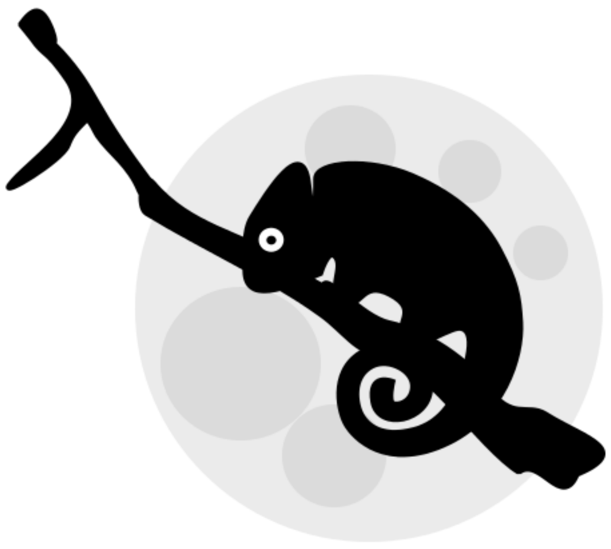
The book of Revelations

Kajetan Rzepecki

2012-11-01



Contents

1 ASVM

1.1 Architecture

1.1.1 Environment

```
+----------------------+
|     Environment      |
|+--------------------+|
||      Compiler      ||
||+------------------+||
|||        VM        |||
||+------------------+||
|+-----------------^--+|
+-------^----^-----|---+
    ^   |    |     |
    |   |    |     |
     ASM    BSM   CSM
```

Source types:

- .asm - high level ASM.
- .bsm - human-readable barebones ASM.
- .csm - raw, compiled bytecode

1.1.2 Compiler

1.1.3 VM

```
Q --- [  ] [  ]...
      +-------+
C --->|       |
E --->|       |
H --->|   S   |
K --->|       |
M --->|       |
      |  ...  |
      +-------+ --- ()
```

Registers & values:

- Q - QValue register
- C - code pointer
- E - environment stack pointer
- H - handler stack pointer
- K - continuation stack pointer
- M - metacontinuation stack pointer
- S - store pointer (memory start)
- () - fnord value (memory end)

1.2 Memory model

1.2.1 Layout

- Tagged memory

```
15 [--|---|--|-|--------] 0
     ^   ^   ^   ^      ^
     |   |   |   |      |_ 8 type/operator bits
     |   |   |   |_ 1 immutability bit
     |   |   |_ 2 cons packing bits (optional)
     |   |_ 3 unused bits (reserved for future use)
     |_ 2 GC bits (may require more)
```

- TValue
  Layout:

```
x86   - [-2-|--4--]     ---> 6
        [-2-|x2x|--4--] ---> 8, word-aligned
amd64 - [-2-|----8----] ---> 10
        [-2-|---6---]   ---> 8, word-aligned, memory-magic
```

  Variants:

```
[TAG|--PTR--] ---> anything that uses pointers
[TAG|--VAL--] ---> anything that uses values
```

  Cons:

  – No big numbers without any special treatment.
  – Interfacing arrays and native calls to D will be complicated.
  – Tag might need additional cons-packing bits.
  – Tag appears twice in a pair.

  Pros:

  – Doesn't need as much padding.
  – No dangling null pointers.
  – Cons packing is trival.
  – Allows for many different kinds of objects to be implemented.
  – Allows other kinds of languages to be implemented.
  – Maps to Lisp well (was used in Lisp machines).

- TBlobs
  Consist of several consecutive QValues.

  Pairs, triples, etc:

```
[pair|--PTR--][fnord|--PTR--]
[triple|--PTR--][fnord|--PTR--][fnord|--PTR--]
```

  Cons packing:

```
[00|--VAL--]               ---> there's no cdr (errors on read/write)
[01|--VAL--]               ---> car is (at) val, cdr is null
[10|--VAL1-][TAG|--VAL2-] ---> car is (at) val1, cdr is at val2
[11|--VAL1-][TAG|--VAL2-] ---> car is (at) val1, cdr is val2
 ^--- Cons encoding part of the tag field.
```

- Tuples and Lists
  Primitives:
  - %car - first pointer of a pair
  - %cdr - second pointer of a pair
- Arrays and Vectors
  Primitives:
  - %ref - array/vector pointer and offset pair
  - %slice - array/vector pointer, start and end pointer triple
- Compound types
  Used to implement sealer/unsealer pattern.

```
# Could be a hash.
(var *compound-type* 0)

# Could use a separate Q type (Type?) and make use of unique references and is? predicate.
(function make-type ()
  (do (var t *compound-type*)
      (set! *compound-type* (+ 1 *compound-type*))
      (tuple t
             (lambda (o)
               (cons t o))
             (lambda (o)
               (if (and (tuple? o)
                        (equal? (car o) t))
                   (cdr o)
                   (error "Type mismatch."))))))
(function typeof (o)
  (when (pair? o)
    (car o)))

(var (T sealT unsealT) (make-type))

(var foo (sealT (tuple 1 2 3)))

# Might facilitate predicate-based type pattern matching.
(function baz (v)
  (case (typeof v)
    (T (unsealT v))
   (X (unsealX v))
    ...))
```

1.2.2 Allocator

1.2.3 Garbage collection

- GC bits

```
[00|--VAL--] ---> unmanaged (pinned)
[01|--VAL--] ---> undecided
[10|--VAL--] ---> undecided
[11|--VAL--] ---> undecided
```

1.2.4 OpCode encoding

Always pairs —> type part of the tag can be used as the operator type.
Example:

```
(%foo bar baz) is...

[%foo|-car-|-cdr-]--->[tag|baz]
        |
        v
        [tag|bar]
```

...instead of...

```
[pair|-car-|-cdr-]--->[pair|-car-|-cdr-]--->[tag|baz]
        |                     |
        v                     v
        [tag|%foo]            [tag|bar]
```

1.3 Threading

1.3.1 Actor model

- Threading
  - %tid - returns current threads ID.
  - %spawn - spawns a thread evaluating given bytecode.
  - %send - sends a bunch of immutable data to a thread.
  - %receive - receives a bunch of data.

1.3.2 Processes

1.3.3 Message passing

1.4 Combinators

1.4.1 Vau calculus

Basics:

```
((vau x e x) foo)         ---> foo
((vau x e e) foo)         ---> dynamic environment
((vau (x y z) e z) 1 2 3) ---> 3
```

Implementing lambda:

```
lambda => (vau (args body) env
            (wrap (eval (tuple 'vau args () body)
                        env)))
```

Primitives

- %vau - creates a lexically scoped operative combinator taking dynamic environment.
- %wrap - induces arg evaluation allowing for applicative combinators.

1.4.2 Argument evaluation

1.5 Formal operational semantics

1.5.1 Environments

1.5.2 Continuations

- @ register
  Instead of value stores the return address where the value should be stored.
- Metacontinuations
  Additional M stack containing continuation segments.

```
(%done @)     (%halt @)
    ^             ^
    |             |
(  ...  )         |
    ^             |
    |             |
(  ...  )         |
          ^       |
           \      |
(%done @)   (%restore @)
    ^             ^
    |             |
    K             M
```

  Primitives:

  - %restore - sets K to the stored continuation stack segment and applies it to the continuation hole.
  - %done - ends the current continuation segment and invokes the M register.
  - %halt - ends the flow of the program.

- Generalized metacontinuations
  Multiple metacontinuation stacks with multiple segments each.

```
(%done @)     (%done @)      (%halt @)    ...
    ^             ^              ^
    |             |              |
(  ...  )         |              |
    ^             |              |
    |             |              |
(  ...  )<--(%restore @)         |
                  ^              |
(%done @)         |              |
    ^             |              |
    |             |              |
(  ...  )         |              |
    ^             |              |
    |             |              |
(  ...  )<--(%restore @)<--(%restore @)
                                 ^
(%done @)     (%done @)          |
    ^             ^              |
    |             |              |
(  ...  )<--(%restore @)<--(%restore @)
                                 ^
(%done @)     (%done @)          |
    ^             ^               |
    |             |               |
(   ...  )<--(%restore @)         |
                 ^               |
(%done @)         |               |
    ^             |               |
    |             |               |
   MK0           MK1             MK2       ...
```

Primitives:

- %restore - pushes a stored continuation stack segment onto the MK register.
- %done - pops the MK register leaving the rest of the meta-stack.
- %halt - ends the flow of the program.

Possible primitives:

- %done-if - premature MK register poping (if @ != ()).
- %select - depending on @ pushes one of its children onto the MK stack.

Notes:

- Might be really cool. Especially because it doesn't require constant consing of the continuation stack.
- All the code can be pre-transformed into dataflow format and then executed with no further transformations.
- Used to implement delimited continuations.

1.5.3 Error handling

1.6 Interfacing with D

1.6.1 Native calls

Implementation:

```
[delegate|ptr|func]---> raw D function pointer
          |
          v
      memory location containing the closure

[native|func]----> raw D function pointer
[nativec|func]---> raw C function pointer
```

Example usage:

```
ASM.define("foo", x => x);
ASM.foo = x => x;
```

1.6.2 Native types

Implementation:

```
[user-type|type|data]---> raw D data
             |
             v
         D typeid
```

Example usage:

```
struct Test {
    int bar;
    string foo;
}

// ...

    ASM.defineType!Test;

    ASM.define("foobar", (scpe, args) {
        if(args.car.type == Type.UserDefined)
        if(args.car.userType == typeid(Type))
        // Do shit
        return ASM.fnord;
    });

    ASM.doString(q{
        (var baz (scope
                    (var _inner (newTest))

                    (function getFoo ()
                      (getTestFoo _inner))
                    (function setFoo (newVal)
                      (setTestFoo _inner newVal))

                    (function getBar ()
                      (getTestBar _inner))
                    (function setBar (newVal)
                      (setTestBar _inner newVal))))
        ((baz setFoo) "Test")
        (foobar baz)
        (writeln (baz getBar))
    });
```

1.6.3 Dynamic FFI

1.6.4 Loading any ASM version

2 ASM programming language

2.1 Phases of evaluation

2.1.1 Lexical analysis

2.1.2 Static analysis

2.1.3 Code generation

2.1.4 Optimisation

2.1.5 Evaluation

2.2 Lexical

2.2.1 Comments

- Think about the comments some more.

  - Metadata
  - Opts
  - Expression comments
  - Shebang parameters problem
  - Multiline comments

2.2.2 Numbers

2.2.3 Symbols

2.2.4 Identifiers

2.2.5 Tuples

2.2.6 Vectors

2.2.7 Arrays

2.2.8 Strings

2.2.9 Reserved keywords & special tokens

2.3 Semantics

2.3.1 Immutability

2.3.2 Numbers and symbols

2.3.3 Booleans

2.3.4 Unit Type

2.3.5 Variables and constants

2.3.6 Applicative combinators

2.3.7 Operative combinators

```
(var lambda (%vau (args body) env
              (%wrap (eval `($%vau $args ignored $body)
                           env))))
(var wrap (%lambda (combinator)
            (%lambda args
              (eval `($combinator $@args)))))
```

2.3.8 Tuples

2.3.9 Vectors

2.3.10 Arrays

2.3.11 Strings

2.3.12 Ranges

2.3.13 Environments

2.3.14 Flow control

2.3.15 Iteration

2.3.16 Pattern matching

```
(match e
       (p1 b1 ...)
```

(p2 b2 . . .) . . .)

- ☐ Binds escaped symbols from pattern to the actual objects.
- ? Escaped symbols = embeded symbols.
- ? Returns a Scope with the symbols defined in it.

2.3.17 Error handling

A pair of condition predicate and condition handler. Signalizing condition invokes iteratively each predicate in the handler stack until it one is true and runs its corresponding handler.

2.3.18 Continuations

2.3.19 Backtracking

- ? Triggered by backtrack expression.
- ? Extended syntax - ?.

3 Code Cube

3.1 cs

3.1.1 core

- Type predicates
  - ☐ ? Accept multiple args.
  - ☒ Evaluate to fnord on false, to one of their args otherwise.
  - ☒ type? - NOT A PREDICATE, returns type tuple of a single object.
  - ☒ fnord? - ‘yup if an expression is fnord.
  - ☒ symbol? - symbol if an expression is a symbol.
  - ☒ number? - number if an expression is a number.
  - ☒ string? - string if an expression is a string.
  - ☒ scope? - scope if an expression is a scope.
  - ☒ function? - function if an expression is a function.
  - ☒ pure? - pure if an expression is a pure function.
  - ☒ syntax? - syntax if an expression is a syntax keyword.
  - ☒ scope? - scope if an expression is repetition, sigh.
  - ☒ builtin? - builtin if an expression is a builtin.
  - ☒ immutable? - immutable if an expression is immutable.
  - ☐ mutable? - mutable if an expression is not immutable.
- Type conversions
  - ☐ ! Convert in place if passed a settable reference, or create a copy.
  - ☐ ? Return fnord on error.
  - ☐ ? string->number - numerical value of a string.
  - ☐ ? string->symbol - returns a symbol version of a string. (both deprecated because of the (read-from-string))
  - ⊟ tupleof:
    - ☒ Makes an immutable tuple version of a passed arg.
    - ☐ ? Should work for atoms aswell.
  - ⊟ listof:
    - ☒ Makes a list representation of an arg.
    - ☐ ? Should work for atoms too.
  - ⊟ setof:
    - ☒ Makes a set representation of an arg.
    - ☐ ? Should work for atoms too.
  - ☒ ! stringof - Makes a string representation of a passed arg.
  - ? etc
- Working with numbers
  - ☐ ASMKit functions accept two arguments.
  - ☐ ! Generic functions built ontop of ASMKit ones, directly in ASM. Return (reduce ASMKitFunc args).
  - ☐ ? Do not use the common operators, so they become redefineable.

  - NEXT ASMKit:
    - ☐ * - a * b
    - ☐ + - a + b
    - ☐ - - a - b
    - ☐ / - a / b
    - ☐ mod - a modulo b
  - NEXT Generics:
    - ☐ sum - generic +
    - ☐ mult - generic *
    - ☐ sub - generic -
    - ☐ div - generic /
    - ☐ modulo - generic mod
    - ? etc
- Equality checks
  - ☐ ASMKit versions taking only two args.
  - ☐ Generic versions returning first arg on true.
  - NEXT ASMKit:
    - ☐ eq? - polimorfic equality check.
    - ☐ leq? - a <= b
    - ? etc
  - NEXT Generics:
    - ☐ ? equal?/=/== - generic equal?
    - ☐ <= - generic leq?
    - ☐ >= - generic ((a eq? b) or (not (a leq? b)))
    - ☐ < - generic ((not (a eq? b)) and (a leq? b))
    - ☐ > - generic ((not (a eq? b)) and (not (a leq? b)))
- import
  Imports symbols, loads modules, manages scopes:

  (import func from ModuleA as AFunc all from ModuleB)
- let
  Immutable let and mutalbe var:

  ```
  (let ((foo bar))
    # foo is immutable
  )

  (var ((foo bar))
    # foo is mutable
  )
  ```
- module/program/class/application etc
  Wraps a bunch of functions and state into a single, named unit:

```
(module Foo
  private (function (foo bar baz)
            (bar baz baz))
  public (var bar)
  export (function (herp derp)
           (derp derp derp)))
```

Dependancy injection:

```
(module Math (printer alocator)
  (function (matrix m n)
    (alocator.malloc (* bar baz)))

  (function (printm matrix)
    (forech e in matrix
            do (printer.print e))))

(import (Math my-logger kewl-alocator))

(Math.printm (Math.matrix 3 3))
```

- case/switch/match/type-dispatch
  Switch-like control structure, with fallthrough + case goto, case ranges etc:

  ```
  (switch a
    case b (foo bar baz)
    case c (faz baz baz)
    default foo)
  ```

3.1.2 memory

- GC
  - ☐ collect! - does a collection.
  - ☐ minimise! - minimises memory use.
  - ☐ disable! - stops GC.
  - ☐ enable! - resumes GCs work.
- Allocator

3.1.3 thread

3.1.4 error

- ☐ (handle e handler) - handler = (error-object handling-function)
- ☐ (raise error-object)
- ☐ warn - runtime warning
- ☐ assert - check condition and rise errors

3.1.5 reader

Based on dynamic PEG parser generator, because it doesn't need separate lexing phase. Reader macros will be grammar based.

Implementation:

```
(syntax (grammar-declarator)
  transform
  ...)

(grammar ((grammar-declarator) transform)
         ...)
```

grammar-declarator:

```
Name arrow Rules
```

transform:

```
Any code, really.
```

Name:

```
Rule name - used inside of it for transforms and outside for parsing.
```

Rules - implicitly wrappend in a sequence:

- (a b c . . .) - sequence
- (/ a b c . . .) - ordered choice
- (* Rules) - zero or more repeats of the Rules
- (+ Rules) - one or more repeats of the Rules
- (? Rules) - optional Rules
- (! Rules) - not Rules
- (& Rules) - and Rules
- (: Rules) - consumes input and drops captures
- (˜ Rules) - concatenates captures

arrow:

- <- - basic
- < - spacing consuming
- <˜ - concatenative

3.1.6 writer

Using pattern matching and string embeds, possibly sewn together with the reader. Migth be of use for the bytecode/crosscode compiler.

3.2 cc

3.2.1 docs

Used for documenting code, using… code in the comments. Something along these lines (needs more work):

```
#? (ASMdoc
#?   This function does some stuff and returns other stuff.
#?   --params
#?       bar - an integer,
#?   --returns - another integer,
#?   --example
#?        (var baz (foo 23))
#? )
(function foo (bar)
  (doStuff bar))
```

3.2.2 test

Automated unittest runner:

```
(unittest Foo
    assert (equal? bar baz)
    assert (foo bar baz)
  test Bar
    assert (foo bar baz)
    assert (foo bar baz)
    log "herp derp"
  test Baz
    assert (bar foo faz)
  finally (derp herp))
```

3.2.3 dbc

function macro - creates a function with all kinds of cool stuff:

```
(function (foo bar baz)
  in (equal? bar 23)
  in (> baz bar)
  out (< result bar)
  body (bar baz))
```

- ☐ erforce - makes sure an operation will succeed.

3.2.4 ranges

- Collection manipulation

  - ⊟ join - if the second argument is a collection - prepends it the the first argument, if it's not a collection - joins both arguments into a pair. Creates a new collection. Examples:
    - (join 1 '[1 2 3]) -> [1 1 2 3]
    - (join '(a b) '[1 2 3]) -> [(a b) 1 2 3]
    - (join 'a 'b) -> (a b)
  - ⊟ append if argument types match - appends element or a collection to another collection, if types don't match - appends the second argument to the collection. Creates a new collection. Examples:
    - (append '[1 2 3] 4) -> [1 2 3 4]
    - (append '[1 2 3] '[4 5 6]) -> [1 2 3 4 5 6]

    * (append ‘(2 3) ‘[2 3]) -> (2 3 [2 3])
  - ☒ first - returns a reference to the first element of a mutable collection, or its value for an immutable collection.
  - ☒ rest - returns a new collection referencing the rest part of the old one.
  - ☒ second, third, fourth etc.
  - ☒ nth - returns nth element of a collection.
  - ☒ map - maps an operation to a collection collecting results.
  - ☒ reduce - maps an operation to a collection reducing it to a single value.
  - ☐ ? slice - slices a collection creating subcollection.
  - ⊟ ? push, push-back, pop, pop-back.
  - ? etc

- Collection creation

  - ☒ list - returns a list consisting of the call args.
  - ☒ tuple - returns a tuple consisting of the call args.
  - ☒ set - returns a set consisting of the call args.
  - ☒ scope - reuturns a scope with call args defined in it.
  - ? etc.

- VLists
  O(log n) indexing. If offset is 0, vlist prealocates additional chunk of data.

```
[offset|list]--->[vector|next]--->[vector|next]--->[vector|next]--->()
   |                |                 |                 |
   |                |                 |                 v
   |                |                 v                 [length|data]--->|0|
   |                v                 [length|data]--->|1|2|
   |                [length|data]--->|3|4|5|6|
   |____________________________________^
```

- vectorize

```
(macro vectorize (fun)
  (let ((old-fun (gensym))
    `(let (($old-fun $fun))
       (function $fun (vec)
         (map fun vec)))))
```

- loop
  Common Lisp like loop macro:

```
(loop for foo in bar
      for baz being each hash-key of goo
      when gaz
      do gar)
```

- for

```
(for x <- foo
     y <- x
     if (> y 23)
     yield y)
```

3.2.5 io

- Input

  - ☐ readln - Unformatted (string) reads.
  - ☐ read - Formatted reads.
  - ☐ ? load/open - Loads a file for reading (as a Scope/Stream with read defined acordingly).
  - ☐ ? close - closes an imput stream.
  - ☐ eof? - returns ‘yup/the object if it has reached EOF.
  - ? etc

- Output

  - ☒ write - writes string representation of the args.
  - ? etc

3.2.6 math

- ☐ sqrt
- ☐ pow
- ☐ exp
- ☐ min/max/clamp
- ☐ etc

3.2.7 random

- ☐ Marsane Twister
- ☐ Gaussian distribution

3.2.8 object

- opElvis syntax

  ```
  foo ? bar == (if-non-fnord foo bar) == (if foo foo bar)
  ```

  Implementation:

  ```
  (macro if-non-fnord (foo bar)
    (let ((_foo (gensym)))
      `(let (($_foo $foo))
         (if $_foo
             $_foo
             $bar))))

  (syntax "\?" (parsed parsing)
    `(if-non-fnord $(pop-front! parsed)
                   $(read-expression! parsing)))
  ```

  Example:

  ```
  # map - 1d or 2d list
  (let ((leny (length map))
        (lenx (length (car map)) ? 1))
    (do-stuff lenx leny) ? 23)
  ```

```
(let ((leny (length map))
      (lenx (if-non-fnord (length (car map))
                          1)))
  (if-non-fnord (do-stuff lenx leny)
                23))

(let ((leny (length map))
      (lenx (let ((__GENSYM0 (length (car map))))
              (if __GENSYM0
                  __GENSYM0
                  1))))
  (let ((__GENSYM1 (do-stuff lenx leny)))
    (if __GENSYM1
        __GENSYM1
        23)))
```

- Dynamic dispatch

  ```
  (defmethod foo (bar baz) body) (foo bar baz) <=> ((get bar 'foo) baz) # Dynamic dispatch
  ```

  or

  ```
  (with bar (foo baz)) # With macro
  ```

  or

  ```
  (function foo (baz) body) (connect bar foo) # Slots
  ```

3.2.9 babel

- JSON

- XML

- SVG
  Returns a wellformed SVG string:

  ```
  (SVG 100 100
       (circle 50 50
               '(255 255 100)))
  ```

- YAML

- LaTeX
  Returns a wellformed LaTeX string:

  ```
  (LaTeX
    "The following equation is herp derp derp:"
    (equation "a^2 + b^2 = c^2")
    (equation "\herp = \derp"))
  ```

- dot
  - ASM AST/module dependancy -> graphviz utility.

- iexpr

```
(package foo
 (function (bar arg0 arg1)
    (if (and (atom? arg0)
             (atom? arg1))
        (* arg0 arg 1)
       (apply + (append arg0 arg1))))
 (var gun (bar 2 3)))

   ||
  \||/
   \/

package foo
 function (bar arg0 arg1)
   if and atom? arg0
          atom? arg1
      (* arg0 arg1)
      apply +
            (append arg0 arg1)
 var gun
     (bar 1 2)
```