Functional Debugging

Sam Myers and Taylor Sheneman

Debugging a Functional Language

- Debuggers are commonly used in imperative languages
- Line-by-line execution → line-by-line debugging
- S-expressions don’t apply as intuitively

```
((fun (n) (let ((m 3)) (+ (if (= n 2) 4 (+ m n)) (* 3 1)) )) 5)
```

Continuations, Again

- CPS “unwinds” recursive expressions into execution order
- Return values explicitly passed to the “continuation” function(s)
- At any point in the program, you have an explicit function representing the rest of the computation
- Very, very powerful
- Also very confusing

Continuations in Debugging

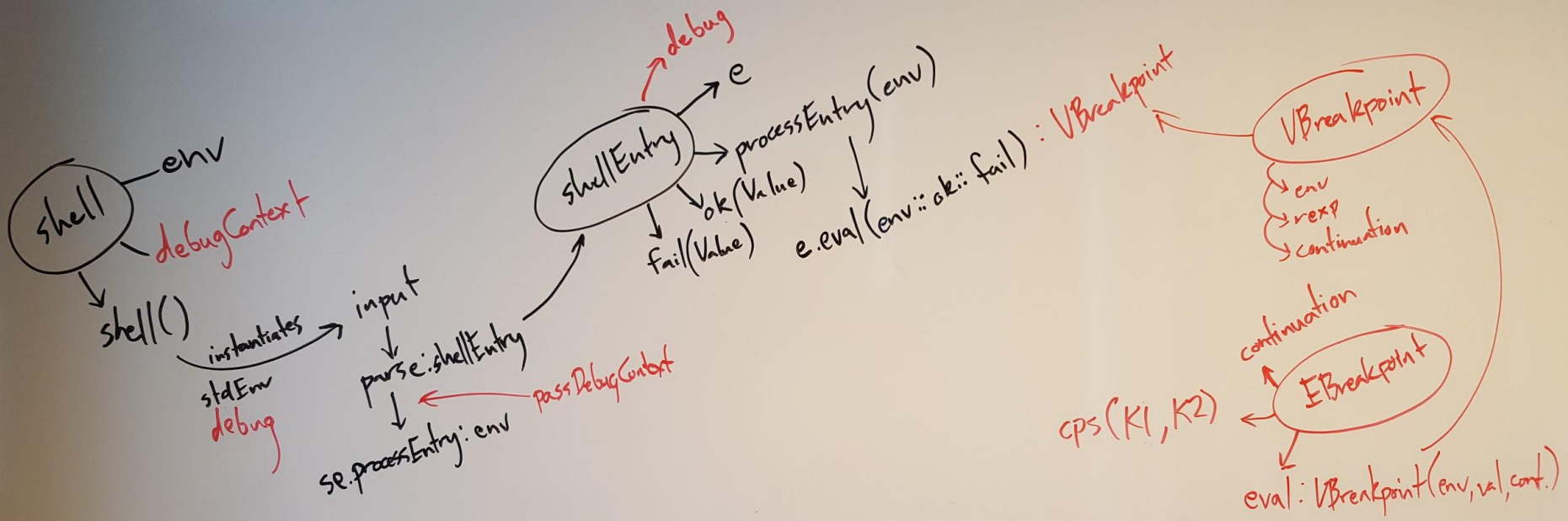
- Having a “rest of the computation” function is very nice for debugging
- Can halt execution and save state:
  - Current environment
  - Current return value
  - Continuation function
- This debugger is built on the Lecture 11: Continuations and Exceptions code

Our Debugger

- A bit more like the backend of a debugger
- Breakpoints and stepping
- View environment at each break/step
- Directives:
  - #continue
  - #step
  - #stepover

Storing Context in the Shell

- The shell is already the top-level environment, stores ENV values
- We add another structure, DebugContext, which stores a snapshot of the environment, the return expression and continuations
- Need to pass a reference to an instance of DebugContext into the SEexpr so internal functions can pass values back up


shell
env
debugContext
shell()
instantiates
stdEnv
debug
input
parse: shellEntry
se.processEntry: env
passDebugContext
shellEntry
debug
e
processEntry(env)
ok(Value)
fail(Value)
e.eval(env :: ok :: fail) : VBreakpoint
VBreakpoint
env
rexp
continuation
continuation
EBreakpoint
cps(K1, K2)
eval: VBreakpoint(env, val, cont.)

DIAGRAM

Breakpoints

- New classes: EBreakpoint and VBreakpoint
- EBreakpoint is inserted as (bkpt *exp*) around any expression, and will:
- VBreakpoint contains the information we need for DebugContext

```
((fun (n) (bkpt (let ((m 3)) (+ (if (= n 2) 4 (+ m n)) (* 3 1))))) 5)
```

```
def break (bkpt: VBreakpoint) : Unit = {
  paused = true
  env = bkpt.getEnv()
  continuations = Some(bkpt.getContinuations())
  returnExp = Some(bkpt.getReturnExp())
}

def continue () : Unit = {
  paused = false
  returnExp.get.cps(continuations.get.head, continuations.get.last).eval(env)
}
```

EBreakpoint

- Prevent the inner expression from being transformed immediately into CPS
- Redirect the continuation to K2, the error continuation
- Return a VBreakpoint when eval( ) is called (the code will hit the error and return)
- Can return the inner expression to remove the breakpoint

```
case class EBreakpoint (val e : Exp) extends Exp {

  var continuations: Option[List[Exp]] = None

  override def toString () : String =
    "EBreakpoint("+ e + ")"

  def cps (K1 : Exp, K2 : Exp) : Exp = {
    continuations = Some(List(K1, K2))
    return new EApply(K2, List(this))
  }

  def eval (env:Env[Value]) : Value = {
    return new VBreakpoint(env, continuations.get, e)
  }

  override def getExp () : Exp = e

  override def isBreakpoint () : Boolean = true

  override def getDebugReadable () : String = e.getDebugReadable()
}
```

```
def fail (v : Value) : Value = {
  if (v.isBreakpoint()) {
    println("Original expression:")
    println(debug.getInput())
    debug.break(v.asInstanceOf[VBreakpoint])
    println("\nNext to execute:")
    println(v.getReturnExp().getDebugReadable() + "\n")
    println("Abstract representation:")
    println(v.getReturnExp() + "\n")
    println("Environment:")
    val nonStandard = v.getEnv().getContent().filterNot(x => keywords.contains(x._1) | x._1 == "")
    println(new Env(nonStandard.filterNot(_._1.startsWith("  "))))
  } else {
    println("EXCEPTION("+v+")")
  }
  return VNone
}
```

Step In

- We implement stepping by programmatically inserting a new breakpoint and continuing from the previous breakpoint
- Step one level into the return expression, save the previous breakpoint context (return expression, env, continuation)

```
override def insertBreakpoint (position: Int) : Int = {
  position match {
    case 0 => {
      ec = new EBreakpoint(ec)
      return position
    }
    case 1 => {
      ec = ec.getExp()
      et = new EBreakpoint(et)
      ee = new EBreakpoint(ee)
      return 2
    }
    case _ => {
      return -1
    }
  }
}
```

```
def stepInto () : Unit = {
  stepPosition = returnExp.get.insertBreakpoint(0)
  if (stepPosition < 0) {
    stepPosition = 0
    stepOver()
    return
  } else {
    prevEnv = env
    prevContinuations = continuations
    parentExp = returnExp
    continue()
  }
}
```

Step Over

- Step across the return expression to the next at the same level
- Step over is trickier; the rest of the expression is already transformed to CPS
- We end up removing the previous breakpoint, setting a new one in the next sub-expression, and rewinding/replaying

```
def stepOutAndContinue () : Unit = {
  paused = false
  parentExp.get.cps(prevContinuations.get.head, prevContinuations.get.last).eval(prevEnv)
}

def stepOver () : Unit = {
  if (parentExp.isEmpty) {
    continue()
    return
  }
  var position = parentExp.get.insertBreakpoint(stepPosition + 1)
  if (position < 0) {
    stepPosition = 0
  } else {
    stepPosition = position
  }
  stepOutAndContinue()
}
```

Demo!

```
(let ((x 4) (y 3)) (bkpt (let ((z (+ x 1)) (w 2)) (if (= z y) (+ w 6) (+ y 3)))))
```

Possible Future Work

- Better interface for inserting breakpoints (probably graphical?)
- ~~More informative readouts at each breakpoint (in surface syntax)~~
  - Nevermind, Sam wrote an anti-parser.
- Modify environment in BKPT> paused state interactive shell
- Other neat things you can do with continuations
  - Cure cancer
  - Time travel

Thanks!