DPVM – The Dynamic Predicate Virtual Machine

Gregory T. Sullivan

March 10, 2000

Following is a description of my entry into the "most amount of infrastructure to implement the smallest language" contest, otherwise known as the "dynamic predicate virtual machine".

1 Overview

The overall goal is to provide a high-level virtual machine as a target for language implementation. The main pieces of the system so far are:

DPVM The dynamic predicate virtual machine.

DPVML A Scheme-like language, with first-class types/predicates and set of primitives, that constitutes the "native" programming language for the DPVM.

MiniMMOO A Miniature Multi-Method Object-Oriented language. Intended as a simple example of using the dpvm as a target platform.

There are two tracks in this project, as currently conceived:

1. To create a very high level virtual machine as a platform on which to implement a range of languages, from Java to CLOS to Aspect-Oriented (sub)languages.
2. Implementation techniques to efficiently execute applications written on top of the virtual machine.

DPVML (with a few changes in syntax) could be implemented easily within a Scheme implementation as a bunch of functions. The only interesting built-in datatypes and functions in DPVML (so

far) that are not already in Scheme are predicates, routines to apply predicates to values, and routines to decide predicate implication. This is (or at least would have been) the most straightforward way to play around with implementing high level languages on the DPVM.

In fact, I reimplemented most of Scheme. The only benefits from implementing Scheme from scratch will come when track 2 is pursued – when we want to apply techniques such as partial evaluation and optimistic optimization to applications running on the DPVM.

The point of this note is to put out what I have and get as much feedback as possible from you, the fine and intelligent reader. I want to know what you think: is this heading anywhere interesting? Has this all been done before? Can you think of any good uses of this stuff? What should I do next? Thanks in advance.

2 DPVML – Dynamic Predicate Virtual Machine Language

2.1 DPVML Syntax

DPVML is mostly a subset of Scheme. The binding forms (`lambda`, `let`, `let*`, and `define`) each contain an additional expression that must evaluate to a predicate. There is one more binding form, `pred`, that is basically shorthand for `lambda` with a single argument. The syntax is given in Figure 1.

2.2 DPVML Discussion

Section 5 lists the predefined values and operations in the DPVM.

Operations purposefully not supported by the dpvm are `set-car!` and `set-cdr!`. The primary mutable data structures in the DPVM are cells. Other cases of mutability are the value-preds for a value and implied preds for a predicate may be explicitly extended. Also, bindings in a scope may be updated or added (and I'm not sure how to manage this effectively yet).

Now a list of things that are not in DPVML/DPVM but that should be: threads, vectors (both mutable and stretchable), unwind-protect, variable length arg lists, modules / separate namespaces, and even symbols.

More caveats: the parser for dpvml is too simplistic. Also, many of the predicate combinations have not been well tested. Performance is bound to be abysmal, as I thread the dynamic context through the interpreter. Minimmoo is really trivial.

Some of the interesting forms:

```
Exp ::= (cond (test-exp then-exp) ... [(else else-exp)])
     |  (and exp+)                  -- short circuits
     |  (or exp+)                    -- short circuits
     |  (lambda vars fun-pred-exp body-exp)  -- abstraction
     |  (pred var body-exp)             -- value-pred-exp
     |  (begin exp+)
     |  (let [loop-id] ((var init-exp) ...) pred-exp body-exp)
     |  (let* ((var init-exp) ...) pred-exp body-exp)
     |  (exp exp*)                       -- call-exp
     |  var
     |  number or string literal

Convenience macros (i.e. could be impl'ed in core language):

Exp ::= (if test-exp then-exp [else-exp])
     |  (-> args-pred result-pred-exp)   -- fun-pred
     |  (define var pred exp)            -- def's var. in top scope
     |  (define (var var*) fun-pred body) -- def's function in top scope
```

Figure 1: DPVML Syntax

`cond` like Scheme, except if `else` clause ommitted, as if there was a default `else` clause with an expression of #f.
Ex:
```
(cond ((< x 3) "small")
      ((< x 10) "medium")
      (else "large"))
```

`lambda` Requires a fun-pred (defined later).
Ex:
```
(lambda (x y)
   (-> (list-pred <integer> <integer>) <string>) ;; the fun-pred
   (if (< x y) "x is smaller" "x is not smaller"))
```
defines a function that takes two integer args and returns a string.

`pred` defines a predicate (a boolean-returning function), that is applicable to a value.
Ex: `(pred x (< x 3))` is a predicate that will return true if $x < 3$.

`let` (non looping version) Like Scheme's let binder, with a required predicate.

Ex:
```
(let ((x 3) (y "hello") (z <))
       (list-pred <integer> <string>
         (-> (list-pred <integer> <integer>) <integer>))
       (apply z (list x 2)))
```
Returns the value of `(< 3 2)`, that is, false. Note that the predicate applies to the entire list of values, so if you want to talk about individual values in the list of bound values, as in the example above, you have to use a "list-pred" (defined later).

`let` (looping version) Like Scheme, but again with required list predicate:

Ex:
```
(let loop ((x 3) (out (list)))
          (list-pred <integer> <list>)
          (if (= x 0)
             out
             (loop (- x 1) (cons x out))))
```
will return the list (1 2 3)

`let*` – actually Scheme's letrec – all bound variables can refer to each other.

Ex:
```
(define x <integer> 5)
    (let ((x 3) (y x)) <top> (+ x y))   ; => 8
    (let* ((x 3) (y x)) <top> (+ x y))  ; => 6
```
Ex:
```
(let* ((f (lambda (x y) (-> (list-pred <integer> <integer>) <list>)
                  (if (< x 1)
                     (list "done with f, " x y)
                     (g (- x 1) (+ y x)))))
            (g (lambda (x y) (-> (list-pred <integer> <integer>) <list>)
                  (if (< x 1)
                     (list "done with g, " x y)
                     (f (- x 1) (* y x))))))
        <top>
        (f 7 5))
```
returns the list ("done with g," 0 623).

The core syntax is almost entirely concerned with control flow and binding/abstraction. All of the interesting behavior is defined within primitives.

3 Values

What sort of values may be returned as the value of expressions or bound to variables? Figure 2 lists a bunch of predefined predicates, each of which matches a type of DPVM expressed value.

Where a set of predicates is indented, the outer predicate matches all of the inner predicates. (e.g. a value satisfying `<fun-pred>` will also satisfy `<pred>`):

```
<top>
  <integer>, <boolean>, <string>
  <pred>
    <fun-pred>, <list-pred>, <value-pred>, <elt-pred>,
    <eq-pred>, <top-pred>, <bottom-pred>
  <list>
  <cell>
  <callable>
    <closure>
    <primop>
  <env>
  <exp>
    <cond-exp>, <and-exp>, <or-exp>, <lambda-exp>, <begin-exp>
    <let-exp>, <let*-exp>, <call-exp>, <var-exp>, <literal-exp>
```

Figure 2: Predefined Predicates for DPVM Values

All values travel with a list of predicates that are known to be true of that value. The list of predicates for a value v is returned by the primitive `(value-preds v)`. It is possible to explicitly expand the list of known-true predicates for a value using the `as` primitive operation.

4 Predicates

4.1 Value-pred's

The most basic kind of predicate is a value-pred. A value-pred consists of a function that takes a single value and returns either true or false.

4.2 List-pred's and Elt-preds – predicates for lists

Both list-pred's and elt-pred's are applicable to list values.

A list-pred is a list of predicates. A `list-pred(p_1, p_2, ..., p_n)`, when applied to a list l, will return true if l is of length n, and for each element `e_i` in l, `(p_i e_i)` returns true.

An elt-pred contains a single predicate. An `elt-pred(p)`, when applied to a list `l`, will return true if for each element `e_i` in `l`, `(p e_i)` returns true.

4.3 Fun-pred's – predicates for callable values

A fun-pred has two parts: an args-pred and a result-pred. Applying a fun-pred `p` to a function returns true if the function's fun-pred implies `p`.

All functions (closures or primops) have an associated fun-pred. The function's args-pred is applied to the list of arguments at application time, and the result-pred is applied to the result value of the function's body expression.

4.4 Eq-pred's – singleton types

An eq-pred predicate contains a value. The predicate `eq-pred(v)`, when applied to some value `v'`, will return true iff `(eq? v v')`.

4.5 Top-pred

All values satisfy the top-pred. All predicates imply the top-pred. The top-pred implies no other predicates.

4.6 Bottom-pred

No values satisfy the bottom-pred. No predicates imply the bottom-pred. The bottom-pred implies all predicates.

4.7 Predicate Combination

`and-pred:` `(and-pred pred_1 ... pred_n)` when applied to a value, returns true iff all `pred_i` return true on that value.

`or-pred:` `(or-pred pred_1 ... pred_n)` when applied to a value, returns true if any `pred_i` returns true on that value.

`not-pred:` `(not-pred pred)` – when applied to a value, returns true if pred returns false on that value

4.8 Predicate Application

When a predicate `p` is applied to a value `v`, the following rules determine whether true or false is returned.

1. if `p` is top, return true.
2. if `p` is in the value-preds of `v`, return true.
3. if any of the preds in value-preds of `v` imply `p`, return true.
4. if `p` is bottom, return false.
5. if `p` = `(and-pred p1 p2)`, return true if `(p1 v)` and `(p2 v)`.
6. if `p` = `(or-pred p1 p2)`, return true if `(p1 v)` or `(p2 v)`.
7. if `p` = `(not-pred p1)`, return true if `(p1 v)` returns false.
8. if `p` = `(list-pred p_1 ... p_n)`, return true if `v` is a list of length n and, for each element `v_i` of list `v`, `(p_i v_i)` is true.
9. if `p` = `(fun-pred args-pred result-pred)`, return true if `v` is a closure and the closure's fun-pred implies `p`.
10. if `p` = `(elt-pred p')`, return true if `v` is a list, and for each element `v_i` of list `v`, `(p' v_i)` is true.
11. if `p` = `(eq-pred v')`, return true if `(eq? v v')`.
12. else if `p` = `(value-pred callable)`, return result of application `(callable v)`.

Note that cases 3 and 9 above rely on predicate implication.

4.9 Predicate implication

Predicates may imply other predicates. The implication relationship (other than w.r.t. top and bottom and reflexivity) is seeded by explicitly declaring implication relationships. The two built-in operations for declaring implication are:

`(sub-pred pred-list)` creates a new value-pred whose callable will always return false when applied to a value. However, the newly-created predicate explicitly implies each pred in pred-list. Note that in the rules for predicate application, implication is tested before a value predicate's closure is run.
Ex.: `(pred-implies? (sub-pred (list ... p ...)) p)` = true.

`(add-implies! p pred-list)` adds pairs `(p p_i)` to the implication relationship for each `p_i` in pred-list.

The rules for predicate implication are as follows:

1. p implies p (implication is reflexive)
2. all pred's imply top
3. top implies no other preds
4. bottom implies all preds
5. no pred's imply bottom
6. (implies? (and-pred p1 p2) p3) if
   (implies? p1 p3) or (implies? p2 p3)
7. (implies? (or-pred p1 p2) p3) if
   (implies? p1 p3) and (implies? p2 p3)
8. (implies? (not-pred p1) p2) if
   (implies? p1 (not-pred p2))
9. (implies? (eq-pred v) p) if (p v)
10. (implies? p1 (and-pred p2 p3)) if
    (implies? p1 p2) and (implies? p1 p3)
11. (implies? p1 (or-pred p2 p3)) if
    (implies? p1 p2) or (implies? p1 p3)
12. (implies? (eq-pred v1) (eq-pred v2)) if `(eq? v1 v2)`.
    – `(implies? not-an-eq-pred (eq-pred v2))` = false.
13. (implies? (fun-pred args-pred1 result-pred1) (fun-pred args-pred2 result-pred2)) if
    (implies? args-pred2 args-pred1) and (implies? result-pred1 result-pred2)
    – note: contrapositive in argument types
    – `(implies? (fun-pred ...) not-a-fun-pred)` = false.
14. `(implies? (list-pred p_1 ..., p_n) p)` if
    p = `list(pred p'_1, ..., p'_n)` and each `(implies? p_i p'_i)`.
    – `(implies? (list-pred ...) not-a-list-pred)` = false.
15. (*base case*) (implies? p1 p2), where p1 is a value-pred and p2 is either a value-pred or a not-pred, if:
    (a) p1 implies p2 explicitly, or
    (b) p2 = not-pred(p2') and p2' implies not-pred(p1) explicitly, or
    (c) for `p_i` in predicates explicitly implied by p1, any `(implies? p_i p2)`. (implication is (mostly) transitive)

Note that `implies?` relies on predicate application in the `eq-pred` case (number 9). The dpvm tracks pending implication obligations through both predicate application and implication to avoid infinite loops.

Note that and-, or-, and not- predicates are created in a normalized way – that is,

- `(and-pred p1 (and-pred p2 p3))` rewrites to `(and-pred p1 p2 p3)`
- `(or-pred p1 (or-pred p2 p3))` rewrites to `(or-pred p1 p2 p3)`
- `(not-pred (not-pred p))` rewrites to `p`
- `(and-pred p1 p2)`, where `(implies? p1 p2)`, rewrites to `p1`.
- `(or-pred p1 p2)`, where `(implies? p1 p2)`, rewrites to `p2`.

5 Primitive Values and Operations in the DPVM

(by convention, predicates have names beginning with "`<`" and ending with "`>`")

`#t, #f, *undefined*`

`<top>` the top predicate – every value satisfies `<top>`; `<top>` implies no predicate.

`<bottom>` the bottom predicate – no value satisfies `<bottom>`; `<bottom>` implies every predicate.

`<pred-list>` a list of predicates.

`<list>, <primop>, <cell>, <env>, <string>, <integer>, <boolean>`

`<value-pred>, <list-pred>, <fun-pred>, <closure>, <pred>, <callable>`

`+, -, *, mod` – some (int, int)→int primops.

`<, >, =` – some (int, int)→bool primops.

`eq?` Scheme eq?

`string=?, string-append`

`print` – calls Scheme format to stdout.

`list, list?  list-ref, cons, car, cdr, cadr, reverse, append`

`(list-list-pred l)` returns a cached list-pred for the list l – guaranteed to be correct, if not the most accurate.

`(list-elt-pred l)` returns a cached elt-pred for the list l – guaranteed to be correct, if not the most accurate.

`length, null?`

`not` : value →boolean

`new-cell` : (predicate, value) →cell
`(new-cell p v)` returns a newly created cell with predicate p and initial value v. Predicate p is used to check every value put into the cell (with `set-cell!`).

`deref-cell` : cell →value

`set-cell!` : (cell, value) →value
– tests new value against cell's pred, and if success, updates cell contents and returns new value (same as argument value).

`cell-pred` : cell →pred

`value-pred` : callable →pred
creates a new value-pred with function callable.

`sub-pred` : pred-list →value-pred
– as discussed above, creates new value-pred which answers false when its function is called, but which implies each predicate in pred-list.

`named-sub-pred` : (pred-list, string) →value-pred
– same as sub-pred, but includes a name, for more meaningful debugging.

`and-pred` : (and-pred pred1 pred2 ... predn) →and-pred

`or-pred` : (or-pred pred1 pred2 ... predn) →or-pred

`list-pred` : (list-pred pred1 pred2 ... predn) →list-pred

`list-pred-preds` : list-pred →pred-list

`fun-pred` : (pred, pred) →fun-pred
– called by `->` syntax

`fun-pred-args-pred, fun-pred-result-pred` : fun-pred →pred
– return various components of a fun-pred

`bottom-pred` : () →bottom-pred

`top-pred` : () →top-pred

`eq-pred` : value →eq-pred

`elt-pred` : pred →elt-pred

`pred-implies?` : (pred, pred) →boolean

`add-implies!` : (value-pred, pred-list) →value-pred

`as` : (pred, val) →val
– explicitly updates predicates that this value is considered to satisfy.

`check-pred` : (pred, val) →bool
– returns #t if val satisfies pred (same as predicate application).

`(error format-string arg1 arg2 ... argn)` – fails, and formats to stdout.

`(assert val format-string arg1 arg2 ... argn)` – if val is not #t, does `(error format-string arg1 ... argn)`.

`push-scope` : () →env
– pushes a new scope and returns it.

`current-scope` : () →env

`top-scope` : () →env

`in-scope?` : (string, env) →bool
– true if variable indicated by string is in scope of env.

`deref-var` : (string, scope) →value

`bind-in-scope` : (env, string, pred, val) →val
– updates scope env to contain a binding from variable indicated by string to val. Applies predicate pred first.

`(apply callable val val ... vals)` – last arg must be a list.

`callable-fun-pred, callable-args-pred, callable-result-pred` : callable →pred
– decompose callable

`(curry callable arg1 ... argn)` – returns a callable s.t.
`((curry f v_1 ... v_i) v_i+1 ... v_n) = (f v_1 ... v_n)`

`(rcurry callable arg1 ... argn)` – returns a callable s.t.
`((rcurry f v_i+1 ... v_n) v_1 ... v_i) = (f v_1 ... v_n)`

`compose` : (fun, fun) →fun – `((compose f g) vals)` == `(f (g vals))`

`typed-compose` : (fun, fun) →fun
– same as compose, but for `(typed-compose f g)`, check that result-pred of g implies args-pred of f.

`(map callable list list ... list)` returns a list.

`(every? callable list ... list)` – true if every element of (map callable list ... list) is #t

`any-pred?` : (pred, list) →bool – true if pred is true for at least one element of list.

`(any? callable list ... list)` – true if any element of (map callable list ... list) is #t

`load-dpvml-file` : string →val – loads the indicated file of dpvml code, returning value of last expression.

5.1 Programming in DPVML

Here are two versions of everyone's favorite function:

```
(defin (fact n)
  (-> (list-pred <integer>) <integer>)
  (if (< n 2)
      1
      (* n (fact (- n 1)))))

(define (iter-fact n)
  (-> (list-pred <integer>) <integer>)
  (let* ((f (lambda (n tot)
                (-> (list-pred <integer> <integer>) <integer>)
                (if (< n 2)
                    tot
                    (f (- n 1) (* n tot))))))
    <top>
    (f n 1)))
```

6 MiniMMOOL (Miniature Multi-Method Object-Oriented Language)

Let's implement a simple OO language by translation into DPVML.

6.1 MMMOOL Syntax

```
DefExp ::= (defclass classname (superclass ...) (SlotSpec ...)
        |  (defmethod methname arg-Vars arg-Classnames result-Classname body-Exp)

Exp ::= (let Var ClassName Exp Exp)
     |  (define Var Exp)
     |  (if Exp Exp Exp)
     |  (begin Exp ...)
     |  (list Exp ...)
     |  (Exp Exp-list)          -- generic function application
     |  Var
     |  Constant

SlotSpec ::= (slotname ClassName)
```

Here is some sample code in mmmool:

```
(defclass *foo* (*object*) ((slot1 *object*)))
(defclass *bar* (*object*) ((slot1 *object*)))
(defclass *baz* (*foo*) ((baz-slot *object*)))

(define foo1 (make *foo* (list "foo1")))
(define bar1 (make *bar* (list "bar1")))
(define baz1 (make *baz* (list "baz1-bazslot" "baz1-slot1")))

(slot1 foo1) ;; => ‘‘foo1’’
(slot1 baz1) ;; => ‘‘baz1-slot1’’
(set-slot1 foo1 ‘‘new foo1’’)
(slot1 foo1) ;; => ‘‘new foo1’’

(defmethod f (x) (*object*) *object* "f on *object*")
(defmethod f (x) (*foo*) *object* "f on *foo*")

(f 2)      ;; => ‘‘f on *object’’
(f foo1)   ;; => ‘‘f on *foo*’’
(f bar1)   ;; => ‘‘f on *object’’
(f baz1)   ;; => ‘‘f on *foo*’’
```

6.2 MMMOOL Semantics, by translation to DPVML

The semantics of mmmool is defined (in about 60 lines of Scheme) by direct translation into DPVML. Most of the interesting work happens by functions in the MMMOO runtime (discussed later). For example, a new class definition translates to a call to the function `make-class`, which does all the work. The translation function from mmmool to dpvml is notated `[[-]]` in the following. The interesting cases are:

```
[[(defclass classname (super1 ...) ((slot1 class1)  ...)]]
 = (define ,classname <class>
      (make-class ``,classname'' (list ,super1 ...)
         (list (list ``,slot1'' class1) ...)))

[[(defmethod methname (arg-var ...) (arg-class ...) result-class body-exp)]]
 = (add-method-to-name ``,methname''
      (lambda (arg-var ...)
         (-> (list-pred (class-pred arg-class) ...)
             (class-pred result-class))
         [[body-exp]]))

[[(Exp0 Exp1 ...)]]
 = (apply-generic [[Exp0]] (list [[Exp1]] ...))
```

6.3 MMMOO Runtime

The mmmoo runtime is written in (about 230 lines of) DPVML, and defines a bunch of predicates and functions. The important predicates include `<object>`, `<class>`, `<method>`, `<generic>`, `<slotspec>`, and `<slot>`.

A class is represented by a three-element list: (class-pred, superclasses, slotspecs), and the list is cast (via `as`) to satisfy the `<class>` predicate. So every class is a triple, and every class has an associated predicate. A class' predicate is created, using `sub-pred`, to explicitly imply the predicates of its superclasses.

An instance (aka object) is a pair consisting of the class of the object and a list of slots. A slot is represented by a cell, as slots are mutable. A generic function is represented by a pair of a predicate (not currently used) and a cell containing a list of methods for the generic.

When a generic function is applied, the dpvm function `apply-generic` is invoked. It first finds all applicable methods, by calling `applicable?`, then finds the most applicable method by calling

`find-mam`. The code for all three functions is below. I think that they give a good idea of what programming in dpvml is like – namely, it is like programming in Scheme.

```
(define (apply-generic gf args)
  (-> (list-pred <generic> <list>) <top>)
  (let ((applicable-methods
         (filter (rcurry applicable? args) (generic-methods gf))))
    (list-pred <method-list>)
    (if (null? applicable-methods)
        (error "No applicable methods for function ~a called with ~a" gf args)
        (let ((mam (find-mam applicable-methods args)))
          (list-pred (or-pred <list> <method>))
          (if (list? mam)
              (error "Ambiguous most applicable methods for call ~a(~a). Methods = ~a"
                     gf args mam)
              (apply mam args))))))

(define (applicable? method args)
  (-> (list-pred <method> <list>) <boolean>)
  (every? check-pred (method-arg-preds method) args))

;; Given a non-empty list of methods known to be applicable to args,
;; find the most applicable method or methods and return it/them.
;; If a list of methods is returned, rather than a single method, then
;; the application if ambiguous.
(define (find-mam meths args)
  (-> (list-pred <method-list> <list>) (or-pred <list> <method>))
  (let loop ((meths (cdr meths))
             (mams (list (car meths)))) ; candidates for mam (all elts mutually ambiguous)
    (list-pred <method-list> <method-list>)
    (if (null? meths)
        (if (= (length mams) 1)
            (car mams)
            mams)
        (let ((m (car meths))
              (m-args-pred (callable-args-pred (car meths))))
          (list-pred <method> <pred>)
          ;; Cannot have situation where both m is < some method m' in mams
          ;; AND m is > some method m'' in mams -- that would imply that
          ;; m' and m'' are comparable and therefore not mutually ambiguous.
          (cond
           ;; if m > any m' in mams, m is not a mam candidate
           ((any? (rcurry pred-implies? m-args-pred)
                  (map callable-args-pred mams))
            (loop (cdr meths) mams))
           ;; if m < any m' in mams, replace all such m' with m
           ((any? (curry pred-implies? m-args-pred)
                  (map callable-args-pred mams))
            (loop (cdr meths)
                  (cons m (filter (lambda (m1) (-> (list-pred <method>) <boolean>)
                                          (not (pred-implies? m-args-pred
                                                              (callable-args-pred m1))))
                                  mams))))
           ;; otherwise, must be incomparable with all elts of mam, so add m
           (else
            (loop (cdr meths) (cons m mams)))))))))
```

The other interesting function, as you might guess from the translation, is `make-class`.

7 Playing Around – some REPL's

Go to a directory with all the .scm and .dpvml sources for DPVM and friends. On the AI file system, look in `/home/ai/gregs/projects/dpvm/release/` Fire up a scheme48 image, with some extra heap. The following will work on ernie (a linux box):

```
/usr/bin/scheme48  -h 8000000
```

Load some scheme48 packages:

```
,open big-scheme defrecord
```

Answer "y" to the "load structure" questions.

Now load Scheme code for the DPVM:

```
,load utils.scm
,load dpvm.scm initialize.scm predicates.scm dpvml.scm contexts.scm
,load mmmoo-front.scm
```

You can get into a REPL (read-eval-print-loop) for DPVML by typing `(dpvm-repl)` at the Scheme prompt.

You can get into a Minimmool REPL by typing `(mmmoo-repl)` at the Scheme prompt. When in the mmmoo-repl, you can drop down to the DPVML level by typing `dpvml` at the mmmoo prompt. To pop back up, just type `bye` or hit Ctrl-D and answer "y".

To leave Scheme, either type ",exit" at the Scheme prompt or hit Ctrl-D and answer "Y".

8 Ideas, Open Issues, Notes

I think it would be really cool to support dynamic, multiple dispatch at the level of the dpvm. Presumably, dispatch à la Dylan, CLOS, and Java could be implemented fairly directly via dpvm primitives.

On the other hand, trying to get rid of dynamic dispatch overhead in the runtime of minimmoo is an appealing target for partial evaluation techniques.

Currently, environments are accessible and mutable. This seems obviously not a good long term plan. For example, it immediately breaks the notion of beta substitution – the binding of a value to a variable can be changed *during* execution of a closure body! On the other hand, futzing with scopes is necessary for language implementation.

Currently in dpvml, applying a pred actually returns the value on success and invokes failure cont. on failure.

Control over when predicates fail – grab/change the failure continuation.