CS558
Programming Languages

Fall 2023

Lecture 6b

Andrew Tolmach

Portland State University

Semantics of first-class functions

- What’s in the “value” of a first-class function f ?
- Roughly speaking, just f’s definition (its parameters and body expression)
- But nested functions can have free variables defined in an enclosing scope, and the behavior of the function depends on their values.
- To find those values, it suffices to record the environment surrounding the declaration of f
    - Store this in a “closure” representing f
- But: must make sure locations used in environment stay live as long as closure does!

Semantics of first-class functions

```
case class ClosureV(x:String,b:Expr,e:Env) extends Value
def interpFun(x:String,b:Expr,cenv:Env):Value ={
  val fp = stack.getSp()
  interpE(cenv + (x -> StackFrameOffset(-4)),fp,b)
}
def interpE(env:Env,fp:Int,expr:Expr):Value = expr match {
  case Fun(x,e) => ClosureV(x,e,makeLocsAbsolute(env))
  case App(f,e) => interpE(env,fp,f) match {
     case ClosureV(x,b,cenv) =>
       val offset = stack.push(fp,1)
       val v = interpE(env,fp,e)
       setLocation(fp,StackFrameOffset(offset),v)
       val r = interpFun(x,b,cenv)
       stack.pop(1)
       r
     case _ => throw InterpException (…)
     }
  case Let(x,e,b) => …
  …
}
```

Makes copy of env converting all stack locations to absolute addresses

But using stack allocation for locals and parameters won't work!

Trouble with first-class functions

- Problem: if a closure value is returned, the environment may point to locations that have been popped from the stack!

```
(let f (fun (x)(fun (y)(+ x y)))
     (let g (@ f 42)
          …))
```

At this point, stack storage for x has been popped.
So we're in trouble if we call g.

- You will explore this scenario in this week's lab.
- Similar problems can occur if closure is stored into a heap data structure.

First-class functions need the heap

- Bottom line: if we want the flexibility of fully first-class functions, we cannot store their free variables on the stack.

- They must instead live in the heap. How to arrange this?

- Simple solution: just allocate all activation record data in the heap instead of the stack

  - Rely heavily on garbage collection to deallocate and re-use them.

  - (A few compilers do this.)

More refined solutions

- One option: heap-allocate just those variables that are free in some closure expression.
  - Usually done by adding an extra layer of boxing around such variables.
  - (Some compilers do this.)
- Pure functional language solution: if the free “variables" are actually immutable, we can store copies of their values in a heap closure record.
  - (Most functional language compilers do this. See this week's lab!)
  - Again, suffices to copy just the free variables of the body.

The tyranny of the stack

- Many older languages support functions as values, but not in fully first-class way—so that variables cans still be stored in the stack.

- For example, C allows fully first-class functions, but has no nested functions. Hence functions have no free variables and closures are not needed: function values are just simple code pointers.

- On the other hand, some languages with nested functions (e.g. Pascal, Modula, Ada) allow functions to be passed downwards as parameters to other functions, but not returned or stored.

“Downwards funargs”

Nested functions on the stack

- In a language with only downwards funargs, if function f defines nested function g, it is impossible for g to be called after f has returned.

Why?

- So the compiler can use conventional stack storage, because every free variable of a function lives in an activation record that is still on the stack when the function might be called.
  - Finding the values of those variables may be non-trivial. For example, compiler might need to maintain static links between frames. (See textbook and this week's lab for more details.)

Closures vs. Objects

- First-class function closures provide a way to package a function with some (relatively) fixed parameters (its free variables)
- Objects can be used to give a similar effect:

```
case class MultiplesOf(n: Int) {
  def apply(xs:List[Int]) : List[Int] = xs match {
    case Nil => Nil
    case (y::ys) => if (y%n == 0) y::apply(ys)
                    else apply(ys)
  }
}

val evens = MultiplesOf(2)
val v = evens.apply(List(1,2,3,4)) // yields List(2,4)
```

Can treat all first-class functions this way

More verbose than implicit closures

Call-by-name using closures

- Recall semantics of call-by-name: bind formal parameter to actual argument expression, with free variables resolved in caller

```
foo x y = if x = 0 then x else y
let x = 1000000 in foo 1 (factorial x)
```

- We can implement this using “thunks”: first-class functions of zero arguments

```
foo x y = if x() = 0 then x() else y()
let x = 1000000 in
  foo (fun () => 1) (fun () => factorial x)
```

- Shows that CBN introduces significant overhead

Pure Functional Programming

- Idea: compose programs out of pure functions that have no side-effects
- Software engineering advantage: greatly improves modularity, because no hidden interactions between functions
- Pure functions won't interfere if evaluated in parallel
- Works well with call-by-name/need (in impure languages we need to worry about evaluation order)
- I/O is problematic, but Haskell has a good solution: program computes an I/O action which is then performed
- ML family is impure: no variables, but mutable heap structures (e.g. boxes) and explicit I/O

Recursion vs. iteration, again

- Recall that tail-recursive functions can be converted into iterations.
- If our language has first class functions, we can code in a style where every call is a tail call.
- To show the idea, consider the familiar list `length` function:

```
def length (xs:List[Int]):Int = xs match {
   case Nil => 0
   case (_::xs1) => 1 + length(xs1)  // not tail-recursive
   }
```

- (We already know how to write this tail-recursively using an accumulating parameter.)

Tail-calls are enough

- Here’s another tail-recursive way to write `length`:

```
def length (xs:List[Int]) : Int = {
   def klength (xs:List[Int],k:Int => Int) : Int = xs match {
     case Nil => k(0)
     case (_::xs1) => klength(xs1,r => k(1+r))
   }
   klength(xs,r => r)
 }
```

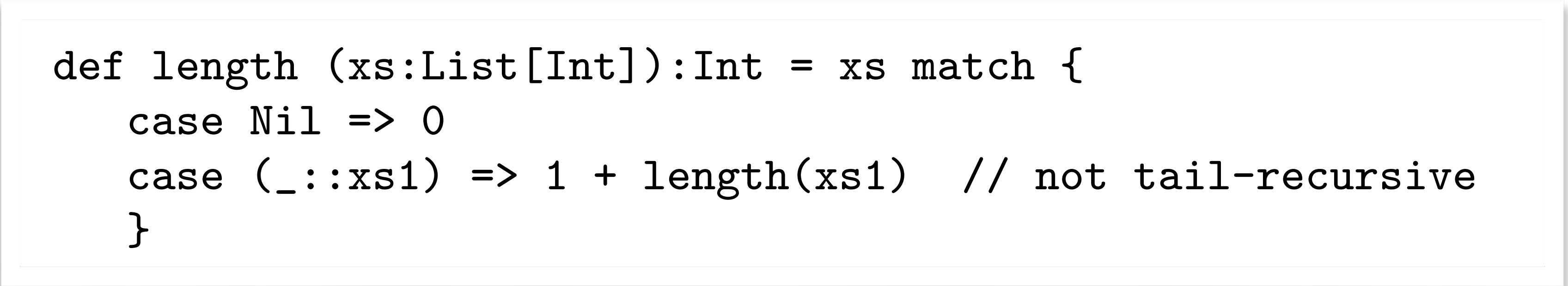
- This rather odd code was constructed by giving `klength` an additional parameter, `k`, of type `Int => Int`. Instead of returning its “result” value, `klength` passes it downwards to `k`.

- The `k` parameter tells `klength` “what to do next”

- Every call in `klength` is a tail-call, so we can evaluate `klength` without a stack!

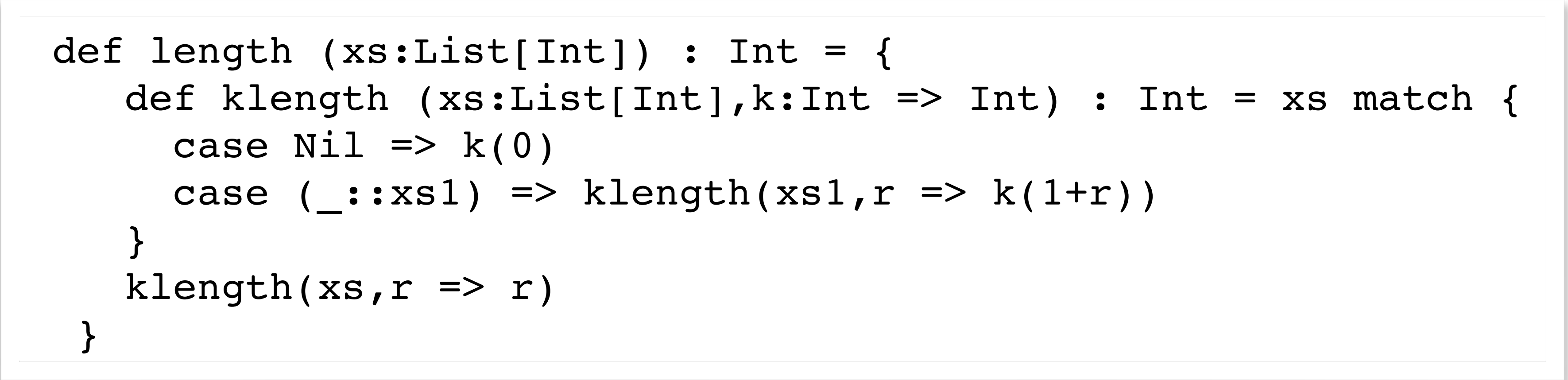
Comparison of computations

length (1::2::Nil) →
1 + (length(2::Nil)) →
1 + (1 + length(Nil))) →
1 + (1 + 0)) →
1 + 1 →
2



```
def length (xs:List[Int]):Int = xs match {
   case Nil => 0
   case (_::xs1) => 1 + length(xs1)  // not tail-recursive
   }
```

length (1::2::Nil) →
klength(1::2::Nil, $\lambda r.r$) →
klength(2::Nil, $\lambda r_{1}.(\lambda r.r)\ (1+r_{1})$) →
klength(Nil,$\lambda r_{2}.(\lambda r_{1}.(\lambda r.r)\ (1+r_{1}))\ (1+r_{2})$) →
$(\lambda r_{2}.(\lambda r_{1}.(\lambda r.r)\ (1+r_{1}))\ (1+r_{2}))\ (0)$ →
$(\lambda r_{1}.(\lambda r.r)\ (1+r_{1}))\ (1+0)$ →
$(\lambda r_{1}.(\lambda r.r)\ (1+r_{1}))\ (1)$ →
$(\lambda r.r)(1+1)$ →
$(\lambda r.r)(2)$ →
2



```
def length (xs:List[Int]) : Int = {
   def klength (xs:List[Int],k:Int => Int) : Int = xs match {
     case Nil => k(0)
     case (_::xs1) => klength(xs1,r => k(1+r))
   }
   klength(xs,r => r)
 }
```

Continuation-passing Style

- Functions like `k` are (loosely) called continuations and programs written using them are said to be in (a form of) continuation-passing style (CPS).

- We might choose to write (parts of) programs explicitly because it makes it easier to express a particular algorithm, or because it clarifies the control structure of the program

- Note that CPS'ed Scala programs are just a subset of ordinary Scala programs that happen to make use of the (existing) enormous power of first-class functions.

- (Remarkably, we can systematically convert any functional language program into an equivalent CPS-style program.)