaside Web Server
  - PLT Scheme Web Server
  - UnCommon Web Framework
  - Apache Cocoon Web application framework also provides continuations (see the Cocoon manual)
- No consensus yet as to whether continuations are really beneficial for Web programming:
  - <http://blogs.sun.com/gbracha/>
  - <http://www.interact-sw.co.uk/iangblog/2006/05/21/webcontinuations>

# Outline

- Recursion, Iteration and Tail Call Optimization
- Short Journey in Continuation Passing Style (CPS)
- CPS and Web Programming
- Continuations as First-Class Values in the Language
- Implementing Continuations

# Implementing Continuations

Any idea as how we can  
implement continuations?

# Implementing Continuations

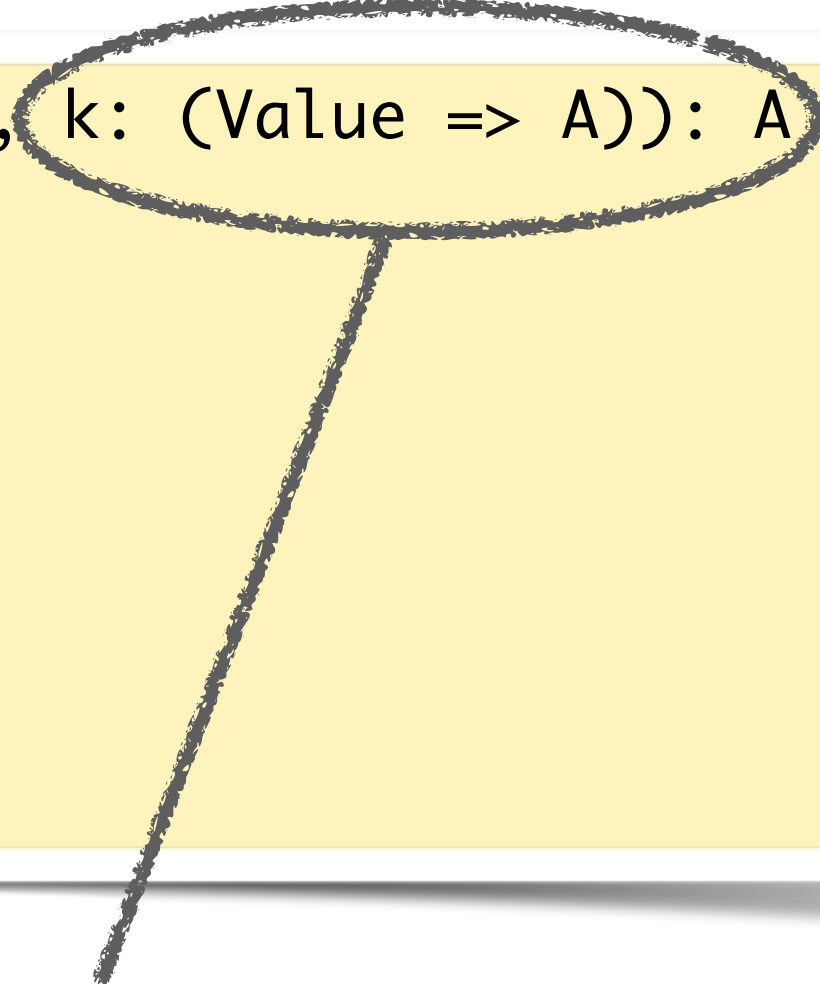
- The interpreter program can also be converted to CPS.
- In a CPS interpreter the current continuation is available at each stage of the interpretation and can be easily provided to programmer, if required.
- **General approach to implementing continuations**
  1. Make them explicit in the interpreter.
  2. Provide access to them in an extended language.



# CFAE Interpreter in CPS

- Will turn the CFWAE interpreter into CPS:

```
def interp[A](expr: KCFWAE, env: Env, k: (Value => A)): A = {  
  ...  
  expr match {  
    case ...  
    ...  
    ...  
  }  
}
```



Continuation expects the result of each expression's interpretation.

1. If interpreter has a value handy, supply it to continuation.
2. Otherwise, interpreter calls itself with a (possibly augmented) continuation.

# The Case of Number, Identifiers, Closures

The simplest cases: What should we do here?

```
...  
case Num(n) => ???  
case Id(name) => ???  
case Fun(arg, body) => ???  
...
```

# The Case of Number, Identifiers, Closures

Numbers and identifiers are simply fed to continuation.

```
...  
case Num(n) => k (NumV(n))  
case Id(name) => k (env(name))  
case Fun(arg, body) => k (Closure(arg, body, env))  
...
```

# CPS Interpretation of Non-Value Expressions

In all other cases:

**Further steps of interpretation** are performed by recursive calls **with a new augmented continuation**.

All pending computation must be moved to the continuation!

**Rule of thumb:**

A call to `interp` never appears as a sub-expression in the `interp` function definition. It is always the first thing we do...

# CPS Interpretation of Arithmetic Operations



What needs to be done?



# Interpretation of Arithmetic OPs

```
case Add(lhs,rhs) =>  
  interp(lhs, env,  
    lv => {  
      interp(rhs, env,  
        rv => (lv, rv) match {  
          case (NumV(n1), NumV(n2)) => k(NumV(n1 + n2))  
        })  
    })  
  })
```



Augmented continuation for  
interpreting the left-hand operand

# CPS Interpretation of Conditionals

```
case If0(c, t, e) =>
{
  interp(c, env, (cv) =>
    cv match {
      case NumV(0) => interp(t, env, )
      case _       => interp(e, env, )
    })
}
```

- Do we need to augment the continuation for interpreting **then** and **else** expressions?
- No: The pending computation after evaluating any of the branches is the same as the one of the **If0** expression.

# CPS Interpretation of Application

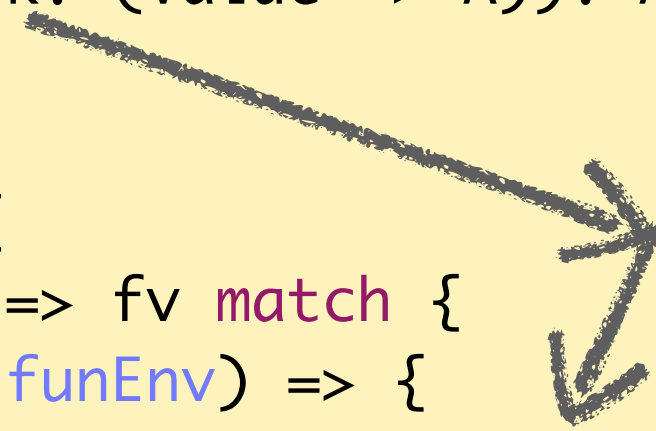


What are the steps?



# CPS Interpretation of Application

```
def interp[A](expr: KCFWAE, env: Env, k: (Value => A)): A = {  
  ...  
  case App(funExpr, argExpr) => {  
    interp(funExpr, env, (fv) => {  
      interp(argExpr, env, (argV) => fv match {  
        case Closure(param, body, funEnv) => {  
          interp(body, funEnv + (param -> argV), k)  
        }  
        case _ => {  
          sys.error("Can only apply closures but got %s")  
        }  
      })  
    })  
  }  
}
```



continuation for interpreting  
the application is the  
dynamic continuation for  
interpreting body of the  
closure

# Adding Continuations to the Language

- **BindCC** corresponds to **let/cc** in Scheme or non-delimited shift in Scala

```
<KCFAE> ::= ...  
          | {<KCFAE> <KCFAE>}  
          | {BindCC <id> <KCFAE>}
```

- The language has a new value type for **first-class** continuations.
- Represented by Scala functions.
- We will overload the procedure application to handle continuation application.

```
sealed abstract class Value  
case class NumV(n: Int) extends Value  
case class Closure(param: Symbol, body: KCFWAE, env: Env) extends Value  
case class Continuation(c: (Value => A)) extends Value
```

# Interpreting BindCC

What needs to be done in this case?

```
case BindCC(name, body) => ???
```

# Interpreting BindCC

Interpret **body** in the environment at continuation reification time extended with a binding for **name**

What should we put  
as the value of the  
continuation?

```
case BindCC(name, body) =>  
  interp(body, env + (name -> ??? ), k )
```

The continuation  
for interpreting the  
body is **k**

# Interpreting BindCC

```
case BindCC(name, body) =>  
  interp(body, env + (name -> Continuation(k) ), k)
```

Are we done?

# Interpreting BindCC

```
def interp[A](expr: KCFWAE, env: Env, k: (Value => A)): A = {  
  ...  
  case App(funExpr, argExpr) => {  
    interp(funExpr, env, (fv) => {  
      interp(argExpr, env, (argV) =>  
        fv match {  
          case Closure(param, body, funEnv) => {  
            interp(body, funEnv + (param -> argV), k)  
          }  
          case Continuation(c) => c(argV)  
          case _ => {  
            sys.error("Can only apply closures but got %s")  
          }  
        }  
      })  
    })  
  }  
  ...  
}
```

- In contrast to closures, continuations do not take a dynamic continuation as an argument.
- The whole point about continuations is to ignore the current continuation and use the stored receiver instead.

# Adding Continuations to the Language

- Continuations in the interpreter are similar to the continuations that we need in the language, except for two things.
- Interpreter continuations capture what's left to be done in the interpreter, not in the program.
  - but, the interpreter simulates the execution of the program
  - so, the continuation of the interpreter simulates the corresponding continuation of the program
- **Interpreter continuations are Scala lambda not escapers.**
  - but, our CPS interpreter never returns
  - thus, no need for escapers

# Initial Continuation

- What is the initial value of **k** to pass to the interpreter?
- Theoretically it is “the rest of Scala runtime”
- Practically, we can use the identity function, or a function that prints the received value and terminates.



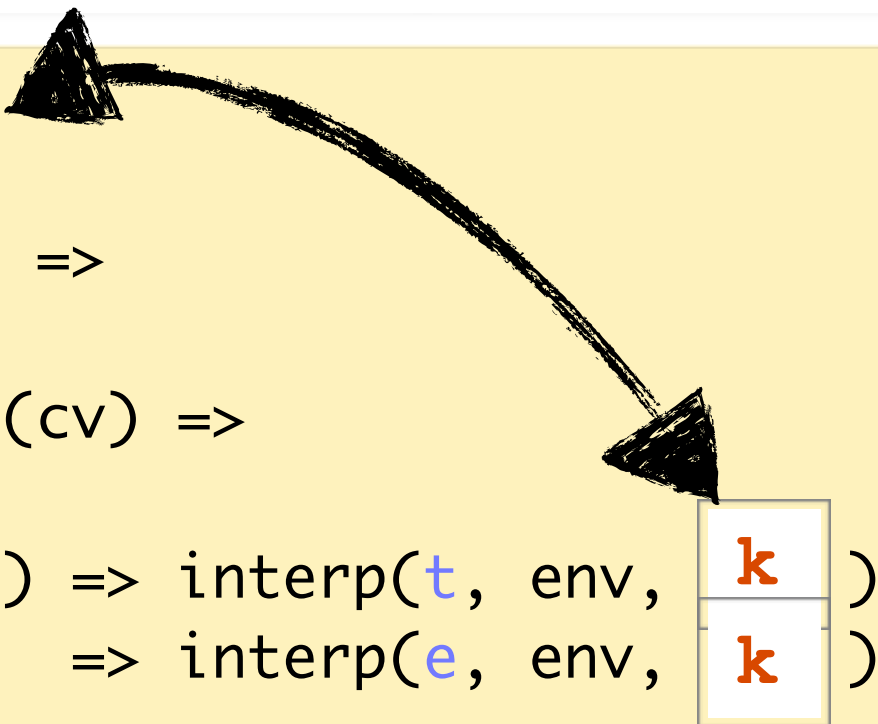
# Tail Calls Revisited

- Converting a function to CPS makes the stack explicit.
- We can see if a function invocation requires something “to be pushed to the stack”.
- If a function invokes another function with the **same continuation** that it received, it doesn't push anything to the stack.
- **Such invocations were tail calls before conversion to CPS.**

# Tail Calls Revisited

We have seen an example that was a tail call before conversion:

```
interp( ... k ...  
  ...  
  case If0(c, t, e) =>  
  {  
    interp(c, env, (cv) =>  
      cv match {  
        case NumV(0) => interp(t, env, k )  
        case _       => interp(e, env, k )  
      })  
  }
```



```
case If0(testExpr, thenExpr, elseExpr) => {  
  val testV = interp(testExpr, funDefs, subRep)  
  if (testV == 0)  
    interp(thenExpr, funDefs, subRep)  
  else  
    interp(elseExpr, funDefs, subRep)  
}
```

# Tail Calls Revisited

- The calls in the implementation of addition have not been tail calls, since they augment the original continuation:

```
case Add(lhs,rhs) =>
  interp(lhs, env,
    lv => {
      interp(rhs, env,
        rv => (lv, rv) match {
          case (NumV(n1), NumV(n2)) => k(NumV(n1 + n2))
        })
      })
  })
```

# Continuations: Summary

- Tail calls and TCO enable the use of functions and recursion without stack-space or run-time penalty.
- But, programs that only contain TCs may have a more complex structure than equivalent programs without TCs.
- Any program can be automatically transformed to a CPS version that only contains tail calls.
- CPS is useful for web programming and allows to implement new control constructs:  
exceptions, backtracking, pattern matching, lazy generators, coroutines

# Continuations: Summary

- To implement continuations:
  - Transform the interpreter to CPS
  - Make continuation available at any point of execution by a special binding construct

# Continuations: Summary

For those of you interested, the following paper, presents how delimited continuations are implemented in Scala:

“Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform”, by T. Rompf, I. Maier, M. Odersky

<https://github.com/scala/scala-continuations>