Programs as Data
Continuations

Zhoulai Fu

Tuesday 2020-11-10

Based on Niels Hallenberg and Peter Sestoft's slides. Thanks!

Intended learning outcomes

- Understand the concept of tail call and why it matters
- Experiment with encoding tail call in accumulators
- Understand continuation-passing styles
- Introduction on some applications of using continuations, including separation of concerns (making continuation explicit), exceptions, and backtracking

Intended learning outcomes

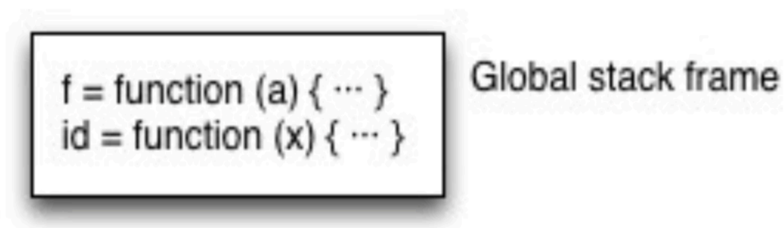
- **Understand the concept of tail call and why it matters**
- Experiment with encoding tail call in accumulators
- Understand continuation-passing styles
- Introduction on some applications of using continuations

Tail call

```
function id(x) {
    return x; // (A)
}
function f(a) {
    let b = a + 1;
    return id(b); // (B)
}
console.log(f(2)); // (C)
```

Javascript code example from https://2ality.com/2015/06/tail-call-optimization.html

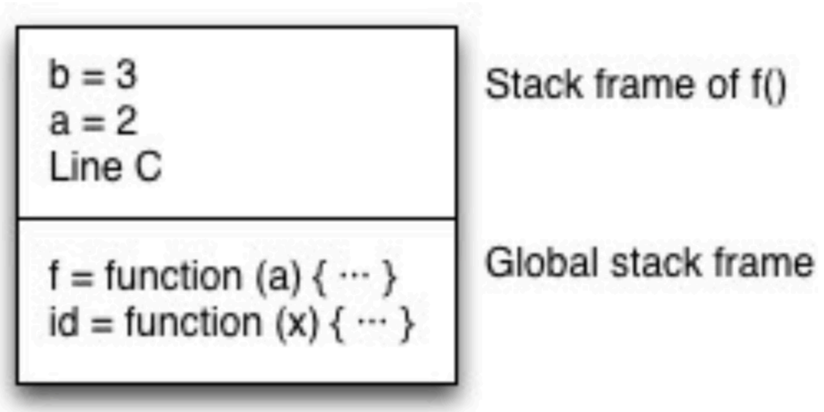
Step 1. Initially, there are only the global variables `id` and `f` on the stack.



Tail call

```
function id(x) {
    return x; // (A)
}
function f(a) {
    let b = a + 1;
    return id(b); // (B)
}
console.log(f(2)); // (C)
```

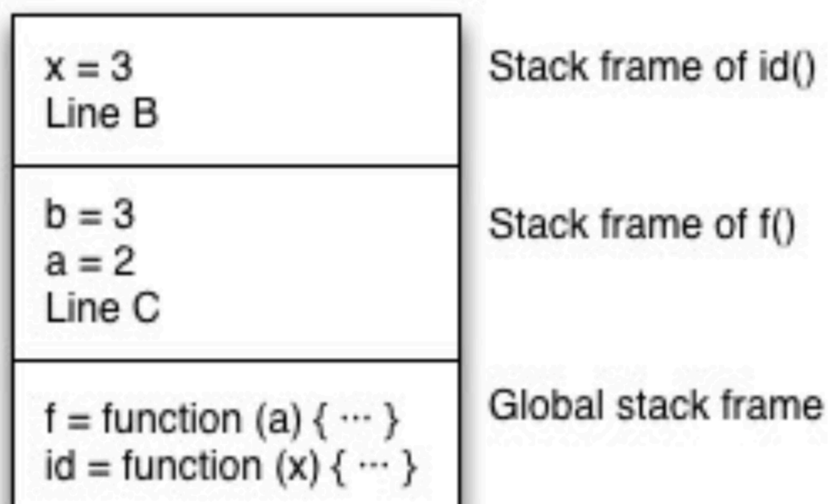
Step 2. In line C, `f()` is called: First, the location to return to is saved on the stack. Then `f`'s parameters are allocated and execution jumps to its body. The stack now looks as follows.



Tail call

```
function id(x) {
    return x; // (A)
}
function f(a) {
    let b = a + 1;
    return id(b); // (B)
}
console.log(f(2)); // (C)
```

Step 3. `id()` is called in line B. Again, a stack frame is created that contains the return address and `id`'s parameter.



Why tail calls matter

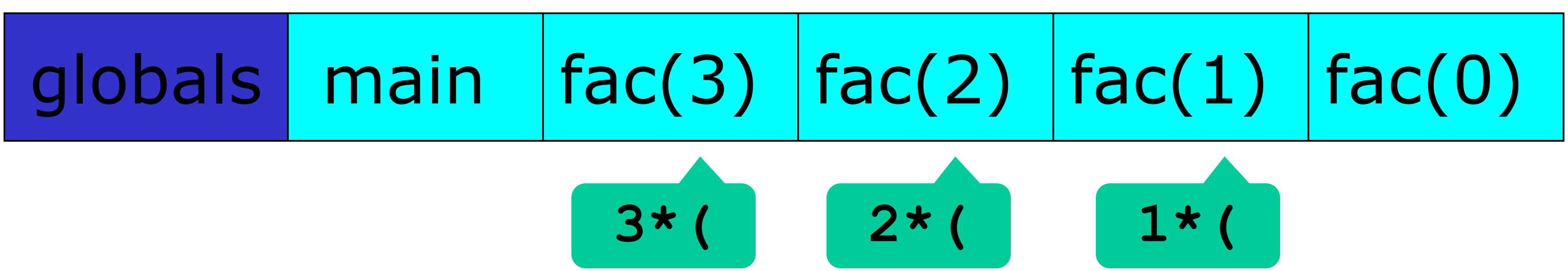
- Compilation if non-tail-call
  - Create a global stack frame at the beginning
  - Create a stack frame when making a call
  - Delete a stack frame when returning from a call
- Optimized compilation if tail-call
  - Create a global stack frame at the beginning
  - Create a stack frame when making a call. But if the call is a tail call, delete the current stack frame
  - Delete a stack frame when returning from a call

Recursive functions without tail calls

```
let rec facr n =
    if n=0 then 1
    else n * facr(n-1)
```

```
facr 3
==> 3 * facr 2
==> 3 * (2 * facr 1)
==> 3 * (2 * (1 * facr 0))
==> 3 * (2 * (1 * 1))
==> 3 * (2 * 1)
==> 3 * 2
==> 6
```

- One stack frame per recursive call
- Each stack frame represents "remember to *multiply result by n*"



Recursive function tail calls

- Rewrite facr with accumulating parameter r

```
let rec faci n r =
    if n=0 then r
    else faci (n-1) (r * n)
```

faci n r = r * (facr n)

facr n = faci n 1

```
faci 3 1
==> faci 2 3
==> faci 1 6
==> faci 0 6
==> 6
```

Uses no stack space!

Such memory efficiency can be important in real-world applications

- Quicksort has a straight-forward implementation with O(n) worst-case memory usage
- Its tail-recursive version uses O(Log n)
- When n=1 million, Log n is about 14

- I happened to work with medical scientists on brain image processing. We wanted to develop a real-time tool for scanning illness.
- We needed to deal with about 10*3 data sequences, each consisting of 10^6 floats ➔ 40G memory ➔ our algorithm was too slow to be useful because it used swaps

Which Calls Are Tail Calls?

Terminology: A call is a tail call if it is the last action of the containing function.

Examples: Below, `g` is a tail call and `h` is not.

```
g 1
g(h 1)
h 1+h 2
if 1=2 then g 3 else g(h 4)
let x = h 1 in g x end
let x = h 1 in if x=2 then g x else g 3 end
let x = h 1 in g(if x=2 then h x else h 3) end
let x = h 1 in let y = h 2 in g(x + y) end end
```

`Quiz 1. - open Emacs -> work_diary.org. Demo.`

Intended learning outcomes

- Understand the concept of tail call and why it matters
- **Experiment with encoding tail call in accumulators**
- Understand continuation-passing styles
- Introduction on some applications of using continuations

Quiz 2

- Given this function

```
let rec prod xs =
    match xs with
    | []    -> 1
    | x::xr -> x * prod xr
```

- Find a tail-recursive version using an accumulator (Demo: visual studio):

```
let rec proda xs a =
    match xs with
    | []    -> ...
    | x::xr -> ...`
```

Intended learning outcomes

- Understand the concept of tail call and why it matters
- Experiment with encoding tail call in accumulators
- **Understand continuation-passing styles**
- Introduction on some applications of using continuations

Making tail calls via continuations

- Continuation = "the rest of a computation"
- Continuation-passing style = CPS:
  - Every function has a continuation argument **k**
  - Do not return **res**; instead call **k(res)**
  - The continuation "knows what to do with **res**"

Normal

```
let rec facr n =
    if n=0 then 1
    else n * facr(n-1)
```

Continuation parameter

Don't return, instead call **k**

New continuation

CPS

```
let rec facc n k =
    if n=0 then k 1
    else facc (n-1) (fun v -> k(n * v))
```

Uses of continuation-passing style (CPS)

- All functions can be transformed to CPS. Compilers can use it as an intermediate representation during optimization
- CPS encodes tail calls and can sometimes be rewritten to use an accumulating parameter, saving memory
- A function in CPS can sometimes stop the computation early, saving time
- An interpreter in CPS can model exceptions and exception handling such as `try-catch`
- Continuations can implement expressions with multiple results, as in Icon and Prolog

Deriving a CPS version of facr

```
let rec facr n =
    if n=0 then 1
    else n * facr(n-1)
```

```
let id = fun v -> v
```

facc n k = k(facr n)

facr n = facc n id

```
let rec facc n k =
    if n=0 then ...
    else ...
```

```
let rec facc n k =
    if n=0 then k 1
    else ...
```

```
let rec facc n k =
    if n=0 then k 1
    else facc (n-1) <new continuation>
```

```
let rec facc n k =
    if n=0 then k 1
    else facc (n-1) (fun v -> k(n * v))
```

Evaluating facc n id

```
let rec facc n k =
    if n=0 then k 1
    else facc (n-1) (fun v -> k(n * v))

let id = fun v -> v
```

```
facc 3 id
==> facc 2 (fun v -> id(3 * v))
==> facc 1 (fun w -> (fun v -> id(3 * v)) (2 * w))
==> facc 0 (fun u -> (fun w -> (fun v -> id(3 * v)) (2 * w)) (1 * u))
==> (fun u -> (fun w -> (fun v -> id(3 * v)) (2 * w)) (1 * u)) 1
==> (fun w -> (fun v -> id(3 * v)) (2 * w)) (1 * 1)
==> (fun w -> (fun v -> id(3 * v)) (2 * w)) 1
==> (fun v -> id(3 * v)) (2 * 1)
==> (fu v -> id(3 * v)) 2
==> id(3 * 2)
==> id 6
==> 6
```

Quiz 3

- Given this function

```
let rec prod xs =
    match xs with
    | []     -> 1
    | x::xr -> x * prod xr
```

- Find the continuation-passing version:

```
let rec prodc xs k =
    match xs with
    | []     -> ...
    | x::xr -> ...
```

Continuations and accumulating parameters

- Both **`(prodc xs k)`** and **`(proda xs r)`** are tail-recursive
- What relation between **`prodc`** and **`proda`**?
- To recall, prodc returns k(res), where res is prod(xs); proda return r* res, where res is prod(xs).
- So, proda can be seen as a special case of prodc
- In fact, all functions can be made CPS
- Only some continuations can be represented simply with an accumulator

Intended learning outcomes

- Understand the concept of tail call and why it matters
- Experiment with encoding tail call in accumulators
- Understand continuation-passing styles
- **Introduction on some applications of using continuations**

Application of continuations

- Separation of concerns, by making the continuation explicit
  - We can ignore it, thus "avoid returning"
  - We can have two continuations, thus "choose how to return"
- Exception handling
- Backtracking
- More on the applications in next lecturess

Quiz 4 (working with an explicit continuation)

- Given this CPS

```
rec prodc xs r =
    match xs with
    | [] -> r 1
    | x::xr -> prodc xr (fun v -> r (x*v))
```

- Apply with three continuations
  - To print the result “The answer is …“
  - To raise a warning if the result is negative (which only makes sense in real, when the inputs are positive)

A simple functional language with exceptions

- Let's add exceptions to our small functional language:

```
type expr =
   | ...
   | Raise of exn                           // raise exn
   | TryWith of expr * exn * expr           // try e1 with exn -> e2
```

- Evaluation of an expression now either gives an integer result, or fails (aborts):

```
type answer =
   | Result of int
   | Abort of string
```

```
let rec coEval1 e env (cont : int -> answer) : answer =
```

Interpreter with continuation, for *throwing* exceptions, part 1

```
let rec coEval1 e env (cont : int -> answer) : answer =
    match e with
    | CstI i -> cont i
    | Var x  ->
      match lookup env x with
        | Int i -> cont i
        | _     -> Abort "coEval1 Var"
    | Prim(ope, e1, e2) ->
      coEval1 e1 env
        (fun i1 ->
          coEval1 e2 env
            (fun i2 ->
             match ope with
               | "*" -> cont(i1 * i2)
               | "+" -> cont(i1 + i2)
               | ... ))
    | Raise (Exn s) -> Abort s
```

Compare lecture 4, `Fun.fs`

Comparing with Fun.fs

```
let rec eval (e : expr) (env : value env) : int =
  match e with
  | CstI i -> i
  | Var x  ->
    match lookup env x with
    | Int i -> i
    | _      -> failwith "eval Var"
  | Prim(ope, e1, e2) ->
    let i1 = eval e1 env
    let i2 = eval e2 env
    match ope with
    | "*" -> i1 * i2
    | "+" -> i1 + i2
    | ...
```

No continuation but same structure.

Error reporting through F# build in exceptions

Interpreter with continuation, for *throwing* exceptions, part 2

```
let rec coEval1 e env (cont : int -> answer) : answer =
    match e with
    | ...
    | If(e1, e2, e3) ->
      coEval1 e1 env
              (fun b -> if b<>0 then
                          coEval1 e2 env cont
                        else
                          coEval1 e3 env cont)
    | ...
```

Examples to provoke Abort:

Ex10: foo used as variable
Ex11: Unknown prim.
Ex12: foo not a function

```
let ex10 = Letfun("foo", "y", CstI 3, Var "foo")
let ex11 = Prim("foo", CstI 3, CstI 4)
let ex12 = Let("foo", CstI 3, Call("foo", CstI 4))
```

Interpretation of exception *handling*

- Add an error continuation to interpreter: econt : exn -> answer
- To throw exception, call error continuation instead of normal continuation
- The error continuation looks at the exception and decides whether it wants to handle it
- The error continuation will contain the chain of exceptions to match, i.e. one new lambda added for each TryWith.

Interpreter with two continuations for throwing and handling (part 1)

```
let rec coEval2 e env (cont : int -> answer)
                      (econt : exn -> answer) : answer =
    match e with
    | CstI i -> cont i
    | If(e1, e2, e3) ->
      coEval2 e1 env (fun b ->
                      if b<>0 then
                        coEval2 e2 env cont econt
                      else
                        coEval2 e3 env cont econt)
                     econt
    | ...
    | Raise exn -> econt exn
    | TryWith (e1, exn, e2) ->
      let econt1 thrown =
          if thrown = exn then coEval2 e2 env cont econt
                          else econt thrown
      in coEval2 e1 env cont econt1
```

Contfun.fs

Interpreter with two continuations for throwing and handling (part 2)

The top-level error continuation returns the continuation, adding the text ”Uncaught exception”

```
let eval2 e env =
  coEval2 e env
    (fun v -> Result v)
    (fun (Exn s) -> Abort ("Uncaught exception: " + s))
```

Examples:

```
let ex4 =
  TryWith(Prim("*", CstI 11, Raise (Exn "Outahere")),
          Exn "Outahere", CstI 999);
> eval2 ex4 [];;
val it : answer = Result 999
```

Catch an exception (ex4)

```
let ex13 = Raise (Exn "Uncaught")
> eval2 ex13 [];;
val it : answer = Abort "Uncaught exception: Uncaught”
```

Uncaught exception (ex13)

Contfun.fs

Interpreter for imperative language with two continuations

```
let rec coExec2 stmt (store : naivestore)
      (cont : naivestore -> answer)
      (econt : exn * naivestore -> answer) : answer =
match stmt with
| Asgn(x, e) ->
    cont (setSto store (x, eval e store))
| If(e1, stmt1, stmt2) ->
    if eval e1 store <> 0 then
      coExec2 stmt1 store cont econt
    else
      coExec2 stmt2 store cont econt
| Throw exn ->
    econt(exn, store)
| TryCatch(stmt1, exn, stmt2) ->
    let econt1 (exn1, sto1) =
      if exn1 = exn then coExec2 stmt2 sto1 cont econt
                    else econt (exn1, sto1)
    coExec2 stmt1 store cont econt1
| …
```

```
type answer =
  | Terminate
  | Abort of string

type exn =
  | Exn of string
```

Contimp.fs

Continuations and Backtracking

Backtracking:

When a subexpression produces result *v* that later turns out to be inadequate, the computation may backtrack to that subexpression to ask for a new result v’ that may be more adequate.

Expressions in Icon:

The *result sequence* of an expression is the sequence of results it may produce.

The sequence is empty if the expression fails.

Expressions that give multiple results; the Icon language

Expression	Result seq.	Print	Comment
5	5		Constant
write 5	5	5	Constant, side effect
(1 to 3)	1 2 3		Range, 3 results
write (1 to 3)	1 2 3	1	Side effect
every (write (1 to 3))		1 2 3	Force all results
(1 to 0)			Empty range, no res.
&fail			No results
(1 to 3)+(4 to 6)	5 6 7 6 7 8 7 8 9		All combinations
3 < 4	4		Comparison success
4 < 3			Comparison fails
3 < (1 to 5)	4 5		Success twice

- Quiz:
  - Code an Icon expression to find the least multiple of 13 that is larger than 100
  - Where does backtracking occur when evaluating the expression?

Micro-Icon interpreter

- The interpreter takes two continuations:
  - A *failure continuation*

    `econt : unit -> value`

    called when there are no (more) results
  - A *success continuation*

    `cont : value -> econt -> value`

    called when there is one (more) result
- The econt argument to cont can be called by cont to ask for more results
- More on backtracking and exceptions in an exercise and next lectures

```
type value =
| Int of int
| Str of string
```

Reading and homework

- This week's lecture:
  - PLCSD chapter 11
  - Exercises 11.1, 11.2, 11.3, 11.4, 11.8
- Next week:
  - PLCSD chapter 12