Monte: A Spiritual Successor to E

Corbin Simpson

August 7, 2017

Abstract

We introduce Monte, a capability-safe programming language in the style of E. We detail several of Monte's differences from E, including new syntax, several new interfaces, and a simple module model. In particular, we examine four features and their consequences for secure distributed computing: named arguments, iterators, modules, and controllers.

1 Introduction

Monte is a programming language cast in the same mold as E. Like E, Monte is a capability-safe language which models capabilities as objects (the "object-capability" security model). Also like E, Monte expands from a rich sugared language, Full-Monte, to a strict subset, Kernel-Monte. In fact, Kernel-Monte is almost exactly like Kernel-E, and we discuss named arguments and iterators, the two largest divergences that Kernel-Monte makes from Kernel-E.

2 Named Arguments

In addition to positional arguments, Monte supports passing arguments keyed by name.

Keys are often strings, and the syntax admits them as a special case:

```
def makeRectangle(=> width, => height):
    return object rectangle:
        to getWidth():
            return width
        to getHeight():
            return height
def rect := makeRectangle("width" => 5, "height" => 7)
```

Default values are supported as well:

```
def adjustAndPad(adjustment, => padding :Str := " "):
    adjust(adjustment)
    pad(padding)
```

Under the hood, Monte accomodates all of this functionality with a very simple change of semantics. In E, a message is [verb :Str, args :List]. However, in Monte, a message is [verb :Str, args :List, namedArgs :Map].

2.1 Locked Methods

The keys can be arbitrary objects. Indeed, consider what happens when a closely-held object is used as a key:

```
object sesame {}
object riches {}
def cave((sesame) => _):
    return riches
```

This function can only be called when the caller passes the right key, and otherwise it will not perform any effects in the method body. Since the key is passed as a named argument, it won't interfere with any other arguments. Therefore, we can apply this pattern to any method. We call this the "locked method" pattern since the method's named parameter is like a lock to which the key is applied.

Locked methods are largely equivalent in functionality to sealed dispatch, where a brand is passed to the ._sealedDispatch() Miranda method. [1] However, there are key differences that hint at different use cases. Locked methods are not as suavely clandestine as sealed dispatch, since they form part of the default alleged interface.

3 Iterators

E has internal iteration. Monte has explicit iterators instead, with an iterable protocol:

```
def counter._makeIterator():
    var i :Int := 0
    return object iterator.next(_ej):
        def rv := [i, i]
        i += 1
        return rv
```

These two methods, ._makeIterator/0 and .next/1, are the entire protocol. Iterators may call the passed-in ejector to end iteration, or they may return a pair. Iteration always proceeds over pairs. The first element of the pair is usually a key or index, while the second element is a keyed value or datum. Iteration over lists yields pairs of indices and values, like Python's enumerate(). Iteration over maps yields key-value pairs.

We can recover classic map-and-filter behavior with list comprehension syntax:

```
def stop := 20
def oddInts := [for i in (0..!stop) ? (i % 2 == 1) i]
```

Here, the "?" should be read as "such-that", just like in such-that patterns, and performs a filtering operation, skipping iterations where the such-that condition fails.

4 Modules

A Monte module is both a compilation unit and a composable code unit. We replace E URI-getters and maker files with a uniform module interface.

4.1 Imports & Exports

At the highest level, a module is a source file with a header which indicates the exports and imports of the module. After compilation, the module can be given a name and composed with other modules by piping exports to imports. The composition is performed by a module loader, which is a plain Monte expression empowered to load code objects.

Module syntax is relatively light. Here is a complete, if facetious, module which imports from "lib/iterators" and exports a single name, "zipSum":

```
import "lib/iterators" =~ [=> zip :DeepFrozen]
exports (zipSum)

def zipSum(left :List, right :List) :List as DeepFrozen:
  return [for [l, r] in (zip(left, right)) l + r]
```

Expanding only the module syntax, the structure of a module is revealed to be a simple anonymous object:

```
object _ as DeepFrozen:
  to dependencies() :List[Str]:
    return ["lib/iterators"]
  to run(package) :Map[Str, DeepFrozen]:
    def [=> zip :DeepFrozen] | _ := package."import"("lib/iterators")
    def zipSum(left :List, right :List) :List as DeepFrozen:
      return [for [l, r] in (zip(left, right)) l + r]
    return [=> zipSum]
```

While arbitrary patterns are allowed in imports, we encourage a convention of map-patterns. Notice that we expand "[=> zip :DeepFrozen]" into "[=> zip :DeepFrozen] | _", which is a convenience that both allows users to import only needed names and also allows for modules to add new exports without breaking users, making it easier to expand a module's interface without breaking compatibility.

4.2 DeepFrozen Exports

In Monte, module exports must conform to DeepFrozen. This constraint forces modules to do any impure or mutable computation prior to freezing and returning exported values.

There are several boons to the DeepFrozen requirement. Like E makers, Monte modules can be cached and composed. Module composition proceeds with the assistance of a module loader, which is a short routine that loads modules and incrementally executes them, routing exports to imports.

There are some difficulties, too. In particular, imported names often must be guarded by DeepFrozen, which is an irritating syntactic burden. The only common exception is importing from the special "unittest" name, which produces an effect on import in order to run unit tests. Such impure effects executed on import don't alter the module formalism at all, just the implementation of the module loader. The impurity of the loader also lets us cache and reuse modules as plain objects, so that modules which occur often in the module dependency graph can be reused.

4.3 Entrypoints

To replace E's URI getters, we instead designate the exported name "main" as an entrypoint for modules. To run an entrypoint, we pass it a message ["run", argv :List[Str], unsafeScope :Map] and expect a Vow[Int]. The return value and list of arguments parallel the classic inputs and outputs to an OS process, while the unsafe scope consists of all of the primitive powers which the loader is willing to grant to the program at large.

This interface permits easy examination of coarse OS-level capabilities, akin to BSD pledge()/tame(). Consider the following entrypoint, from an IRC client:

```
def main(argv :List[Str],
         => Timer,
         => currentRuntime,
         => getAddrInfo,
         => makeFileResource,
         => makeTCP4ClientEndpoint,
         => unsealException) :Vow[Int] as DeepFrozen:...

  return 0
```

We can consider the broad capabilities of this program entirely by considering its unsafe named arguments in the entrypoint. As we do so, we can imagine a code reviewer's rationale for each name:

Timer It can read the system clock and measure time taken during vat turns. This is needed to provide a wall-clock IRC message throttle, so that messages aren't lost and the bot isn't kicked.

currentRuntime It can access various runtime-specific capabilities. This object is meant for staging non-standard platform-specific functionality. In this particular case, Typhon exports a non-standard cryptographic interface based on libsodium via currentRuntime.getCrypt().

getAddrInfo It can resolve hostnames using the system DNS configuration. This is needed to find IRC server addresses.

makeFileResource It can read and write to the filesystem. This is needed for storing todo lists.

makeTCP4ClientEndpoint It can create TCP/IPv4 outgoing connections. This is needed to connect to IRC.

unsealException It can examine exceptions caught by try-expressions. This is needed for REPL functionality.

Most of these unsafe powers have been well-studied, although the ability to unseal exceptions has traditionally not been separated and treated as unsafe.

In addition to the unsafe named parameters, an entrypoint is also passed a list of input strings and should return an integer indicating success or otherwise, as is traditional. One change from tradition, however, is the ability to return a promise for an integer instead. This enables an entrypoint to implicitly indicate that it has more turns pending before it will be finished running. This design is based on Twisted Python's twisted.internet.task.react() helper, which was designed to make it easier to produce Twisted-compatible entrypoints.

5 Controllers

We have experienced use cases for the experimental "lambda-args" syntax suggested for E. In particular, we have found that computations which carry an implicit context or effect can be well-modeled with the syntax, and so we have adapted it for Monte, dubbing this syntax the controller interface.

5.1 Introduction

A controller is a DeepFrozen object which can build a plan incrementally from small program fragments and instructions. The plan is encoded syntactically. A listing is worth a thousand words. Here is a controller in Full-Monte:

```
controller (in0, ..., ini) do arg0, ..., argj { action() }
```

And the same controller, with only controller syntax desugared:

```
(controller :DeepFrozen).control("do", i, j, fn {
  [[in0, ..., ini], fn param0, ..., paramj, ej {
    def [arg0, ..., argj] exit ej := [param0, ..., paramj]
    action()
  }]
}).controlRun()
```

This controller receives a single command, consisting of an instruction "do", an arity i of inputs $[in_0, in_1, \ldots, in_i]$, an arity j of arguments $[arg_0, arg_1, \ldots, arg_j]$, and a thunk known as the block. When called, the block returns a pair containing a list of all i inputs and another thunk, known as the lambda. The lambda is the user-provided part of the plan; when called, it will take in j parameters and run the user-specified action. The controller is expected to generate a plan based on the contents of the block; the plan will probably call the block, extract the inputs, pass the inputs to the lambda, and return the lambda's return value. Since the block and lambda are both thunks, they may be called zero-or-more times based on the design of the controller.

5.2 Monads

Monadic actions can be fully captured by controllers. Compare and contrast the following Haskell:

```
return 42 >>= \x -> do {
  return $ x + 1
} >>= \x -> do {
  return $ x * 2
}
```

With the following Monte:

```
m (m.unit(42)) do x {
  m.unit(x + 1)
} lift x {
  x * 2
}
```

We must pass our monad controller m since we have neither implicit parameters nor typeclasses. The "do" instruction takes some values extracted from the monadic context and returns a monadic action. The "lift" instruction is similar, but performs an automatic unit wrapping to return to the monadic context.

5.3 Kanren

Logic programming can be difficult without a domain-specific language. The Kanren family, including miniKanren and μKanren, is a popular choice for embedding logic programming into general-purpose languages.

We can construct a Kanren program from a controller. This simple program creates two logic variables, x and y, and asserts that $x \equiv \text{true} \wedge x \equiv y$:

```
kanren () exists x, y {
  kanren.unify(x, true)
  kanren.unify(x, y)
}
```

When values are extracted from this program, there will be one satisfying assignment, where both x and y are assigned true.

While the "exists" instruction might be the main workhorse for program fragments, our Kanren controller is also capable of composing logic programs. In this example from real-world code, we compose many fragments of logic at the edges of a sheaf into a single coherent assignment, using the "forAll" instruction to unify a list of logic variables:

```
kanren (fullSection) forAll v, value {
  kanren.unify(vars[index[v]], value)
} (consistency) forAll vs, goalMaker {
  def swizzle := [for v in (vs) vars[index[v]]]
  M.call(goalMaker, "run", swizzle, [].asMap())
```

}

References

[1] Monte miranda protocol. `http://monte.readthedocs.io/en/latest/miranda.html`. Accessed: 2017-08-07.

A Example Monte Module

Listing 1: lib/entropy/pi.mt from the Typhon standard library

```
import "lib/continued" =~ [=> continued :DeepFrozen]
import "unittest" =~ [=> unittest]
exports (makePiEntropy)

# Do not use this module for anything other than testing; it is completely
# deterministic.

def makePiEntropy() as DeepFrozen:
    def pi := continued.pi().extractDigits(2)
    # Move past the first digit (3)
    pi.produceDigit(null)

    return object piEntropy:
        "
        A totally deterministic but random-appearing entropy source based on
        the binary expansion of π.
        "

        to getAlgorithm() :Str:
            return "π"

        to getEntropy():
            return [1, pi.produceDigit(null)]


def testPiEntropy(assert):
    # Vectors generated by, well, pi.
    # 3.14159...
    # -.125
    # =.01659...
    def x := makePiEntropy()
    assert.equal(x.getEntropy()[1], 0x0) # 0.5
    assert.equal(x.getEntropy()[1], 0x0) # 0.25
```

```
    assert.equal(x.getEntropy()[1], 0x1) # 0.125
    assert.equal(x.getEntropy()[1], 0x0) # 0.0625
    assert.equal(x.getEntropy()[1], 0x0) # 0.03125

unittest([testPiEntropy])
```