# Freeing the Essence of a Computation[1]

Kenneth R. Anderson
**BBN Systems and Technologies, 10 Moutlon St. Cambridge, MA, 02138**
**KAnderson@bbn.com**

In theory, abstraction is important, but in practice, so is performance. Thus, there is a struggle between an abstract description of an algorithm and its efficient implementation. This struggle can be mediated by using an interpreter or a compiler. An interpreter takes a program that is a high level abstract description of an algorithm and applies it to some data. Don't think of an interpreter as slow. An interpreter is important enough to software that it is often implemented in hardware. A compiler takes the program and produces another program, perhaps in another language. The resulting program is applied to some data by another interpreter.

Partial evaluation is a less widely known technique that is between interpretation and compilation.[2] A partial evaluator applies a program to part of its input data. As much of the program as possible is executed and the result is a simplified program that can be applied to the remaining data. Thus, the partial application:[3]

```
(partial-apply (lambda (a b) (if (> a 5) (+ b 3) (+ b 2)))
         (list 7 (make-placeholder 'b)))
```

would have the result:

```
(lambda (b) (+ b 3))
```

Where `(make-placeholder 'b)` returns an object that represents the value of `b` that will be supplied later (this is explained further below). Since `partial-apply` applies a program, it is an interpreter. However, since it produces a program, it is also a compiler. Thus by executing your program, compiled code is produced. A minipartial evaluator is relatively easy to write, similar in effort to writing minicompilers for specialized languages (see for example [BA] and [VM]). If your application is object oriented, it may require little more than a few additional methods.

This article introduces the basics of partial evaluation. First an abstract, object oriented library for manipulating points on the earth's surface is developed. Then a simple partial evaluator is developed which provides an efficient implementation of the library. The partial evaluator is also object oriented and works with compiled code. This suggests that augmenting an object oriented library with a partial evaluator can be quite powerful, and several other examples are outlined. An object oriented partial evaluator provides the programmer with disciplined access to compiled code, thus it could be used as part of an Open Implementation [KMW]. We demonstrate this by sketching how one might provide compile time access to Common Lisp's `slot-value` protocol.

## Capturing the Essence of a Computation

When computing the distance or azimuth between two points on the earth, formulas from spherical trigonometry are often used (see Exhibit 1).[4] Here `lat1`, `lon1` and `lat2`, `lon2` are latitude and longitude pairs (in radians). While these functions are trivial to compute on today's computers, they were quite complex for seismologists locating earthquakes in the 1920's, the days of hand cranked calculators. Then, sines and cosines required table lookup and interpolation. A faster approach was to represent a point on the earth's surface as a three dimensional unit length vector [BU]. While this required three numbers rather than two, it allowed distance to be calculated by a dot product (see `vdot` below) and one table lookup, rather than six. Thus, it was an excellent space time tradeoff. As an aid to seismologists, the International Seismological Centre published a

---

[1] This is the second of a series of articles on Lisp performance, called "Performing Lisp".

[2] Peter Norvig's excellent book on Lisp and AI devotes only one paragaraph to partial evaluation [NO, p. 267]. Luckily, it references [BW] on which this article is based.

[3] Software examples are given in a Scheme like dialect of Common Lisp where `define` replaces `defvar`, `defun`, and `defmethod`. While this may be confusing to some, the idea is to keep the software abstract and free of unnecessary detail, since the point is to show how such a language can be easily and efficiently implemented. The ideas expressed here should port to other languages as well, perhaps with a bit more work. For a partial evaluator in Fortran, see [KKZG].

[4] Shortly, we will make a first order correction for the fact that the earth is not a sphere.

table of unit vectors of seismic stations until 1971. By then, electronic computers were widely available and the spherical formula was widely used. This is unfortunate, since not only is the unit vector approach more efficient, it also allows problems to be solved using vector geometry rather than spherical geometry.

```
;;; Exhibit 1 - Spherical distance and azimuth.
(define (spherical-distance lat1 lon1 lat2 lon2)
  (acos (+ (* (cos lat1) (cos lat2) (cos lon2))
           (* (sin lat1) (sin lat2)))))

(define (spherical-azimuth lat1 lon1 lat2 lon2)
  (atan (* (sin (- lon2 lon1)) (cos lat2))
        (- (* (cos lat1) (sin lat2))
           (* (sin lat1) (cos (- lon2 lon1)) (cos lon2)))))
```

We start by providing some vector algebra primitives (specialized for three dimensions, though this is not essential [BW]). `Vref` is the vector access primitive. Vectors are immutable, so the application is strictly functional.

```
;;; Location primitives
(define (vmake x y z) (vector x y z))   ; Make a location vector.


(define north (vmake 0.0 0.0 1.0))      ; North Pole.

(define (vdot v1 v2)                    ; Dot product.
  (+ (* (vref v1 0) (vref v2 0))
     (* (vref v1 1) (vref v2 1))
     (* (vref v1 2) (vref v2 2))))

(define (vlength v) (sqrt (vdot v v)))  ; Euclidian length.

(define (vscale v s)      ; Scale V by S.
  (vmake (* (vref v 0) s) (* (vref v 1) s) (* (vref v 2) s)))

(define (vnormalize v) (vscale v (/ 1 (vlength v)))) ; Make V unit length.

(define (vcross a b)                    ; Cross product.
  (vmake
   (- (* (vref a 1) (vref b 2)) (* (vref a 2) (vref b 1)))
   (- (* (vref a 2) (vref b 0)) (* (vref a 0) (vref b 2)))
   (- (* (vref a 0) (vref b 1)) (* (vref a 1) (vref b 0)))))

(define (v+ a b)           ; Add A to B.
  (vmake (+ (vref a 0) (vref b 0))
         (+ (vref a 1) (vref b 1))
         (+ (vref a 2) (vref b 2))))
```

We define an object to hold the information needed to correct for the earth's flattening (the radius through the poles is less than the radius through the equator). Thus the software is easily customized for other spheroidal planets:

```
(defclass planet ()
  ((radius     :initarg radius     :reader radius)
   (flattening :initarg flattening :reader flattening)))

(define earth (make-instance 'planet
                'radius 6378.137 ; In km (WGS 84 [HFJH]).
                'flattening (/ 1 298.257223563)))
```

Next, we make a first order correction to latitude. This shifts the latitude by as much as 20 nautical miles in the middle latitudes (around +45 or -45 degrees).

```
(define (geocentric-latitude planet geographic-latitude)
  (atan (* (expt (- 1.0 (flattening planet)) 2) (tan geographic-latitude))
        1.0))

(define (geographic-latitude planet geocentric-latitude)
  (atan (/ (tan geocentric-latitude) (expt (- 1.0 (flattening planet)) 2))
        1.0))
```

Now, we can convert between geographic (latitude and longitude) and the unit vector representation:

```
;;; Convert to/from radians/degrees
(define (radians x) (* x (/ pi 180)))
(define (degrees x) (* x (/ 180 pi)))

(define (geo->v planet lat lon)   ; Convert lat,lon to vector.
  (let* ((theta (radians lon))
         (rlat  (geocentric-latitude planet (radians lat)))
         (c (cos rlat)))
    (vmake (* c (cos theta)) (* c (sin theta)) (sin rlat))))

(define (v->geo planet v)         ; Convert from vector to lat,lon.
  (values (degrees
           (geographic-latitude planet
            (atan (vref v 2) (vlength (vmake (vref v 0) (vref v 1) 0.0)))))
          (degrees (atan (vref v 1) (vref v 0)))))
```

Once we are in the unit vector representation, distance and azimuth are easily computed. Since the dot product of two unit vectors is the cosine of the angle between them, and the cross product is the sine, the angular distance (in radians) between two points is:

```
(define (vdistance v1 v2)          ; Angular distance between V1 and V2.
  (atan (vlength (vcross v2 v1)) (vdot v2 v1)))

(define (vazimuth v1 v2)           ; Azimuth from V1 to V2.
  (let* ((n1 (vcross north v1))
         (s1 (vlength n1)))
    (if (< s1 single-float-epsilon) 180.0      ; V1 is north pole.
        (let* ((n2 (vcross v2 v1))
               (az (degrees (atan (- (vdot north n2)) (vdot n1 n2)))))
          (if (> az 0.0) az (+ 360.0 az))))))
```

Computing azimuth is only a bit more complicated. First, a great circle (the intersection of the sphere and a plane that passes through its center) can be represented by a unit vector, `g`, normal to it. Any point, `x`, on the great circle satisfies the two equations:

```
(dot g x) = 0; (vdot x x) = 1
```

Since the cross product of two vectors is normal (perpendicular) to both, it can be used to represent the great circle between them. Thus, `n1` is the normal to the great circle between `north` and `v1` (the meridian of `v1`). Similarly, `n2` is the normal of the great circle containing `v1` and `v2`. The angle between `n1` and `n2` is the azimuth, though we can avoid the explict cross product. Also, `vazimuth` must check to return a positive azimuth. Given a database of locations, we can compute some freqent flyer kilometers, as show in Exhibit 2. Some people still use the geographic representation, so we provide a convenient function for them:

```
(define (distance-and-azimuth planet lat1 lon1 lat2 lon2)
  (let ((v1 (geo->v planet lat1 lon1))
        (v2 (geo->v planet lat2 lon2)))
    (values (* (radius planet) (vdistance v1 v2 ))
            (vazimuth v1 v2)))))
```

As a final example, `vbetween` returns the point on the great circle between `v1` and `v2` that is the fractional distance, `x`, between them:

```
(define (vbetween v1 v2 x)
  (vnormalize (v+ (vscale v1 x) (vscale v2 (- 1.0 x)))))
```

We now have a working library of software that is compact (about 60 lines), and follows the underlying math well enough that little documentation is needed beyond the code itself. However, it is slow, even slower than the software it was trying to replace. For example, in a call to `distance-and-azimuth`, there are many function calls, three vectors created, and lots of generic math done. The poor performance has nothing to do with the language the application is written in, but with its implementation.

We could make many hand optimizations as the following table suggests:

| **Performance Problem** | **Optimization** |
|---|---|
| Generic math operations | Declare variables |
| Function call overhead | Inline functions |
| Heap vector allocation | Stack vector allocation |
| Vector creation overhead | Pass in a vector to put results into |
| Vector access overhead | Spread vector components into arguments |

Until we make all these optimizations, the performance will be suboptimal. Unfortunately, once they are made the software is a fair distance from the original abstraction, and is significantly more complicated. Worse, we must apply these optimizations by hand to the entire library and any future extension.

```
;;; Exhibit 2- International Civil Aviation Codes for selected airports:
(define ENFB (geo->v earth 59.8947   10.6189))      ;  Oslo Norway.
(define KBOS (geo->v earth 42.3636  -71.0061))      ;  Boston MA.
(define PHNL (geo->v earth 21.3192 -157.9294))      ;  Honolulu.

? (* (radius earth) (vdistance kbos enfb)) -> 5636.911771058637
? (vazimuth kbos enfb)                     -> 40.17142339551045
? (* (radius earth) (vdistance kbos phnl)) -> 8210.330778139354
? (vazimuth kbos phnl)                     -> 284.1243367750494
```

## Trace is a Compiler

Luckily, partial evaluation will provide these optimizations for us. To motivate the idea, suppose we are interested in optimizing a function that computes the length of a cross product vector, `v`, with the vector `north`, as is done in the azimuth computation:

```
(define (northward v) (vlength (vcross north v)))
```

Since only one component of `north` is nonzero and the vector produced by the cross product is temporary, one can optimize this by hand to a few operations using algebraic substitution. An alternative is to run `northward` on some data while tracing the math primitives (see Exhibit 3). The components of the vector are chosen so they can be easily distinguished in the trace. Only the operations marked by a "!" are useful, and an efficient function can be constructed from the trace by inspection, as shown on the right. Even though computing `(- y)` isn't necessary, this isn't bad software considering it was written by t`race`.

## Freeing the Essence of a Computation

This works because the traced functions record the computation directly involving the input variables (represented by 2, 16, and 512). The untraced computation involves transient things like function calls, vector creation and access that aren't involved in the final result. Unfortunately, this tracing approach only works for a "straight line" program, because it would only trace one branch of an if statement. This will be handled below.

The partial evaluation approach explored by Berlin and Weise [BW] generalizes this idea. The function to be partially evaluated, such as `northward`, is called with arguments that are normal values, if they are known, or placeholders, if they are not. The math operations are modified to deal with placeholders. Operations that cannot compute a value at partial evaluation time place a placeholder on a list used to produce a binding list of a `let*` expression as in the example above. We outline the software required to follow this recipe.

```
;;; Exhibit 3 - Using trace as a compiler
? (trace + - * / sqrt)
NIL
? (northward (vmake 2 16 512))          (define (northward x y z)
  (* 0.0 512)     -> 0.0                  (let* (
  (* 1.0 16)      -> 16
! (- 0.0 16)      -> -16.0                       (a (- y))
  (* 1.0 2)       -> 2
  (* 0.0 512)     -> 0.0
  (- 2 0.0)       -> 2
  (* 0.0 16)      -> 0.0
  (* 0.0 2)       -> 0.0
  (- 0.0 0.0)     -> 0.0
! (* -16.0 -16.0) -> 256.0                      (b (* a a))
! (* 2 2)         -> 4                          (c (* x x))
  (* 0.0 0.0)     -> 0.0
  (+ 4 0.0)       -> 4
! (+ 256.0 4)     -> 260.0                      (d (+ b c)))
! (SQRT 260.0)    -> 16.1245154965971           (e (sqrt d))
16.1245154965971                              e))
```

Depending on your language, you may not be able to modify the mathematical operations directly. For example, in Common Lisp, the operations `+`, `-`, `*` and `/` cannot be redefined in the `common-lisp` package, but can be in another package. In Scheme, these operations can be redefined as long as primitive integration is turned off. In C++, the operators can be overwritten and classes must be defined to shadow built in types, such as numbers. In the worst case, the application level software can be rewritten so that math operations can be renamed, so + might become `plus`, for example. In the following, this step has already been taken — math operations can be overloaded, and actual primitive operations are prefixed with a "`%`".

Extending a unary operation to handle a placeholder is straightforward. The function `emit` takes "a piece of code" that can compute the value of the operation. A list structure is convenient for this.

```
(define (cos (a number)) (%cos a))
(define (cos (a placeholder)) (emit `(cos ,a)))
```

An operation should attempt to compute as much of its value as possible. The operations + and `*` take any number of arguments, and `-` and `/` are written in terms of them respectively. The software for `+` and `-` looks like:

```
(define (+ &rest args) (placeholder-reduce-+ (sort args #'placeholder-<)))

(define (- &rest args)
  (cond ((null? args) (error "not enough arguments."))
        ((cdr args)  (--2-arg (%car args) (apply #'+ (cdr args))))
        (t (--2-arg 0.0 (%car args)))))

(define (--2-arg (a number) (b number)) (%- a b))
(define (--2-arg (a number) (b placeholder)) (emit `(- ,a ,b)))
(define (--2-arg (a placeholder) (b number)) (%if (zero? b) a (emit `(- ,a ,b))))
(define (--2-arg (a placeholder) (b placeholder))
  (%if (eq? a b) 0.0 (emit `(- ,a ,b))))
```

The operator `+` sorts its arguments, using `placeholder-<`, so that numbers appear before placeholders. The function `placeholder-reduce-+` then reduces the value of the `+` to minimum form. The operator `-` is written in terms of `+` and a two argument version, `--2-arg`.

Exhibit 4 shows the core functions of the partial evaluator. The functions `peval` and `emit` manage the current binding environment. `Peval` is analogous to `eval`, but instead of taking a symbolic expression, it takes a thunk (a function of no arguments). This lets it work with compiled software.

`Bindings` is a stack of lists of placeholders. The topmost list of placeholders is the list of l`et*` bindings currently being formed. Lower level lists are for calls to `peval` already in progress recursively. `Peval` adds a

new binding list, and funcalls its thunk. If the returned value is a placeholder, it is combined with the let bindings and returned as a new placeholder. Otherwise, a normal Lisp value is returned.

```
;;; Exhibit 4 - Core of the partial evaluator.
(define bindings '())    ; List of lists of placeholders.

(define (peval thunk)
  (set! bindings (cons '() bindings))
  (let* ((result (multiple-value-list (funcall thunk))))
    (%if (cdr result) (values-list result)
        (begin
          (set! result (car result))
          (when (placeholder? result)
            (when (%eq? result (%car (%car bindings)))
              (set! result
                    (make-placeholder 'result
                     (final-value result (%car bindings))))))
          (set! bindings (cdr bindings))
          result))))

(define (emit value)
  (let ((binding (find-binding value bindings)))
    (%if binding binding
        (let ((binding (make-placeholder (next-placeholder-name) value)))
          (set-car! bindings (cons binding (%car bindings)))
          binding))))

(define (find-binding value bindings)
  (%if (null? bindings) nil
       (let ((binding (find value (%car bindings)
                            :test #'equal :key #'placeholder-value)))
         (%if (not (null? binding)) binding
              (find-binding value (cdr bindings))))))

(define (final-value p bindings)
  (let ((result (placeholder-name p)))
    (when (%eq? p (%car bindings))
      (set! result (placeholder-value p))
      (set! bindings (cdr bindings)))
    (%if bindings
         `(let* ,(mapcar (lambda (b)
                           (list (placeholder-name b)
                                 (substitute-placeholders
                                  (placeholder-value b))))
                         (reverse bindings))
            ,(substitute-placeholders result))
         (substitute-placeholders result))))
```

`Emit` takes a value (piece of code) and returns a placeholder representing that value. It searches the binding stack for an existing binding of that value so that previous computation can be reused. If an existing binding can't be found, one is created using `make-placeholder`. `Next-placeholder-name` creates a new name.
`Final-value` constructs the `let*` expression representing the computation and returns it. This version does some simplifications. `Substitute-placeholders` takes a list structure and replaces each placeholder with either its name if it is in the binding environment,S or its value otherwise.

`Partial-eval` is the primary user interface. Given a thunk, it returns a piece of code. Here's how it works on the `nortward` example:

```
(define (partial-eval thunk)
  (set! placeholder-count 0)
  (let ((result (multiple-value-list (peval thunk))))
    (%if (cdr result) (values-list result)
         (placeholder-value (car result)))))

? (partial-eval
   (lambda ()  (northward
                (vmake (make-placeholder 'x)
                       (make-placeholder 'y)
                       (make-placeholder 'z)))))

(LET* ((PH0 (- 0.0 Y))
       (PH1 (* PH0 PH0))
       (PH2 (* X X))
       (PH3 (+ PH1 PH2)))
  (SQRT PH3))
```

`if` is handled using the equivalence between `(if <condition> <then> <else>)` and

```
(let ((a (lambda () <condition>))
      (b (lambda () <then>))
      (c (lambda () <else>)))
  (%if (funcall a) (funcall b) (funcall c)))
```

However, we use `peval` rather than `funcall,` and `emit` code for both `if` branches if the condition returns a placeholder:

```
(defconstant null-else (lambda () nil))

(defmacro if (condition then &optional else)
  `(if-runtime (lambda () ,condition)
               (lambda () ,then)
               ,(%if else `(lambda () ,else) 'null-else)))

(define (if-runtime condition then else)
  (let ((c (peval condition)))
    (%if (placeholder? c)
         (emit `(if ,c ,(peval then) ,(peval else)))
         (%if c (peval then) (peval else)))))
```

Exhibit 5 shows a serious example (simplified to save space). This is the essence of `distance-and-azimuth`. The azimuth calculation reuses pieces of the distance calculation so it is provided almost for free. When this software is run in the partial evaluator, where every operation is a generic function, it is twice as fast as the original. When compiled into the `common-lisp-user` package, it is thirty times faster.

To produce production quality software, we need to add type declarations, which are all the same here, `single-float`. A more general strategy is to add type information to the placeholder so that it can be propagated throughout the computation [BW]. It is not much harder to produce efficient software in one or more languages. This allows the library to be delivered in several languages while maintaining one set of high level sources.

## Other Applications

While the domain of this example may seem relatively narrow, Berlin and Weise [BW] applied this technique to several numerical applications. The applications were written in Scheme and their partial evaluator generated C software which was incorporated into the Scheme application. Speedups between a factor of 4 and 90 were reported.

The following subsections outline several applications where partial evaluation is appropriate. See [J, Ch. 13]. Partial evalution should be useful when used as a component of an object oriented library or framework either 1) by the library builder to provide an efficient library implementation, 2) by the library user to build an efficient application, or 3) by a language developer to allow programmers disciplined access to the compilation of the underlying language.

```
;;; Exhibit 5 The essense of distance-and-azimuth.
? (partial-eval
   (lambda ()
     (distance-and-azimuth earth (make-placeholder 'lat1)
                                 (make-placeholder 'lon1)
                                 (make-placeholder 'lat2)
                                 (make-placeholder 'lon2))))

(LET* ((PH0 (* 0.017453292519943295 LON1))
       (PH4 (ATAN (* 0.9933056200098587
                     (TAN (* 0.017453292519943295 LAT1)))
                  1.0))
       (PH5 (COS PH4))
       (PH7 (* PH5 (COS PH0)))
       (PH9 (* PH5 (SIN PH0)))
       (PH10 (SIN PH4))
       (PH11 (* 0.017453292519943295 LON2))
       (PH15 (ATAN (* 0.9933056200098587 (TAN (* 0.017453292519943295 LAT2)))
                   1.0))
       (PH16 (COS PH15))
       (PH18 (* PH16 (COS PH11)))
       (PH20 (* PH16 (SIN PH11)))
       (PH21 (SIN PH15))
       (PH24 (- (* PH10 PH20) (* PH21 PH9)))
       (PH27 (- (* PH21 PH7) (* PH10 PH18)))
       (PH30 (- (* PH18 PH9) (* PH20 PH7)))
       (PH42 (- 0.0 PH9)))
  (VALUES (* 6378.137
             (ATAN (SQRT (+ (* PH24 PH24) (* PH27 PH27) (* PH30 PH30)))
                   (+ (* PH18 PH7) (* PH20 PH9) (* PH10 PH21))))
          (IF (< (SQRT (+ (* PH42 PH42) (* PH7 PH7)))
                 1.1107651257113995E-16)
            180.0
            (LET* ((PH53 (* 57.29577951308232
                            (ATAN (- 0.0 PH30)(+ (* PH24 PH42) (* PH27 PH7))))))
              (IF (> PH53 0.0) PH53 (+ 360.0 PH53))))))
```

## OA1: Affine transformations in a graphics library

A graphics library is an essential part of an application that provides a graphical user interace. Such a library provides coordinate transformation primitives in various forms. Computer graphics texts use a redundant representation, referred to as "homogenous coordinates" where points in N space are represented as vectors in N+1 dimensional space. So, for example, the two dimensional point (x,y) would be represented by the vector `(vmake x y 1)`. (Neural networks also use such a representation for a neuron's weight vector.) A coordinate transformation can then be represented by a matrix [G] [NS]. Transforming a point becomes a matrix-vector multiplication and the composition of two transformations becomes matrix-matrix multiplication. Rather than always performing a complete 3x3 matrix multiplication, specializing the transformation matricies into classes such as `identity`, `translation`, `scaling`, and `rotation` can be more efficient. Composition of these classes leads to more complex classes, such as `scaling-translation`, `scaling-rotation`, `rotation-translation` and `scaling-rotation-translation`. These classes are often used in graphics libraries.

While using these eight subclasses of transformation is more efficient than using the general transformation, `scaling-rotation-translation`, alone, there are additional important special cases, such as reflection about the y-axis (for displaying text down a page) or rotation by 90 degrees (for displaying the y-axis of a graph), that are not as efficient as possible (because they contain unnecessary multiplication by 0, 1, or -1). Adding reflections about x and y axes, and the 90 degree rotations as special transformations bring the set up to 22 transformation classes. Optimized code for transforming and composing each case would be difficult by hand, but is easy using partial evaluation, because the same matrix multiplication routine can be used in each branch of the case statement. Since the same theory underlies 2-D, 3-D, and perspective transformations, one could imagine an efficient compact library based on partial evaluation that provides both for the price of one.

## OA2: Map Projections

A map projection library is another potential application of partial evaluation (suggested by Robert Fleishman, rmf@bbn.com). A map projection is a nonlinear transformation, but it is convenient to think of it as an object much like a window in a graphical user interface. This is because a projection:

- falls naturally into classes, such as "cylindrical", "equal area", or "orthographic"
- must provide operations such as transforming a point, deciding if a point is visible (clipping), or drawing a line (a line going off one edge of the map could appear again on another)
- must take advantage of special cases, like which great circles are straight lines.

Partial evaluation is particularly valuable here because there are many special cases. For example, a Mercator projection maps points on the sphere onto a cylinder. Great circles that are parallel or perpendicular to the axis of the cylinder are straight lines. The special case, where the cylinder is aligned with the north pole can be computed more efficiently than the general case known as "transverse Mercator".

It is valuable to make partial evaluation available to the library's users. This allows one to compose several transformations and produce efficient inlined software. This is important in applications were many points must be plotted quickly. For example, a reasonable looking outline map of the earth on today's 1,000 x 1,000 pixel displays requires about 100,000 points.

## OA3: Turning an Interpreter into a Compiler

Sometimes a program needs to be adjusted so that partial evaluation can be used to full advantage. For example, `(define (length=1 L) (= (length L) 1))` cannot be optimized if `L` is a placeholder. However, this version can:

```
(define (length=1 L) (same-length '(x) L))
(define (same-length L1 L2)
  (if (pair? L1) (and (pair? L2) (same-length (cdr L1) (cdr L2)))
      (not (pair? L1))))
```

This works because `L1` is known and controls the unrolling of the recursion. If we had reversed the arguments to `same-length` our partial evaluator would go into an infinite loop, while a fancier one might not.

Taking this idea one step further, consider a simple Scheme-like interpreter ([NO] p. 758) in Exhibit 6a. As is typical of interpeters (such as string or pattern matchers), there is a case statement the arms of which may have recursive calls to the interpreter. The important thing is that when such an interpreter is partially evaluated against a fixed program, `P`, each recursive call is replaced by the appropriate branch of the case statement, effectively compiling the program. This trick of organizing the case statement of an interpreter so that a partial evaluator can produce compiled code from it is so important, it is called "The Trick"[J].

Going one step further, if `P` was `interp` itself, and `interp` was powerful enough to interpret itself, then specializing `interp` with respect to `interp` would produce a compiler directly from the interpreter. In fact, powerful partial evaluators are interpreters that can interpret themselves.

While this is beyond our simple partial evaluator, The Trick is worth using even if you have to do it by hand. For example, Exhibit 6b shows the result of factoring interp into two pieces, `generate-code` that generates code for a program, `P`, and `execute` which executes that code in an environment, `env`. Here, it is convenient to represent code as closures and have `execute` be `funcall`, but there are other choices [FL].

## OA4: Opening the Language Implementation

Not only is Common Lisp an object oriented language, but its implementation is also object oriented in a way that allows a program access to the implementation objects (metaobjects). An application is written in terms of classes, generic functions, and methods. These metaobjects are also objects that can be inspected and manipulated by the programmer. The specification of the programmatic access to the metalayer is called a metaobject protocol (MOP). By programming in the metalayer, the developer can extend the behavior of the underlying language.

The current Common Lisp MOP provides reasonable runtime support but compile time support is limited and implementation specific. Thus while it is possible to extend the language in powerful ways, a portable extension may not be as efficient as the behavior provided by the underlying implementation.

```
;;; Exhibit 6a - A simple Scheme-like interpreter.
(define (interp P env)
  (cond
   ((symbol? P) (get-var P env))
   ((atom? P) P)
   ((case (car P)
      ((quote) (cadr P))
      ((begin) (last1 (mapcar (lambda (y) (interp y env)) (cdr P))))
      ((set!)  (set-var! (cadr P) (interp (caddr P) env) env))
      ...))))

;;; Exhibit 6b - Turning interp into a simple compiler, by hand.
(define (interp P env) (execute (generate-code P) env))
(define execute #'funcall)

(define (generate-code P)
  (cond
   ((symbol? P) (lambda (env) (get-var P env)))
   ((atom? P) (lambda (env) P))
   ((case (car P)
      ((quote) (let ((v (cadr P))) (lambda (env) v)))
      ((begin) (let ((items (mapcar #'generate-code (cdr P))))
                 (lambda (env)
                   (last1 (mapcar (lambda (y) (execute y env)) items)))))
      ((set!) (let ((var (cadr P))
                    (val (generate-code (caddr P))))
                (lambda (env) (set-var! var (execute val env)))))
      ...))))
```

An example metaprotocol is the `slot-value` protocol used to describe one's access to the value of a slot (instance variable, or data member) in an object. Such a protocol might be (this is slightly different than the one used in CLOS):

```
(define (slot-value object slot-name)
  (let* ((class (c-class-of object))
         (key (c-slot-key-using-class class object slot-name)))
    (if key
      (let ((bound? (c-slot-bound-using-class? class object key)))
        (if bound? (c-slot-access-using-class class object key)
            (slot-unbound-error object slot-name)))
      (slot-not-found-error object slot-name))))
```

Each of the `c-...-value-using` class functions is a compile time generic function that lets the class of the class of an object (the object's "metaclass") get involved in how one of its slots are accessed. Each could simply compile into a call to the equivalent runtime generic function. If `slot-value` was actually implemented that way, it would be extremely slow. However, the compiler steps in to provide an optimized version of `slot-value` when the metaclass of the object is known to be `standard-class`, for example. The idea behind a compile time MOP is to allow the programmer to step in and also provide code for metaclasses he defines as well.

We sketch how making `slot-value` a partially evaluated function can provide a disciplined way for this to happen. Suppose a user defines `light-weight-class,` a metaclass that does no runtime slot-bound check and can compute the position of a slot at compile time if the name of the slot is known. He could then use that metaclass to define a class and a method:

```
(defclass point ()
  ((x :initform 0 :initarg x)
   (y :initform 0 :initarg y))
  :metaclass light-weight-class))

(define (get-x (object point)) (slot-value object 'x))
```

When the method is compiled, the object system would invoke the partial evalutator to produce a macroexpansion of the `slot-value` form:

```
(peval (lambda () (slot-value (make-placeholder 'x 'class-of 'point) 'x)))
```

The following methods on l`ight-weight-class` would allow `slot-value` to be partially evaluatated into a primitive `%object-ref`:

```
(define (c-slot-bound-using-class? (class light-weight-class)) t)

(define (c-slot-key-using-class (class light-weight-class) object slot-name)
  (if (symbol? slot-name) (class-find-slot-key class slot-name)
      (emit `(class-find-slot-key ,class ,slot-name))))

(define (c-slot-access-using-class (class light-weight-class) object key)
  (emit `(%object-ref ,object ,(key-position key)))
```

## Conclusion

The partial evaluator developed in this article is simple compared to the state of the art [J]. It does not handle much of Common Lisp, and cannot partically evaluate itself, as fancier ones can [BO]. It is "on-line" which means it does its work in one pass. While this allows it to unroll loops and recursions, it could go into an infinite loop. It also assumes that functions have no side effects. However, this simplicity provides simple power. It is understandable - only the math and comparison functions and if were modified to be aware of the partial evaluator. It works with compiled code while other partial evaluators interpret the code in tree structured form.

Even such a minimal partial evaluator can be a useful component in an object oriented library. As it allows a user to write abstract code that is efficiently implemented, it is more like a language extention than a library in the usual sense. In Lisp, such language extensions are normally provided by the macro facility. A partial evaluator strengthens this facility by providing constant folding and redundant expression elimination (another way to put `defmacro` on steroids [BA]).

The power of a compiler does not come from knowing how to produce code. It comes from knowing when not to. Partial evaluation helps make this power accessable to the programmer.

## References


[BA] Baker H.G., Pragmatic Parsing in Common Lisp, ACM LISP Pointers, IV, 2, (April-June 1991), p. 3-15.

[BO] Bondorf, Andrers, Similix 5.0, ftp://ftp.diku.dk/pub/diku/dists/Similix.tar.z

[BU] Bullen, K.E., 1965, An Introduction to the Theory of Seismology, Cambridge Univ. Press, London, p. 154-155.

[BW] Berlin, Andrew, and Weise, Daniel, December 1990, Compiling scientific code using partial evaluation, IEEE Computer, p. 25-37.

[FL] Feeley, Marc and Lapalme, Guy, Using closures for code generation, Comput. Lang. Vol. 12, No. 1, pp. 47-66, 1987.

[HFJH] Hager, J.W. Fry, L.L, Jacks, S.S, and Hill D.R., The Defense Mapping Agency, Datums, Ellipsoids, Grids, and Grid Reference Systems, DMA TM 8358.1, Defense Mapping Agency, 1992, p. 2-4.

[GI] Giloi, Wolfgang K., Interactive Computer Graphics, Prentice-Hall, Inc. Englewood Cliffs, NJ, 1978, p. 92-111.

[GR] Graham, Paul, On Lisp Advanced Techniques for Common Lisp, Printice Hall, Englewood Cliffs, NJ, 1994.

[J] Jones, N.D., Gomard, C.K., Sostoft, Peter, Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993.

[KMW] Kiczales, Gregor, Theimer, Marving, and Welsch, Brent, A New Model Of Abstraction For Operating System Design. in Proceedings Of International Workshop On Object Orientation And Operating Systems, Pages 346--350, 1992.

[KKZG] Kleinrubatscher, Paul, Kriegshaber, Albert, Zochling, Robert, and Gluck Robert, Fortran program specialization, ACM SIGPLAN Notices, 30, 4, April 1995, p. 61-70.

[NO] Norvig, Peter, 1992, Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp, Morgan Kaufmann Publishers, San Mateo, CA, p. 267.

[NS] Newman, W.M. and Sproull, R.F., Principles of Interactive Computer Graphics, McGraw-Hill Book Co., New York, 1979, p. 53-62.

[VM] van Melle, Bill, Implementation of an iterator macro, ACM Lisp Pointers, III-2, p. 29-40.