Monte: A Spiritual Successor to E

Corbin Simpson

August 14, 2017

Abstract

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

1 Introduction

Monte is a programming language cast in the same mold as the E programming language. [13] 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 this paper, we will introduce several facets of Monte, including Kernel-Monte expansions and applications to secure distributed computing.

1.1 Surface Syntax

Listing 1: A mint, the "hello world" of capability-safe languages.

```
def makeMint():
  def [sealer, unsealer] := makeBrandPair("mint")
  return def mint.makePurse(var balance :(Int >= 0)):
    def decr(amount :(0..balance)) :Void { balance -= amount }
    return object purse:
      to getBalance() :Int { return balance }
      to sprout() { return mint.makePurse(0) }
      to getDecr() { return sealer.seal(decr) }
      to deposit(amount :(Int >= 0), src) :Void:
        unsealer.unseal(src.getDecr())(amount)
        balance += amount
```

We start with the "hello world" of capability-safe languages, the basic mint. For readers familiar with E, Monte will immediately be readable. The two largest differences are the use of the `object` keyword instead of `def` for object literals, and the allowance of Python-style colons instead of braces for delimiting blocks. Additionally, other minor enhancements are on display in this example. `def mint.makePurse(balance) { … }` is shorthand for `object mint { to makePurse(balance) { … } }`, defining a single-method object. The Monte manual, available at [4], may be helpful.

2 Named Arguments

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

```
def f(x, y, "key" => value):
    return value
```

Keys are usually strings, and the syntax admits them as a special case, allowing `=> argumentName` in place of ``` ``argumentName'' => argumentName```. The same syntax is used for both callers and receivers.

Default values and guards are supported as well, so that callers can optionally omit passing certain named arguments:

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

Listing 2: Making rectangles with named arguments.

```
def makeRectangle(=> width :Int, => height :Int):
    return object rectangle:
        to getWidth() :Int:
            return width
        to getHeight() :Int:
            return height
        to getArea() :Int:
            return width * height
def width := 5
def height := 7
def rect1 := makeRectangle("width" => width, "height" => height)
def rect2 := makeRectangle(=> width, => height)
```

2.1 Semantics

Under the hood, Monte accommodates all of this functionality with a relatively small change of semantics. In E, a message always conforms to `[verb :Str, args :List]`.

However, in Monte, a message conforms to `[verb :Str, args :List, namedArgs :Map]`, with a map of named arguments. When a method is called, after unifying all of the positional arguments, the named arguments are extracted and unified one-by-one.

2.2 Optional Named Arguments

Crucially, any named arguments which are passed but not used by the receiving object are discarded. Callers may use this fact to pass optional named arguments which may or may not be used by the receiver.

Since named arguments can be optional both for callers and receivers, named arguments can be used to ease compatibility concerns when upgrading module code or object versions.

2.3 Locked Methods

The named-argument 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/1` Miranda method. [5] However, there are key differences that hint at different use cases, and in general, we do not recommend ever using locked methods in favor of sealed dispatch. We document the technique here for completeness only.

2.4 Miranda FAIL

Monte grants one Miranda named argument to all calls. The "FAIL" named argument is a callable which takes a single value and aborts the computation. In a turn without a resolver, it is equivalent to `throw.run`, but if there is a resolver r, then it is equivalent to `r.smash`. The purpose of "FAIL" is to ease the effort required to write code which both can error and is often called from both immediate and asynchronous contexts.

3 Iterators

E has internal iteration, where collections are passed a loop body and iterate by calling the loop body. Monte has external iteration and explicit iterator objects instead.

3.1 Rationale

Compared to internal iteration, external iteration with iterators offers several concrete benefits to Monte's design:

- Iterators only take one user-provided callable in their entire API, and that callable should be an ejector. (An idiomatic iterator will use `throw.eject/2` to fire the ejector, ensuring that the callable cannot fool the iterator.) Instead of passing loop bodies to potentially untrusted collections, users can now be confident that an iterator can, at worst, loop forever. As a result, the expansion to Kernel-Monte does not have to generate defensive assertions in for-loops and while-loops to prevent the loop bodies from being stolen by malicious collections, as in E.
- An iterator is slightly more attenuated, in terms of authority, than its corresponding list of pairs. While a list can be examined many times, an iterator can only be consumed once in a linear fashion. Additionally, non-list collections can be iterable, allowing automatic attenuation from iterables to iterators.
- Iterators intrinsically permit pausing iteration. In particular, since iterators must complete their computation within a single turn, concurrency-aware patterns like asynchronously iterating over one item per turn are collection-independent and require no special container code. This means that iterators can represent partially-done computations which would be too expensive to consume in a single turn. For example, Monte Kanren logic programs can be turned into iterables and run incrementally. The Monte parser and lexer use iterables to incrementally parse Monte source code.
- Composition of iterators is relatively easy; a limited version of the classic `zip()` function can be written in fewer than ten lines. A complete version of `zip()` can be found in the Typhon "lib/iterators" module. [7] This version allows arbitrary numbers of arguments and has configurable support for iterables of unequal ("ragged") length.

Listing 3: `zip()` via composition of iterators.

```
def zip(left, right) as DeepFrozen:
  return def zipper._makeIterator():
    def l := left._makeIterator()
    def r := right._makeIterator()
    var counter := 0
    return def iterator.next(ej):
      def rv := [counter, [l(ej), r(ej)]]
      counter += 1
      return rv
```

3.2 Interface

The iterator protocol consists of just one method, `.next/1`, which accepts an ejector and either returns a pair of values or ejects. There is also an iterable protocol, similar to Python's iterable protocol, [1] consisting of another single method, `._makeIterator/0`, which merely makes iterators:

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

These two methods are the entirety of the iterator API.

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.

3.3 Integrating Iterators

Monte syntax uses iterators in the same places where E would use internal iteration. Under the hood, for-loops and while-loops use iterators with the help of the `_loop()` combinator.

We can recover classic mapping-and-filtering behavior with list comprehension syntax. For example, to make a list of odd integers from 0 to 20:

```
[for i in (0..!20) ? (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 named value, "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 single 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 the import map-pattern `[=> 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. This is analogous to our usage of named arguments to allow optional names in calls.

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 usually must be guarded by DeepFrozen . 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 call it with a message `["run", [argv :List[Str]], unsafeScope :Map]`, expecting a `Vow[Int]` in return. 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 during code review, akin to the OpenBSD `pledge()` syscall. [14] Consider the following entrypoint, from a real-world 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 [3] via `currentRuntime.getCrypt()`. This is needed for cryptography services.

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 to-do 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 we highlight the interesting case of `unsealException` and will discuss it more later.

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. [6]

5 Controllers

We have discovered some 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 $[\text{in}_0, \text{in}_1, \ldots, \text{in}_i]$, an arity j of arguments $[\text{arg}_0, \text{arg}_1, \ldots, \text{arg}_j]$, 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. [9, 11]

We can construct a Kanren program from a controller. This short logic 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())
}
```

6 Sealed Exceptions

Monte exceptions are easily thrown. Any object may become an exception by being passed to `throw.run/1`. However, catching an exception does not grant the catcher a reference to the originally-thrown object. Instead, the catcher receives a sealed exception which is opaque.

This asymmetry leads to two distinct tools for non-local control flow. The first tool is the escape-expression. An escape-expression introduces an ejector, and computation may be aborted by firing the ejector. In this way, an escape-expression syntactically indicates where and when control flow will resume, and who gets the result of the inner computation; control flow resumes at the end of the escape-expression and the result is passed to the catcher. The code which holds the escape-expression gets to manage the computation inside of it, and none of the intervening code may catch or otherwise interfere with the ejection.

Now, examining exceptions, we would expect that try-expressions fill the same role as escape-expressions. However, a try-expression does not grant direct access to any caught exceptions, because they are sealed. So, who gets the result instead? In general, we say that the result should be delivered to the caller who caused the current turn, and not to any intervening code inside the turn.

There are several ways to unseal exceptions. Exceptions which reach the end of a turn are unsealed and used to smash the turn's resolver, generating either a broken promise in a subsequent turn, or an exception in the runtime debug log. An exception can be logged in the debug log with the convenience method `traceln.exception/1`, which logs both the exception and its call stack. And, finally, the unsafe scope contains `unsealException`, for cases when an exception must be unsealed for display.

7 Conclusion

We have only skimmed the surface of Monte, but we have demonstrated several interesting facets of its capability-security design.

7.1 Future Directions

- Controllers could be extended in several directions. Possibilities include asynchronous or promise-aware variants of current loop syntax, monad comprehensions, [12] LINQ-style queries, and comonadic ("codo") syntax. [15]

- Work is ongoing to produce a "lib/iterators" library with useful combinators for iterators, with inspiration from the Python "itertools" library. [2]

- Research into ways to manipulate modules is also ongoing. In particular, it may be possible to make a "muffin" module, where subordinate modules have been statically bundled with an entrypoint to create a monolithic program.

- The Monte language and standard library continue to evolve as our contributors explore their own interests. Several contributors have started building alternative runtimes for Monte. We are considering including support for modern capability-relevant technologies like Cap'n Proto [10] and Named Data Networking [8] as standard library modules. Community members have suggested auditors for enforcing algebraic properties, or for enforcing the correctness of primitive recursive arithmetic.

7.2 Acknowledgments

We thank the Monte community at large for many helpful comments made while this paper was being drafted. We also thank the original creators of E for motivating our work. Finally, we thank W. Allen Short for being the other author of Monte, and Emily Dunham for convincing us to write and submit this paper.

References

[1] *Glossary - Python 2.7 documentation*. Accessed: August 8, 2017.

[2] *itertools - Python 2.7 documentation*. Accessed: August 9, 2017.

[3] libsodium. `https://github.com/jedisct1/libsodium`. Accessed: August 14, 2017.

[4] *Monte is Serious Business*. Accessed: August 9, 2017.

[5] *Monte Miranda Protocol*. Accessed: August 7, 2017.

[6] *Twisted API Documentation*.

[7] Typhon mast/lib/iterators.mt. `https://github.com/monte-language/typhon/blob/master/mast/lib/iterators.mt`. Accessed: August 10, 2017.

[8] Named data networking project. `https://named-data.net/wp-content/uploads/TR001ndn-proj.pdf`, 2010.

[9] William Byrd. minikanren. `http://minikanren.org/`. Accessed: August 7, 2017.

[10] Kenton Varda et al. Cap'n proto. `https://capnproto.org/`. Accessed: August 13, 2017.

[11] Daniel Friedman. *The reasoned schemer*. MIT Press, Cambridge, Mass, 2005.

[12] George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. Bringing back monad comprehensions. `http://db.inf.uni-tuebingen.de/staticfiles/publications/haskell2011.pdf`, 2011.

[13] Mark Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, John Hopkins University, 2006.

[14] OpenBSD Foundation. *pledge(2) - OpenBSD manual pages*.

[15] Dominic Orchard and Alan Mycroft. A notation for comonads. `https://www.cl.cam.ac.uk/~dao29/publ/codo-notation-orchard-ifl12.pdf`, 2012.