

Monte: A Spiritual Successor to E

Corbin Simpson

Matador Cloud LLC

October 24, 2017

Overview

- Where did we come from?
- Why are we here?
- Where are we headed?

What is E?

E is a family of lambda calculi with:

- local state
- mutable closures
- managed runtime with:
 - garbage collector, promises, safe scope
 - compiler/interpreter tools
 - communicating event loops (“vats”)

Timeline

- 1993: Original-E
- 1996: Joule
- 1997: E-on-Java
- 2004: Joe-E
- 2005: E-on-CL
- 2008: Caja, Ecrú
- 2012: Experiments (Greyface, Secret)
- 2013: “Original-Monte”
- 2014: Monte-on-Typhon

Whither E?

Knowledge of E is relatively rare. Why? We can only guess:

- People were not ready for E in the 90s
- Capability-security theory remains unpopular
- October 2016: Winter finally arrived

Named Arguments

Monte messages have one additional parameter.

```
[verb : Str , args : List , namedArgs : Map]
```

This enables named arguments, including:

- Optional arguments, for configuration or forward-compatibility
- “Miranda” named arguments provided by the runtime to every call

Sealed Exceptions

Exceptions cannot be examined without `unsealException!`

We can deliver errors to only two places:

- The caller, via ejectors
- The debugger, via exceptions

```
try { throw(42) } catch p { traceln.exception(p) }
```

Changes to Safe Scope

`safeScope` Objects which have no dangerous authority

- Many gratuitous renamings
 - Less Java flavoring, more consistency
- Adjustments for changes to Kernel-Monte
- Iterators
- Int guards the ring of integers
- `b''` and Bytes for bytestrings
- `makeLazySlot` to work around lack of `EventuallyDeepFrozen` for the common case of wanting lazy private state
 - Perhaps memoization is the only other interesting use case?

Iterators

- Small, tight, self-attenuating API
 - `iterable._makeIterator()`
 - `iterator.next(ej)`
- Easy composition
- Incremental computation
 - Iteration can be performed cleanly across many turns
 - The cost of inverting control has been minimal

Motivation for Modules

- Composition is good
- Compilation units are good
- Mutable global state is bad
- Petnames can be ugly, but globally-unique identifiers are bad
 - Filesystem paths are dangerous and should be abstracted away
- Our solution should be boring and easy to bootstrap

Modules on the Outside

- Zero or more export values
 - Exports are selected by name, as in Haskell
 - Exports must pass DeepFrozen
 - Export name `main` is an entrypoint
- Zero or more import patterns
 - Patterns are indexed by petname
 - Each pattern matches against a map of names to import values
 - One module's exports are another module's imports

```
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]
```

Modules on the Inside

- Modules are plain old (zero-magic) Monte objects
 - `module.dependencies() : List[Str]`
 - `module.run(package) : Map[Str, DeepFrozen]`
 - `package."import"(petname : Str)`

```
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]
```

What's Next?

- Slide contents TBD

The End

- Questions?
- Thanks!