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: 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. The keys can be arbitrary objects:

```
object namedKey {}
def requireKey((namedKey) => value):
    return value
```

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].

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]
```

4 Modules

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

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.

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])
```

Consider this short real module from the Typhon interpreter's standard library. It imports from two modules, and exports a single name which can be imported elsewhere. While the module is named "lib/entropy/pi" by its filename, that name doesn't appear in the module itself; modules are anonymous to themselves.

Module exports must conform to DeepFrozen. As a result, imported names often must be guarded by DeepFrozen. We can make an exception for unit tests because they can be treated as impure effects executed on import; this doesn't alter the module formalism at all, just 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.

To replace E's URI lookup mechanism, we instead designate the export 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 argu-

ments 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.

5 Controllers

We have experienced use cases for the E experimental "lambda-args" syntax. 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())
}
```