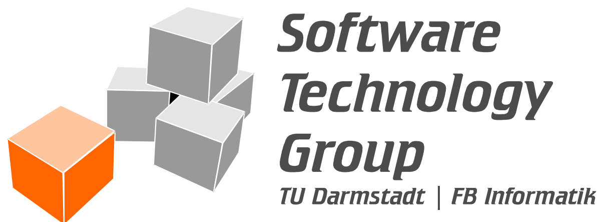


Tail Call Optimization, Continuation Passing Style and Continuations

Prof. Dr. Mira Mezini
MSc. Vaidas Gasiunas
Dipl.-Inform. Karl Klose

Continuations: Mysterious? Frightening?

- Not really,… for a moment just **functions that, once called, never return control to callers**

- Hmm… what they are good for?

- What about the following?
  - enable a stack-less model of computation
  - enable optimizations that make recursion as cheap as loops
    - user-defined control structures
  - facilitate web programming
  - …

- Let see …

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

Recursive Factorial Function

```
(define (factorial1 n)
  (if (= n 1)
    1
    (* n (factorial1 (- n 1)))))
```

Control flow as a growing and shrinking stack of executing frames

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

A More Iterative Factorial Function

- „Bottom up“ perspective on computing factorials:
  - multiply 1 by 2, multiply result by 3, …, until we reach n

```
(define (factorial2 n)
  (local

     ([define (factIter counter result)
          (cond
            ((> counter n) result)
            (else (factIter
                      (+ 1 counter)
                      (* result counter))))])

  (factIter 1 1)))
```

A More Iterative Factorial Function

- Process does not grow and shrink
- The process is actually iterative:
  - Computation state is captured in a fixed number of variables: `result` and `counter`
  - fixed rule describes how state variables should be updated as the process goes on
  - (optional) end test specifies the termination condition

No need for a stack at all

```
(factorial 6)
(fact-iter   1 1 6)
(fact-iter   1 2 6)
(fact-iter   2 3 6)
(fact-iter   6 4 6)
(fact-iter  24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

Comparison

- `factorial1` :
  - \+ Code structure reflects the mathematical definition
  - – The information as “where the process is” not exlicitly contained in program variables
    - Need a stack
- `factorial2`:
  - \+ Program variables contain a complete description of the state of the process at any point
  - – More complex, less elegant definition
    - Accidental complexity

Comparison

```
(define (factorial1 n)
  (if (= n 1)
    1
    (* n  (factorial1 (- n 1)) )))
```

Needs to return to caller; The latter needs the result to complete

```
(define (factorial2 n)
  (local
      ([define (factIter counter result)
          (cond
            ((> counter n) result)
            (else (factIter
                      (+ 1 counter)
                      (* result counter)) ))])
    (factIter 1 1)))
```

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

Tail Calls

A call to a function `g` within a function `f` is a **tail call** if:

in the control path that leads to `g`'s call, the value of `f` is determined by the invocation of `g`.

- `g` sends its value directly to whoever is expecting `f`'s value ➔ calling `g` does not require any manipulation on stack

A special case of a tail call is **tail recursion**. It is a tail call from a procedure to itself.

Tail Call Optimization (TCO)

- No „return statements“ anywhere. Once a function call has finished, control never returns back
- No need for stack frames sitting around to use after returning from function calls
  - We can just throw away each stack frame after triggering the rest of the computation

- This is exploited by compilers to replace calls in a tail position with jump statements ➔ Saves stack space!
  - **tail call optimization** (abbreviated to **TCO**)
  - **tail call elimination**

Consequences of TCO

Can use recursive algorithms without the fear of running out of stack space

- Built-in looping constructs are 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

- Custom loop constructs as efficiently as build-in loops
  - Especially interesting for new data structures

A User-Defined `for`-Loop

Test

```
(for 10 positive? sub1 + 0)
```

```
(define (for init cond change body result)
  (local
    ([define (loop init result)
        (if (cond init)
            (loop (change init)
                  (body init result))
            result)])
    (loop init result))
)
```

`loop` contains a recursive tail call

`for` contains tail call to `loop`

Tail Calls versus Tail Recursion

- Some compilers support only tail recursion optimization.
- Tail recursion is an important special case, but not the only useful one
- Sometimes it makes sense to split computations across two procedures for the sake of clear program structure
  - TCO avoids performance penalty of stack-based procedure calls, if the latter are tail calls

Languages and TCO

- 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 from the course web page for more details and for investigating the reasons

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

Eat the cake and have it too?

- Functions with tail calls are more efficient but can be more complex than non-tail-call counterparts
  - Recall e.g., the two definitions of factorial

- Can we have both clarity of definitions and efficiency?

Continuation Passing Style (CPS)

- "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**)

- Details about the CPS transformation in PLAI book …

- For now think of 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` also passes to `g` a function of one parameter, to which `g` should pass its result
  - **Receiver function (also continuation)**
- Receiver captures the rest-of-computation at the point of executing the call!
- No function ever returns

Factorial in CPS

- Let‘s try to derive the CPS version of `factorial` …

```
(define (factorial n)
  (if (= n 1)
    1
    (* n  (factorial (- n 1)) )))
```

Factorial in CPS

- First of all, the top-level factorial is itself called from somewhere … with some receiver `r`

```
(define (factCPS n r)
  (if (= n 1)
     ???
  ))
```

- Next step: What should `factCPS` do when `n` is `1`?
  - pass the answer for this special case, to the receiver

```
(define (factCPS n r)
  (if (= n 1)
     (r 1)
     (???)
  ))
```

Factorial in CPS

```
(define (factorial n)
  (if (= n 1)
    1
    (* n (factorial (- n 1)) )))
```

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

- Rest-of-computation informally:
  - multiply the result of the recursive call, `restResult`, with `n`
  - pass the result of this multiplication to `r`

```
(lambda (restResult)
            (r (* n restResult))
```

Factorial in CPS

- If `n` is not `1`, `factCPS` simply calls itself, passing the new receiver (rest-of-computation) as a parameter

```
(define (factCPS n r)
  (if (= n 0)
      (r 1)
      (factCPS (- n 1)
               (lambda (rest-result)
                       (r (* n rest-result))))))
```

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

A Simple Program …

- Present user with a prompt for a number
- Given the 1st input, prompt for a second number
- Given the 2nd input, display the sum of the two inputs

```
(define (simpleApp)
  (displayResult
    (+
     (promptRead "First number: ")
     (promptRead "Second number: "))))

(define (promptRead str)
    (begin
      (print str)
      (read)))

(define (displayResult n) (printf "Output: ~a~n" n))
```

A Simple Program …

- … does not work on the Web ☹
- **Web protocol is stateless**:
  - Every time a Web program sends a web page to the user, it terminates
  - when the user resumes the computation (by clicking on a link/button) some other program must do the rest

The Simple Program on the Web…

- In our example:
  - computation would be terminated by the server after the first number is read
  - all state is lost, including the information about the **pending computation**
- We simulate web behavior on the desktop by using Scheme's `error` function instead of `print`

Run webApp in web1.scm

A Simulated Naive Web Program

```
(define (webApp)
  (webDisplay
    (+
     (webRead "First number: ")
     (webRead "Second number: "))))

(define (webRead str)
    (begin
      (error 'web-read str)
      (read)))

(define (displayResult n) (printf "Output: ~a~n" n))
```

The Program Becomes Complicated

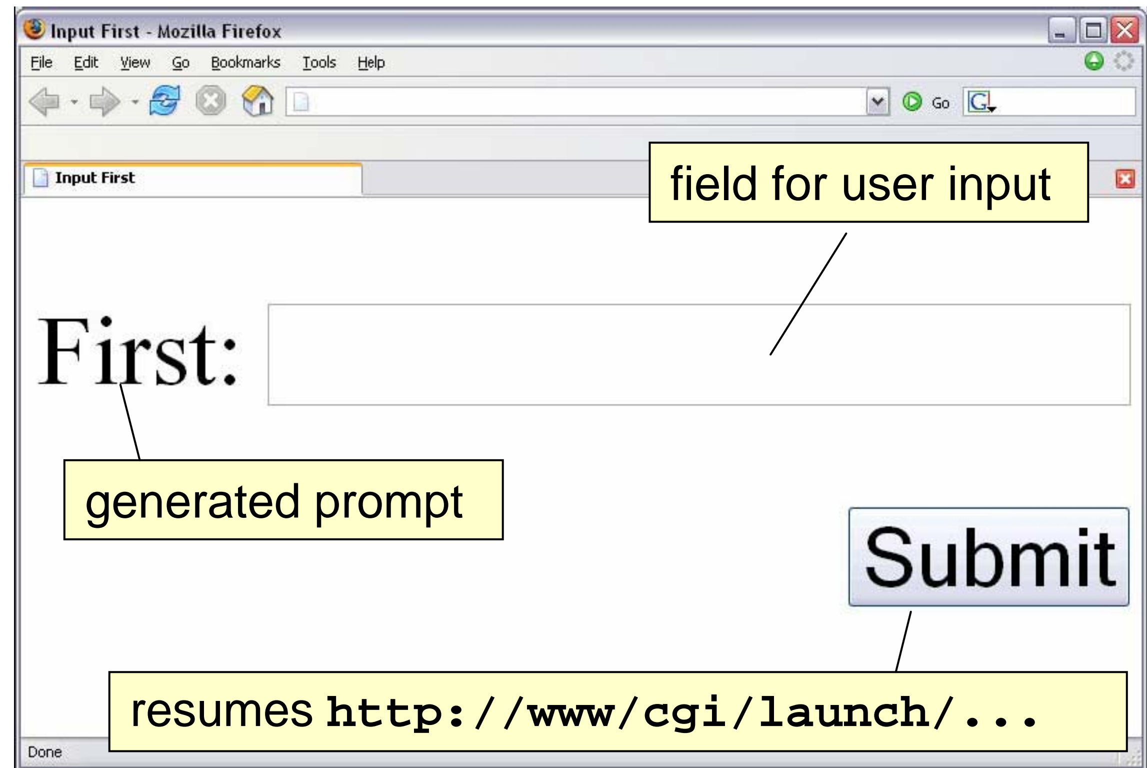
- Stateless protocols are great for scalability, but …

- Awkward to implement
  - Onus of managing the computation state shifted from the programming framework to the programmer

- Our simple program must be split into 3 independent applications …

The Program Becomes Complicated

1. Display 1st form with prompt for 1st input and a submit action field



2. Consume value from 1st form, generate a 2nd form

3. Consume value of 2nd form, compute and display result

The Program Becomes Complicated

- The web version of the program must explicitly treat pending computations
  - Pending computation is specified in the URL of the “action” field of the form generated by 1st program
  - **The application at that URL must be capable of resuming the computation**

- Value of 1st form must also be transmitted to 3rd step
  - typically done using hidden fields
  - if stored in a session object or a database field, and developer works in multiple windows, the application can compute the wrong answer

Let‘s Try Some JSPs

- Run http://localhost:8080/state/first.jsp
- Run http://localhost:8080/state2/

- And, have fun ☺

A possible web experience?

I want to book a hotel via the web. The web server asks: where? I say Alaska. I love polar bear rides. The server shows me a list of hotels in Alaska. I pick one, it shows me the details. I think that maybe Alaska in December is too cold, so I clone the browser window, press back a couple of times, until I'm back to the question: where do you want to go today? This time I answer: Hawaii. I get a list of hotels in Hawaii, and pick one. Now I can compare the two destinations side by side.

Upon reflection, I decide to go to Alaska after all. When will I ever get a chance to go Polar Bear Back riding again? I press the button that says "Book hotel and tickets Now!". I go to the airport on the designated day. As I walk off the plane into the Honolulu sunshine, I find that my parka is uncomfortably warm.

Traditional Web Programming

- Even if we know how to store conversational state, it's tiresome to
  - Decide about the part of the state to be remembered
  - Manually store/restore these parts in cookies, sessions etc.

- More important:
  - We have
    - a single logical sequence of interaction steps split into many different programs, one for each interaction step (Look at the code of the example JSPs)
    - dispatch to these programs manually
  - Hard to deal correctly with browser cloning and "back" buttons

CPS in Web Programming

- Overall control in `webApp`
  - calls the `webRead` primitive to read the 1st number
  - captures the rest of computation into a receiver procedure before calling `webRead`
  - passes the receiver to `webRead`

- `webRead` consumes prompt string and the pending computation
  1. Creates a new entry in a hash table to store pending computation (maintained in memory of the Web server)
  2. Generates a form action URL that contains the key of the new hash entry
  3. Generates a page and terminate the Web application

CPS in Web Programming

- When user submits response, the server invokes application named `launch`, which does the following:
  - Use the key associated with the `id` argument to obtain a receiver closure from the hash table

```
http://www/cgi/launch/id=k2592
```

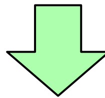
  - Extract user input
  - Apply the extracted value to the receiver

- This resumes the pending computation

Explicating Pending Computations

- Will the following work?

```
(webDisplay
   (+ (webRead “First number:”)
      (webRead “Second number:”)))
```



```
(webDisplay
   (+ (webRead “First number:”
                  (lambda (???) ???)
      (webRead “Second number:”)))
```

Explicating Pending Computations

- This won't work because Web application terminates at each interaction
- The application resumes only the "remembered" computation in the receiver – the identity function

Any computation that isn't explicitly mentioned in the receiver never gets performed

- All the pending computations must be moved to receiver
  - The second call to `webRead` as well as the addition operation and the call to `webDisplay`

Explicating Pending Computations

- What's pending at the point of the 1st interaction?
  - Consume the result of the first form (the first number)
  - Generate a form for the second number
  - Add them and display the result

```
(webDisplay
  (+ firstNumber
     (webRead "Second number:" ...)))
```

- By binding **firstNumber** with lambda, the pending computation after the 1st interaction is:

```
(lambda (firstNumber)
  (webDisplay
    (+ firstNumber
       (webRead "Second number:" ...))))
```

Explicating Pending Computations

- Next, we have to explicate the pending computation before second **webRead** call

```
(define webApp
  (webRead “First number:”
    (lambda (firstNumber)
       (webDisplay
         (+ firstNumber
            (webRead “Second number:” ??? )
)))
```

Explicating Pending Computations

- What's the computation after the 2nd interaction?

```
(lambda (secondNumber)
  (webDisplay
    (+ firstNumber
       secondNumber)))
```

must be in the closure of the procedure

- Than **webApp** becomes:

```
(define webApp
  (webRead "First number:"
    (lambda (firstNumber)
      (webRead "Second number:")
         (lambda (secondNumber)
           (webDisplay
             (+ firstNumber secondNumber)))))))
```

Let‘s run a simulation of the web program in Scheme…

Simulating the Web Program…

- Would like to test interactive Web programs without HTML (we must use HTML if we want to test the program in a Web browser)
- Testing a program at the console can be misleading
  - Scheme programs do not terminate after each interaction
  - So, the computation that is not moved to a receiver still can execute correctly
    - we won’t notice the problem
- Need a testing suite which models the Web interaction in Scheme

Simulating the Web Server…

- **webRead** stores the receiver and terminates

```
(define the-receiver (box 'dummy-value))
(define receiver-prompt (box 'dummy-value))

(define (webDisplay n)
  (printf "Web output: ~a~n" n))

(define (webRead p k)
  (begin
    (set-box! receiver-prompt p)
    (set-box! the-receiver k)
    (error 'webRead "run resume...")))
```

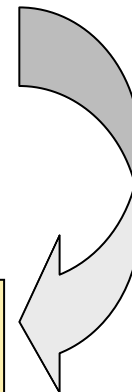
- **resume** uses the stored values to restore the computation

```
(define (resume)
  (begin
    (display (unbox receiver-prompt))
    ((unbox the-receiver) (read))))
```

Reflecting ...

- What have we actually performed when going from the first version of `webApp` to the second?

```
(define (webApp)
  (webDisplay
    (+
     (webRead "First number: ")
     (webRead "Second number: "))))
```




```
(define webApp
  (webRead "First number:"
    (lambda (firstNumber)
      (webRead "Second number:")
         (lambda (secondNumber)
           (webDisplay
             (+ firstNumber secondNumber)))))))
```

Interim Summary

- Web programming is hard
- Web programmers need to manage conversational state of the computation “manually” as well as data flow between small chunks of stateless computations
- Program structure gets lost
- Things may go wrong easily

- CPS organization of web programs helps with managing computation state and data flow by lambdas
- Automatic transformation into CPS takes the burden of doing so from the programmer

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

But, 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 receivers

3. Assumes that the run-time system will not needlessly create stack frames
   – many languages, e.g., Java, C, do this anyway; the program consumes memory unnecessarily

- The first problem is particularly important since it affects correctness of a program.

Continuations

- Some PLs provide an operation to construct and return the **"rest of the computation"** at any program point
  - Examples: Scheme; Smalltalk; Ruby, Standard ML,

"rest of the computation" is represented as a procedure of one argument called a **continuation.**

Computation resumed by applying it to a value.

In contrast to CPS:

Program does not need to be transformed a priori. Continuations can be created on-the-fly.

Continuations in Scheme

- `call/cc` (call with current continuation) construct
  - Captures the continuation at the point of its evaluation
  - Invokes lambda with `k` bound to that continuation

```
(call/cc
  (lambda (k) ( ... k ... ))
```

- Equivalent construct with a nicer syntax: `let/cc`
- `let/cc` is a binding construct
  - introduces a new scope and
  - binds continuation to a named identifier in this scope

```
(let/cc k  ( ... k ... ))
```

A First Example

- What is the value of this program?

```
(let/cc k (k 3))
```

- Which procedure represents the rest of the computation at the point of executing `let/cc`?

- Since there is no computation outside `let/cc`, → `k` will be simply an identity function,

- Hence:

```
   (let/cc k (k 3))
~> ((lambda (v) v) 3)
~> 3
```

A Second Example

```
(+ 1
    (let/cc k (k 3)))
```

```
k = (lambda (v) (+1 v))
```

- What's the result of the computation?
- Let's reduce

```
    (+ 1 (k 3)))
~> (+ 1 ((lambda (v) (+ 1 v)) 3))
~> (+ 1 (+ 1 3))
~> 5
```

- **Problem**:
  - We apply the "rest-of-computation" twice
  - The computation should halt after the application of `k`

Escapers

- We will symbolically use the notation **lambda↑** to represent a procedure that halts computation after execution of its body ➔ ESCAPERS

- Then continuations are bound to ESCAPERS

```
(+ 1 (let/cc k (k 3)))
```

```
k = (lambda↑ (v) (+1 v))
```

- Let's try to substitute again

```
   (+ 1 (k 3)))
~> (+ 1 ((lambda↑ (v) (+ 1 v)) 3))
~> ((lambda↑ (v) (+ 1 v)) 3)
~> (+ 1 3)
~> 4
```

Web Programming Revisited

- `webRead` explicitly consumes the rest of computation
- In `webApp` we have to create explicit representations of the rest-of-computation at two points in the program
- Contrived structure of `webApp`

```
(define webApp
  (webRead "First number:"
    (lambda (firstNumber)
      (webRead "Second number:")
        (lambda (secondNumber)
          (webDisplay
            (+ firstNumber  secondNumber)))))))
```

Continuations in Web Programming

- With continuations, we no longer create representations of rests-of-computations and pass them around
- `webRead` can capture the current continuation itself using `let/cc`
- The rest is the same:
  - `webRead` stores the continuation in a hash table entry and generates a URL containing the key to that entry
  - launcher extracts the continuation from the table and applies it to the user input

The web program will work without transformation to CPS!

Let‘s run a simulation of the web program that uses Scheme continuations …

Simulated Continuation Web Program…

```
(define the-receiver (box ‘dummy-value))
(define receiver-prompt (box ‘dummy-value))

(define (webDisplay n) (printf “Web output: ~a~n” n))

(define (webApp)
  (webDisplay
    (+ (webRead1 "First number: ")
       (webRead1 "Second number: "))))

(define (webRead str)
  (let/cc restOfComputation
     (begin
      (set-box! receiver restOfComputation)
      (set-box! prompt str)
      (error 'webRead "call (resume) to enter number"))))

(define (resume)
  (begin
   (display (unbox prompt))
   ((unbox receiver) (read))))
```

Continuations in 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 harmful or beneficial for Web programming
  - For more information about this follow corresponding links from the course web page

Exceptions with Continuations

- Imagine a division by zero in the middle of a computation:

```
(define (f n)
  (+ 10
     (* 5 (/ 1 n))))

(+ 3 (f 0))
```

Can produce division by 0

- Consider this version:

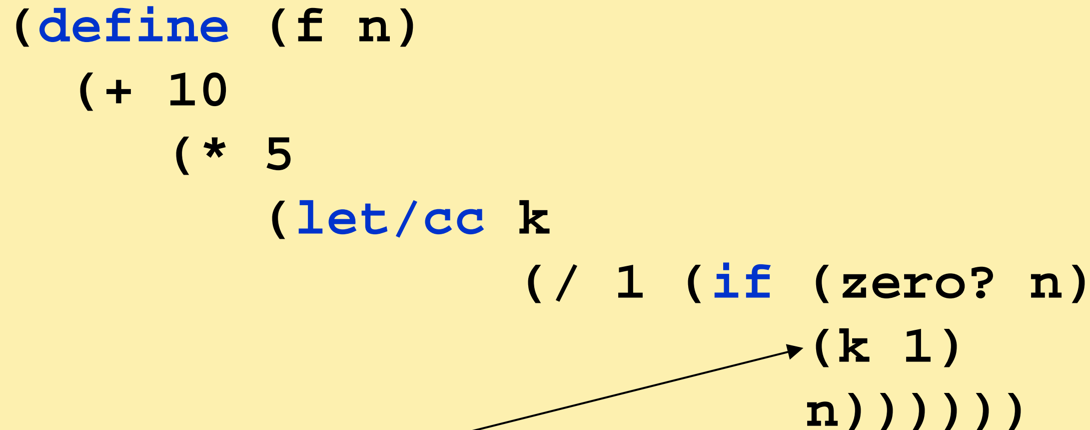
```
(define (f n)
  (+ 10
     (* 5
        (let/cc k
                (/ 1 n)))))

(+ 3 (f 10))
```

What is k bound to?

Exceptions with Continuations

- Instead of dividing by zero we can return 1



```
(define (f n)
  (+ 10
     (* 5
        (let/cc k
                (/ 1 (if (zero? n)
                         (k 1)
                         n))))))
```

- If `n == 0` substitute `v` by `1` in `k` and escape the rest, i.e., bypass division completely

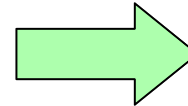
- `k` is acting as an exception handler
- calling it corresponds to raising the exception

Exceptions with Continuations

- We can see `let/cc` as a point at which we can cut out a part of the program and replace it with some value

```
(define (f n)
  (+ 10
     (* 5
        (let/cc esc
              (/ 1 (if (zero? n)
                       (esc 1)
                       n)))))))
```

`(esc 1)`



```
(define (f n)
  (+ 10
     (* 5
        1)))
```

cut-and-paste semantics

Exceptions with Continuations

- What if the continuation does not get invoked in the body of `let/cc`?

```
(define (f n)
  (+ 10
     (* 5
        (let/cc esc
                (/ 1 n)))))
```

- In this case, the value of the whole `let/cc` expression is the value of its body
  - if we don’t raise an exception, the expression evaluates as usual

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

- As any other program, an interpreter can also be converted to CPS
- In a CPS interpreter the current continuation is available at each stage of the interpretation
  – 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 start with the meta-interpreter that represents closures as Scheme functions

```
(define-type CFAE-Value
   [numb (n number?)]
   [closure (p procedure?)]
```

CFAE Interpreter in CPS

- Will turn it into CPS, i.e., the interpreter takes an extra argument – the receiver

```
(define (interp expr env k )
   (type-case CFAE expr
     [ ... ]
       ...
     [ ... ]))
```

Receiver expects the result of each expression's interpretation.

1. If the interpreter has a value handy, it supplies that value to the receiver.
2. Otherwise, the interpreter calls itself recursively with a (possible augmented) receiver

Interpreting Values in CPS …

- Numbers and identifiers are simply fed to the receiver

```
[num (n) (k (numb n))]
```

```
[id (v) (k (lookup v env))]
```

- Closures are also passed to receiver
  - But, prior to that, they're transformed to consume the receiver at the point of application (**dynamic receiver**)

```
[fun (param body)
     ( k (closure
             (lambda (arg-val dyn-k )
                (interp
                   body
                   (anEnv param arg-val env)
                   dyn-k )))))]
```

Non-Value Expressions in CPS …

- In all other cases:
  - further steps of interpretation are performed by recursive calls with a new augmented receiver
  - all pending computation must be moved to the receiver!

- Rule of thumb:
  - A call to the `interp` function never appears as a sub-expression
  - It is always the first thing we do…

CPS Interpretation of Addition

```
[add(l r)
   (interp l env
           (lambda (lv)
              (interp r env
                 (lambda (rv)
                   ( k (num+ lv rv))))))]
```

Augmented receiver for interpreting the left hand operand

Do have to transform **num+** into CPS?

CPS Interpretation of Application

receiver for interpreting the application is the dynamic receiver of the closure being applied

```
[app (fun-exp arg-exp)
     (interp fun-exp env
       (lambda (fv)
         (interp arg-exp env)
           (lambda (av)
             ((closure-p fv) av k )))) )]
```

receiver for interpreting the function expression

receiver for interpreting the argument expression

CPS Interpretation of Conditionals

```
[if0 (test then else)
     (interp test env
       (lambda (tv)
         (if (num-zero? tv)
             (interp then env  k )
             (interp else env  k ))))]
```

- Do we need to augment the receiver for interpreting `then` and `else` expressions?
- The pending computation after evaluating one of the branches is the same as the pending computation of the whole `if` expression, so they get `k` as the receiver

Adding Continuations to Language

- bindcc corresponds to let/cc in Scheme

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

- New value type for representing continuations

```
(define-type KCFAE-Value
   [numb (n number?)]
   [closure (p procedure?)]
   [cont (c procedure?)])
```

- We will overload the procedure application to handle continuation application

Adding Continuations to Language

- Receivers in the interpreter are similar to the continuations that we need, except for two things:
  - They 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 receiver of the interpreter simulates the corresponding continuation
  - They're Scheme procedures, not escapers (**lambda↑**)
    - but, our CPS interpreter never returns
    - this is effectively the same as termination

Interpreting `bindcc`

- To interpret bindcc, we have to interpret its body in an environment extended with a binding for the continuation

```
[bindcc (id body)
        (interp body
                (anEnv id
                       (cont ???)
                       env)
                k)]
```

What should we put as the value of the continuation?

The receiver for interpreting the body is `k`

Continuation closes over the environment at creation time → its body is scoped statically.

Interpreting bindcc

```
[bindcc (id body)
        (interp body
                (anEnv id
                       (cont (lambda (val)
                               (k val)))
                       env)
                k)]
```

- The whole point about continuations is to ignore the current receiver and use the stored receiver instead
- In contrast to “standard procedures”, the continuation procedure does not take a dynamic receiver as an argument

Overloaded Application

```
[app (fun-exp arg-exp)
     (interp fun-exp env
       (lambda (fv)
         (interp arg-exp env)
           (lambda (av)
             (type-case KCFAE-Value fv
               [closure (p) (p av k)]
               [cont (c) (c av)]
               [else (error "...")])))))]
```

- Function application uses the receiver at the point of application
- Continuation application uses the receiver at the point of creation

What's the initial continuation?

- What is the initial value of `k` to pass to the interpreter?
  - Theoretically is "the rest of DrScheme"
  - Practically,
    - we can use the identity function
    - Or a function that prints the received value and terminates

Properties of Continuations

- We ignore the continuation at the point of the application and use the continuation at the point of creation.
- We can employ the metaphor of replacing the body of `let/cc` with some value.
- But continuation is still a dynamic object, because it depends on the history of calls.
- The continuation closes over the environment
  - its body is scoped statically not dynamically

Some Thoughts On Stacks

- Consider again the procedure application rule:

```
[app (fun-exp arg-exp)
     (interp fun-exp env
       (lambda (fv)
         (interp arg-exp env)
           (lambda (av)
             ((closure-p fv) av k ))) )]
```

- Our interpreter does not need a stack,
  - because it never returns
- The stack is, however, modeled in the receivers
- When the function and the argument positions of the application are evaluated, the receiver is augmented

Some Thoughts On Stacks

```
[app (fun-exp arg-exp)
     (interp fun-exp env
        (lambda (fv)
          (interp arg-exp env)
             (lambda (av)
                ((closure-p fv) av k))) )]
```

- We need the stack only while we are evaluating the argument to the function and the function position
- The actual invocation does not put anything new to stack

Tail Calls Revisited

- Converting a function to CPS makes the stack explicit

- Hence, 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 does not push anything to the stack
  - Such invocations correspond to **tail calls**.

Tail Calls Revisited

- We have seen an example of a tail call:

```
(define interp expr env k
 ...
  [if0 (test then else)
     (interp test env
       (lambda (tv)
         (if (num-zero? tv)
             (interp then env k )
             (interp else env k )))))]
...)
```

```
(define interp expr env
 ...
  [if0 (test then else)
      (if (isZero (interp test env))
              (interp then env)
              (interp else env))]
...)
```

Tail Calls Revisited

- The calls in the implementation of addition are not tail calls since they augment the original receiver:

```
[add(l r)
   (interp l env
          (lambda (lv)
             (interp r env
                (lambda (rv)
                   ( k (num+ lv rv))))))]
```

```
[add(l r)
   (num+ (interp l env) (interp r env)]
```

Summary

- Tail calls and TCO
  - Enable to use recursion with the cost of simple loops
  - But, programs that only contain TCs may have a more complex structure than equivalent programs that do not have TCs
- CPS
  - Any program can be transformed to a version that only contains tail calls
  - CPS is important also for web programming
- Yet, transformation to CPS has disadvantages

Summary

- CPS disadvantages can be avoided in a language that allows to construct a continuation at any point
  - (`let/cc` in Scheme)
- Continuations are useful not only for Web programming,
  - they also allow to implement new control constructs
- Continuations are implemented,
  - by transforming the interpreter to CPS and
  - making the continuation available at any point of execution by a special binding construct