DVM – A Dynamic Virtual Machine

Gregory T. Sullivan

June 30, 2000

1 Overview

The overall goal is to provide a high-level virtual machine as a target for language implementation. The machine should support traditional, static languages such as Java, as well as dynamically-typed languages such as Scheme, Lisp, and Dylan. The machine should support dynamic redefinition and runtime MOP's, yet be able to optimize effectively when these features are not used.

The main pieces of the system so far are:

DVM The dynamic virtual machine.

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

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.

2 DVML – Dynamic Virtual Machine Language

2.1 DVML Syntax

DVML is very similar to Scheme. Where features of Scheme are missing, it is because I haven't needed them yet. The main addition to Scheme is that wherever in Scheme you would specify a variable (i.e. the LHS of a let binding or as lambda argument), you may specify a (variable, predicate) pair instead, where the predicate denotes a type constraint on the contents of the variable. More on predicates later.

Another difference from Scheme is that DVML does not support the SET! assignment operator. DVML has first-class cells, which may be dereferenced and updated. DVML also does not support SET-CAR! and SET-CDR! operations.

A minor difference is that rest arguments are specified via a ":" token, instead of "." – this is because the current implementation, out of laziness, uses the Scheme read function, and it likes to interpret the dot. Another minor difference is that LET* is really LETREC – all the definitions are mutually recursive.

Finally, there are first-class generic functions, which are invoked using the GF-CALL operator.

The syntax of DVML is given in Figure **??**.

Section **??** lists the predefined values and operations in the DVM.

3 Values

What sort of values may be returned as the value of expressions or bound to variables? Figure **??** lists a bunch of predefined predicates, each of which matches a type of DVM 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>`).

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. In the following, the `(pred v ...)` form specifies a new value predicate:

```
(define <even> <pred>
  (and-pred <integer> (pred v (= 0 (mod v 2)))))
```

```
Exp ::=
     |  (if test-exp then-exp [else-exp])
     |  (pred-if pred-exp val-exp then-exp [else-exp])
     |  (and and-exps)                   -- short circuits
     |  (or or-exps)                     -- short circuits
     |  (lambda Vars [result-pred] body-exp)
                 -- ex: (lambda (x (y <integer>)) <integer> (+ x y))
                 -- ex: (lambda (x (y <integer>) : (z <string-list>) ...)
     |  (pred var body-exp)              -- value-pred-exp
     |  (begin exps)
     |  (let [loop-id] ((Var init-exp) ...) body-exp)
                 -- ex: (let (((x <integer>) 2) (y 3)) ...)
     |  (let* ((Var init-exp) ...) body-exp)
     |  (gf-call exp exps)                -- generic function call
     |  (exp exps)                        -- call-exp
     |  var
     |  number or string literal

                The single Var after the ":" gets the rest-vals:
Vars ::= (Var* [: Var])      -- ex: (x (y <integer>) : (z <int-list>)
Var ::= Id | (Id Id)

convenience macros:

     |  (cond (test-exp then-exp) ... [(else else-exp)])
     |  (-> args-pred-exp result-pred-exp)   -- fun-pred
     |  (define var pred-exp exp)         -- binds variable var in top scope
     |  (define var exp)                  -- same, with <top> predicate
     |  (define (var Vars) [result-pred] body) -- binds fun named var in top scope
```

Figure 1: DVML Syntax

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

Figure 2: Predefined Predicates for DVM Values

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 is applicable to a `<callable>` and has three parts:

- arg-preds – a list of predicates, one for each formal argument,
- rest-pred – an optional predicate – if non-false, then the function accepts additional values (i.e. beyond the formal arguments), and the list of additional values must satisfy the rest-pred.
- result-pred – any value returned by the function must satisfy the result-pred.

In the long term, I'd rather use a dot, but first I need to write my own reader. Applying a fun-pred `p` to a function returns true if the function's fun-pred implies `p` (predicate implication is presented in Section **??**).

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 **??** and **??** 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 transitive)

Note that `implies?` relies on predicate application in the `eq-pred` case (number **??**). The dvm 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 DVM

(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.

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

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

<pred-list>, <string-list>, <list-list>

<exp>, <if-exp>, <fun-exp>, <seq-exp>, <let-exp>

<let*-exp>, <call-exp>, <var-exp>, <literal-exp>, <gf-call-exp>, <fast-call-exp>

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

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

eq? Scheme eq?

string=?, string-append, string-remove

print – calls Scheme format to stdout.

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

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. Called by `(pred v ...)` syntax.

`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-arg-preds, fun-pred-result-pred` : fun-pred →pred

– return various components of a fun-pred

`fun-pred-rest-pred/f` : fun-pred →false-or-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.

`(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.

`static-pred` : (string, scope) →pred
– for a given scope, the declared (or inferred) predicate for a binding.

`set-static-pred!` : (string, pred, scope) →pred

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

`callable-arg-preds, callable-result-pred` : callable →pred
– decompose callable

`callable-rest-pred/f` : callable →false-or-pred

`(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))`

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

`generic` : pred →generic – creates a new generic function, with the indicated predicate.

`generic-functions` : generic →callable-list

`add-fun` : (generic, callable) →generic – adds the callable to the generic.

5.1 Programming in DVML

Here are four versions of everyone's favorite function:

```
(define (fact (n <integer>)) <integer>
  (if (< n 2)
      1
      (* n (fact (- n 1)))))

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

(define (loop-fact n)
  (let loop ((n n) (tot 1))
    (if (< n 2)
        tot
        (loop (- n 1) (* n tot)))))

(define (reduce fn initval : lists)
  (let loop ((lists lists) (val initval))
    (if (null? (car lists))
        val
        (loop (map cdr lists)
              (apply fn (cons val (map car lists)))))))

(define (reduce-fact n)
  (let ((list-to-n (let loop ((l (list 1)) (i 2))
                     (if (<= i n)
                         (loop (cons i l) (+ i 1))
                         l))))
    (reduce * 1 list-to-n)))
```

6 Playing Around – some REPL's

Go to a directory with all the .scm and .dvml sources for DVM and friends. On the AI file system, look in `/home/ai/gregs/projects/dvm/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 DVM:

```
,load utils.scm
,load dvm.scm dvml.scm initialize-dvm.scm predicates.scm contexts.scm
```

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

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

7 Ideas, Open Issues, Notes

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 dvml, 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.