Pearls of Theory: Intensional and Extensional Aspects of Partial Evaluation

Olivier Danvy
Computer Science Department
Aarhus University [*]
(danvy@daimi.aau.dk)

March 14 & 17, 1995

This pearl of theory is dedicated to partial evaluation: a program-specialization technique. Extensionally, partial evaluation paraphrases Kleene's S_n^m-theorem. It comes in two flavors — online and offline. Section 2 expresses both with commuting diagrams. Intensionally, a partial evaluator unfold calls, propagates constant values, and folds constant expressions. This is the topic of Section 3. As a warmup, we briefly consider formatting and pattern matching. We then address a continuation-passing interpreter for the λ-calculus. Following tradition, some oysters are also proposed.[1]

[*]Ny Munkegade, DK-8000 Aarhus C, Denmark. Supported by the DART project (Design, Analysis and Reasoning about Tools) of the Danish Research Councils. WWW: `http://www.daimi.aau.dk/~danvy`

[1]Had this been a pearl about Unix, we probably would have concentrated on the shells instead. Sic transit.

Contents

1 Introduction **3**

2 Extensional aspects of partial evaluation **3**

2.1 Definitional condition . . . 3

2.2 Binding times . . . 4

2.3 Offline partial evaluation . . . 4

2.4 Self-applicable partial evaluation . . . 5

2.5 Self-applicable offline partial evaluation . . . 5

3 Intensional aspects of partial evaluation **6**

3.1 Pattern matching . . . 6

3.1.1 Control-flow aspects of pattern matching . . . 7

3.1.2 Data-flow aspects of pattern matching . . . 10

3.2 A CPS interpreter for the λ-calculus . . . 11

4 Problems **15**

4.1 Oyster of the week . . . 15

4.2 Delicious oyster of the week . . . 15

4.3 Spicy oyster of the week . . . 15

4.3.1 Scheme . . . 16

4.3.2 Standard ML . . . 16

A Interpreter paraphernalia **16**

List of Figures

1 Test for flat-list equality . . . 8

2 Test for binary-tree equality . . . 8

3 Optimized test for binary-tree equality in CPS . . . 9

4 Equality test + flatten in CPS . . . 9

5 CPS interpreter . . . 12

6 A specializer for Figure 5 . . . 13

1 Introduction

Partial Evaluation is a program transformation technique for specializing a program with respect to some known part of its input. Given the remaining part of the input, the resulting specialized program will yield the desired result [1, 5].

Since Neil Jones's Mix project in the mid-eighties at DIKU, we have assisted to a veritable Renaissance of partial evaluation. Partial evaluation has successfully been used in areas such as compiler generation, scientific computing, computer graphics, and pattern matching. These applications have in common one fact: they can be shown to be particular cases of more general programs.

Section 2 reviews some extensional aspects of partial evaluation: *what* is it? Section 3 reviews some intensional aspects of partial evaluation: *how* does it work? Section 4 proposes some oysters.

These notes were assembled from bits and pieces of "Continuation-based partial evaluation", by Julia L. Lawall and Olivier Danvy [7] for Section 2, and of "Partial Evaluation in Procedural Languages", by Charles Consel and Olivier Danvy [2] for Section 3.1. Sections 3.2 and 4 are fresh.

NB: about Exercises 1 to 5, either pencil-and-paper solutions or keyboard-and-screen solutions[2] (along the lines of the enclosed note "More about Formatting" [3]) are acceptable.

2 Extensional aspects of partial evaluation

2.1 Definitional condition

Given a source program p, let us (arbitrarily) split its input into a pair $\langle \mathsf{s}, \mathsf{d}\rangle$. Partial evaluation of p with respect to the static input $\langle \mathsf{s}, \_\rangle$ yields the specialized program $\mathsf{p}^{\langle \mathsf{s}, \_\rangle}$, under a definitional condition that paraphrases Kleene's S_n^m-theorem [6]: applying this specialized program to the remaining input $\langle \_, \mathsf{d}\rangle$ should yield the same result as applying p to the complete input $\langle \mathsf{s}, \mathsf{d}\rangle$ — provided that the source program p, the partial evaluator PE, and the residual program $\mathsf{p}^{\langle \mathsf{s}, \_\rangle}$ all terminate.

This definitional condition is captured in the following diagram:

$$\begin{tikzcd}
\langle\langle \mathsf{p}, \langle \mathsf{s}, \_\rangle\rangle, \langle \_, \mathsf{d}\rangle\rangle \arrow[r, "\langle \text{eval } \mathsf{PE}, \text{id}\rangle"] \arrow[d, shift left, "\text{merge}"] & \langle \mathsf{p}^{\langle \mathsf{s}, \_\rangle}, \langle \_, \mathsf{d}\rangle\rangle \arrow[d, "\text{apply}"] \\
\langle \mathsf{p}, \langle \mathsf{s}, \mathsf{d}\rangle\rangle \arrow[u, shift left, "\text{split}"] \arrow[r, "\text{apply}"'] & r
\end{tikzcd}$$

[2] Usually known as "programs".

where eval is a curried version of apply.

$$\begin{aligned} \text{eval} &: \text{Program} \to \text{Input} \to \text{Output} \\ \text{apply} &: \text{Program} \times \text{Input} \to \text{Output} \end{aligned}$$

These two functions make it possible to distinguish between a function and the program implementing the function. (eval p) denotes the function implemented by the program p. (apply p i) applies the program p to the input i.

2.2 Binding times

The notion of binding time arises naturally in partial evaluation since source programs are evaluated in two stages: at partial-evaluation time (*i.e.*, in the top arrow of the diagram) and at run time (*i.e.*, in the right arrow of the diagram). Both values and expressions have binding times. A value that is available to the partial evaluator has binding time "static". A value that is not available until run time has binding time "dynamic". An expression that can be reduced at partial evaluation time, given the available static values, is referred to as a "static expression". An expression that must be evaluated at run time, because it depends on dynamic values, is referred to as a "dynamic expression". A partial evaluator reduces the static expressions and reconstructs the dynamic expressions to produce a residual program.

2.3 Offline partial evaluation

In practice, distinguishing between static and dynamic values incurs a definite interpretive overhead in a partial evaluator. This overhead is limited by analyzing the binding times of a source program prior to its actual specialization. This strategy is known as *offline* partial evaluation [1, 5]. An offline partial evaluator is staged into a binding-time analysis $\mathcal{A}$ and a specializer Spec (implementing a function $\mathcal{S}$):

$$\begin{aligned} \text{eval } \mathsf{PE} &= (\text{eval } \mathsf{Spec}) \circ \mathcal{A} \\ &= \mathcal{S} \circ \mathcal{A} \end{aligned}$$

The binding-time analysis identifies whether each term is static or dynamic. The specializer processes each term following these annotations. The situation is summarized in the following diagram.

$$\begin{tikzcd} \langle\langle \mathcal{A}\,\mathsf{p},\ \langle \mathsf{s},\ \_\rangle\rangle,\ \langle \_,\ \mathsf{d}\rangle\rangle \arrow[r, "\langle \text{eval } \mathsf{Spec},\ \text{id}\rangle"] \arrow[d, shift left, "\text{merge}"] & \langle \mathsf{p}^{\langle \mathsf{s},\ \_\rangle},\ \langle \_,\ \mathsf{d}\rangle\rangle \arrow[d, "\text{apply}"] \\ \langle \mathsf{p},\ \langle \mathsf{s},\ \mathsf{d}\rangle\rangle \arrow[u, shift left, "\text{split}"] \arrow[r, "\text{apply}"'] & r \end{tikzcd}$$

2.4 Self-applicable partial evaluation

Sometimes, one needs to specialize the same source program p several times, *e.g.*, with respect to different input values. In other words, one needs to apply PE several times to an input $\langle \mathsf{p},\ \_\rangle$ where p is fixed. This repeated specialization can be made more efficient by first specializing PE with respect to p, as captured in the following diagram.

$$\begin{tikzcd}
\langle\langle \mathsf{PE},\ \langle \mathsf{p},\ \_\rangle\rangle,\ \langle \_,\ \langle \mathsf{s},\ \_\rangle\rangle\rangle \arrow[r, "\langle \text{eval } \mathsf{PE},\ \text{id}\rangle"] \arrow[d, shift left, "\text{merge}"] & \langle \mathsf{PE}^{\langle \mathsf{p},\ \_\rangle},\ \langle \_,\ \langle \mathsf{s},\ \_\rangle\rangle\rangle \arrow[d, "\text{apply}"] \\
\langle \mathsf{PE},\ \langle \mathsf{p},\ \langle \mathsf{s},\ \_\rangle\rangle\rangle \arrow[r, "\text{apply}"'] \arrow[u, shift left, "\text{split}"] & \mathsf{p}^{\langle \mathsf{s},\ \_\rangle}
\end{tikzcd}$$

This diagram is a particular instance of the diagram of Section 2.1, where the source program is PE and the static input is $\langle \mathsf{p},\ \_\rangle$.

Furthermore, one may need to specialize PE several times, *e.g.*, with respect to different source programs. In other words, one may need to apply PE several times to an input $\langle \mathsf{PE},\ \_\rangle$. This repeated specialization can be made more efficient by first specializing PE with respect to PE, as captured in the following diagram. For clarity, we index the occurrences of PE.

$$\begin{tikzcd}
\langle\langle \mathsf{PE}_1,\ \langle \mathsf{PE}_2,\ \_\rangle\rangle,\ \langle \_,\ \langle \mathsf{p},\ \_\rangle\rangle\rangle \arrow[r, "\langle \text{eval } \mathsf{PE}_0,\ \text{id}\rangle"] \arrow[d, shift left, "\text{merge}"] & \langle \mathsf{PE}_1^{\langle \mathsf{PE}_2,\ \_\rangle},\ \langle \_,\ \langle \mathsf{p},\ \_\rangle\rangle\rangle \arrow[d, "\text{apply}"] \\
\langle \mathsf{PE}_1,\ \langle \mathsf{PE}_2,\ \langle \mathsf{p},\ \_\rangle\rangle\rangle \arrow[r, "\text{apply}"'] \arrow[u, shift left, "\text{split}"] & \mathsf{PE}_2^{\langle \mathsf{p},\ \_\rangle}
\end{tikzcd}$$

Again this diagram is a particular instance of the diagram at the beginning of Section 2 Here the source program is PE_1 and the static input is $\langle \mathsf{PE}_2,\ \_\rangle$.

Since $\mathsf{PE}_1^{\langle \mathsf{PE}_2,\ \_\rangle}$ transforms a program into a partial evaluator dedicated to this program, it acts as a partial-evaluation compiler, "pecom" [1]. In the particular case where the source program is an interpreter (see our delicious oyster of the week in Section 4.2), the diagrams above specialize into the "Futamura projections" [4, 5].

2.5 Self-applicable offline partial evaluation

In the context of offline partial evaluation, it is the specializer, not the partial evaluator, that is self-applied. Thus the diagrams above need to be adjusted. The first diagram is rewritten as follows:

$$\begin{tikzcd}
\langle\langle \mathcal{A}\,\mathsf{Spec},\ \langle \mathcal{A}\,\mathsf{p},\ \_\rangle\rangle,\ \langle \_,\ \langle \mathsf{s},\ \_\rangle\rangle\rangle \arrow[r, "\langle \text{eval } \mathsf{Spec},\ \text{id}\rangle"] \arrow[d, shift left, "\text{merge}"] & \langle \mathsf{Spec}^{\langle \mathcal{A}\,\mathsf{p},\ \_\rangle},\ \langle \_,\ \langle \mathsf{s},\ \_\rangle\rangle\rangle \arrow[d, "\text{apply}"] \\
\langle \mathsf{Spec},\ \langle \mathcal{A}\,\mathsf{p},\ \langle \mathsf{s},\ \_\rangle\rangle\rangle \arrow[u, shift left, "\text{split}"] \arrow[r, "\text{apply}"'] & \mathsf{p}^{\langle \mathsf{s},\ \_\rangle}
\end{tikzcd}$$

This diagram is a particular instance of the diagram in Section 2.3, where the source program is Spec and the static input is $\langle \mathcal{A}\,\mathsf{p},\ \_\rangle$. In particular, the binding time of the input to p is fixed and thus the resulting dedicated specializer $\mathsf{Spec}^{\langle \mathcal{A}\,\mathsf{p},\ \_\rangle}$ expects an input with a fixed binding-time format.

Similarly, the second diagram is rewritten as follows:

$$\begin{tikzcd}
\langle\langle \mathcal{A}\,\mathsf{Spec}_1,\ \langle \mathcal{A}\,\mathsf{Spec}_2,\ \_\rangle\rangle,\ \langle \_,\ \langle \mathcal{A}\,\mathsf{p},\ \_\rangle\rangle\rangle \arrow[r, "\langle \text{eval } \mathsf{Spec}_0,\ \text{id}\rangle"] \arrow[d, shift left, "\text{merge}"] & \langle \mathsf{Spec}_1^{\langle \mathcal{A}\,\mathsf{Spec}_2,\ \_\rangle},\ \langle \_,\ \langle \mathcal{A}\,\mathsf{p},\ \_\rangle\rangle\rangle \arrow[d, "\text{apply}"] \\
\langle \mathsf{Spec}_1,\ \langle \mathcal{A}\,\mathsf{Spec}_2,\ \langle \mathcal{A}\,\mathsf{p},\ \_\rangle\rangle\rangle \arrow[u, shift left, "\text{split}"] \arrow[r, "\text{apply}"'] & \mathsf{Spec}_2^{\langle \mathsf{p},\ \_\rangle}
\end{tikzcd}$$

Again, this diagram is a specialized version of the previous diagram, where the static input is $\langle \mathcal{A}\,\mathsf{Spec}_2,\ \_\rangle$. In particular, the binding-time of the input to Spec_2 is fixed.

3 Intensional aspects of partial evaluation

3.1 Pattern matching

Pattern matching determines whether a datum belongs to a set of data. The set of data is finitely described with a pattern. Given a pattern and a datum, a pattern-matching procedure computes whether the pattern matches the datum, *i.e.*, whether the datum belongs to the set specified by the pattern. To eliminate their interpretive overhead, patterns are usually compiled. Partial evaluation offers a natural way to compile patterns and to generate pattern compilers.

Pattern matching has been abundantly and successfully described in the literature. Not only is it a nice application of partial evaluation, but partial evaluation also offers a generic framework for compiling patterns.

Let us address the control-flow and the data-flow aspects of pattern matching.

3.1.1 Control-flow aspects of pattern matching

A pattern matcher basically performs a recursive descent on both the pattern and the datum. This situation is ideal for a partial evaluator: since patterns are static and finite, all recursive calls can be statically unfolded. The resulting specialized programs are structured just like the static data.

Let us take an example: equality between two lists of numbers. It is a very simple instance of pattern-matching: the data are lists of numbers and the patterns are singleton sets of one list, represented as that list. Figure 1 displays the source program.[3]

Exercise 1: Specialize this source program with respect to the static list

```
(IntCons 7 (IntCons 2 (IntNil)))
```

in a way that satisfies the following definitional equation where `main.1` is the name of the specialized program.

```
(main '(IntCons 7 (IntCons 2 (IntNil))) d)
== (main.1 d), for any d.
```

What do you observe about the shape of this residual program? How does it compare with the static list? □

In addition to performing a recursive descent, pattern matchers also maintain information — typically about failure and success. In case of failure, a pattern matcher routinely stops or backtracks. Both actions can be achieved using control operators such as `call/cc` in Scheme. Alternatively, pattern matchers are expressed as tail-recursive programs accumulating intermediate results and managing their control flow iteratively.

Let us consider pattern matching in binary trees. Figure 2 displays the source program.

Exercise 2: Specialize this source program with respect to the static tree

```
(IntNode (IntNode (IntLeaf 1) (IntLeaf 9))
         (IntNode (IntLeaf 7) (IntLeaf 2)))
```

in a way that satisfies the following definitional equation where `main.2` is the name of the specialized program.

```
(main '(IntNode (IntNode (IntLeaf 1) (IntLeaf 9))
                (IntNode (IntLeaf 7) (IntLeaf 2)))
      d)
== (main.2 d), for any d.
```

What do you observe about the shape of this residual program? How does it compare with the static tree? □

[3] `/users/danvy/Scheme/records.ss` contains definitions for `define-record` and `case-record` as Chez Scheme macros.

```
(define-record IntList
  (IntNil)
  (IntCons Int IntList))

(define main
  (lambda (p d)
    (match p d)))

(define match
  (lambda (p d)
    (case-record p
      [(IntNil) (case-record d
                  [(IntNil) #t]
                  [(IntCons - -) #f])]
      [(IntCons i p) (case-record d
                       [(IntNil) #f]
                       [(IntCons j d) (and (= i j)
                                           (match p d))])])))
```

Figure 1: Test for flat-list equality

```
(define-record IntTree
  (IntLeaf Int)
  (IntNode IntTree IntTree))

(define main
  (lambda (p d)
    (match p d)))

(define match
  (lambda (p d)
    (case-record p
      [(IntLeaf i) (case-record d
                     [(IntLeaf j) (= i j)]
                     [(IntNode - -) #f])]
      [(IntNode pl pr) (case-record d
                         [(IntLeaf -) #f]
                         [(IntNode dl dr)
                          (and (match pl dl)
                               (match pr dr))])])))
```

Figure 2: Test for binary-tree equality

```
(define main
  (lambda (p d)
    (match p d (lambda () #t))))

(define match
  (lambda (p d k)
    (case-record p
      [(IntLeaf i)
       (case-record d
         [(IntLeaf j) (if (= i j) (k) #f)]
         [(IntNode - -) #f])]
      [(IntNode pl pr)
       (case-record d
         [(IntLeaf -) #f]
         [(IntNode dl dr)
          (match pl dl (lambda ()
                         (match pr dr k)))])])))
```

Figure 3: Optimized test for binary-tree equality in CPS

```
(define-record Result
  (Failure)
  (Success IntList))

(define main
  (lambda (p d)
    (match p d '() (lambda (l)
                     (Success (reverse l))))))

(define match
  (lambda (p d l k)
    (case-record p
      [(IntLeaf i)
       (case-record d
         [(IntLeaf j) (if (= i j)
                          (k (cons i l))
                          (Failure))]
         [(IntNode - -) (Failure)])]
      [(IntNode pl pr)
       (case-record d
         [(IntLeaf -) (Failure)]
         [(IntNode dl dr)
          (match pl dl l (lambda (l)
                           (match pr dr l k)))])])))
```

Figure 4: Equality test + flatten in CPS

The source program is expressed in direct style and thus if matching fails, the value `#f` "bubbles up" with one test for each recursive call. A continuation-passing source program could avoid these tests very simply, by not applying the continuation at a mismatch point. Figure 3 displays a CPS (continuation-passing style) version of the source program, where at each node, the continuation is only applied if matching succeeds.

Exercise 3: Specialize this source program with respect to the same static tree as above. As usual, the following definitional equation needs to be satisfied, where `main.3` is the name of the specialized program.

```
(main '(IntNode (IntNode (IntLeaf 1) (IntLeaf 9))
                (IntNode (IntLeaf 7) (IntLeaf 2)))
      d)
== (main.3 d), for any d.
```

What do you observe about the shape of this residual program? What happens when a mismatch occurs? Why? □

3.1.2 Data-flow aspects of pattern matching

Partial evaluation usually works by propagating constant values forward. In the last example (Figure 3), it is the success / failure information that has been propagated forward: the continuation is only applied if matching has not failed yet. But the continuation itself is a zero-ary procedure, a thunk. It only delays the rest of the continuation until the next equality test succeeds.

More could be propagated. For example, Figure 4 displays a slight extension of Figure 3. In addition of testing the equality of two binary trees, this source program returns the list of leaves of the binary tree. Now the continuation is passed the list of leaves so far.

Exercise 4: Specialize this source program with respect to the same static tree as above. As usual, the following definitional equation needs to be satisfied, where `main.4` is the name of the specialized program.

```
(main '(IntNode (IntNode (IntLeaf 10) (IntLeaf 20))
                (IntLeaf 30))
      d)
== (main.4 d), for any d.
```

What do you observe about the shape of this residual program? What happens when a mismatch occurs? What happens when there is no mismatch? □

Exercise 5: The source program of Figure 4 builds the list of leaves out of the (static) pattern. We can make it build this list out of the (dynamic) data by changing the expression

```
(k (cons i l))
```

where `i` denotes a piece of pattern into the following expression, where `j` denotes a piece of data.

```
(k (cons j l))
```

How would the "success" branch of the specialized program read? Is this reasonable? □

3.2 A CPS interpreter for the λ-calculus

In this section, we play the same game with a CPS interpreter for the fully-parenthesized call-by-value lambda-calculus. Figure 5 shows the interpreter. Here is a small interactive session with it:

```
> ((meaning '2)
   (lambda (a) a))
2
> ((meaning '(lambda (x) x))
   (lambda (a) a))
#<procedure>
> ((meaning '((lambda (x) x) 10))
   (lambda (a) a))
10
> ((meaning '(add1 10))
   (lambda (a) a))
11
> ((meaning '(+ 10 20))
   (lambda (a) a))
30
> ((meaning '((lambda (x) (* x 100)) 10))
   (lambda (a) a))
1000
> ((meaning '(if #t 1 2))
   (lambda (a) a))
1
> ((meaning '(if #f 1 2))
   (lambda (a) a))
2
> ((meaning '(((lambda (fix)
                 (fix (lambda (fac)
                        (lambda (n)
                          (if (zero? n)
                              1
                              (* n (fac (- n 1))))))))
               (lambda (f)
                 ((lambda (g) (f (lambda (x) ((g g) x))))
                  (lambda (g) (f (lambda (x) ((g g) x)))))))
              5))
   (lambda (a) a))
120
>
```

Along the lines of the accompanying note "More about Formatting" [3], Figure 6 shows the corresponding generating extension. Here is the corresponding session:

```
(define meaning
  (lambda (e)
    (lambda (k)
      (letrec ([eval
                (lambda (e r k)
                  (cond
                    [(constant? e)
                     (k e)]
                    [(symbol? e)
                     (k (r e))]
                    [(lambda? e)
                     (k (lambda (actual k)
                          (eval (lambda->body e)
                                (extend (lambda->formal e) actual r)
                                (lambda (v) (k v)))))]
                    [(if? e)
                     (eval (if->test e)
                           r
                           (lambda (v)
                             (if v
                                 (eval (if->then e) r k)
                                 (eval (if->else e) r k))))]
                    [(prim1? e)
                     (eval (prim1->arg1 e)
                           r
                           (lambda (v1)
                             (k ((do1 (prim1->prim e)) v1))))]
                    [(prim2? e)
                     (eval (prim2->arg1 e)
                           r
                           (lambda (v1)
                             (eval (prim2->arg2 e)
                                   r
                                   (lambda (v2)
                                     (k ((do2 (prim2->prim e)) v1 v2))))))]
                    [(call? e)
                     (eval (call->fun e)
                           r
                           (lambda (v0)
                             (eval (call->arg e)
                                   r
                                   (lambda (v1)
                                     (v0 v1 (lambda (v) (k v)))))))]
                    [else
                     (error 'meaning "syntax error: ~s" e)]))])
        (eval e init-env (lambda (v) (k v)))))))
```

Figure 5: CPS interpreter

```
(define meaning-c
  (lambda (e)
    (let ([k (gensym! "k")])
      `(lambda (k)
         ,(letrec ([eval
                    (lambda (e r k)
                      (cond
                        [(constant? e)
                         (k e)]
                        [(symbol? e)
                         (k (r e))]
                        [(lambda? e)
                         (k (let ([actual (gensym! "a")]
                                  [k (gensym! "k")])
                              `(lambda (,actual ,k)
                                 ,(eval (lambda->body e)
                                        (extend (lambda->formal e) actual r)
                                        (lambda (v) `(k ,v))))))]
                        [(if? e)
                         (eval (if->test e)
                               r
                               (lambda (v)
                                 `(if ,v
                                      ,(eval (if->then e) r k)
                                      ,(eval (if->else e) r k))))]
                        [(prim1? e)
                         (eval (prim1->arg1 e)
                               r
                               (lambda (v1)
                                 (k `(,(prim1->prim e) ,v1))))]
                        [(prim2? e)
                         (eval (prim2->arg1 e)
                               r
                               (lambda (v1)
                                 (eval (prim2->arg2 e)
                                       r
                                       (lambda (v2)
                                         (k `(,(prim2->prim e) ,v1 ,v2))))))]
                        [(call? e)
                         (eval (call->fun e)
                               r
                               (lambda (v0)
                                 (eval (call->arg e)
                                       r
                                       (lambda (v1)
                                         (let ([v (gensym! "v")])
                                           `(,v0 ,v1 (lambda (,v) ,(k v))))))))]
                        [else
                         (error 'meaning "syntax error: ~s" e)]))])
            (eval e init-env (lambda (v) `(k ,v))))))))
```

Figure 6: A specializer for Figure 5

```
> (meaning-c '2)
(lambda (k0) (k0 2))
> (meaning-c '(lambda (x) x))
(lambda (k1) (k1 (lambda (a2 k3) (k3 a2))))
> (meaning-c '((lambda (x) x) 10))
(lambda (k4)
   ((lambda (a5 k6) (k6 a5)) 10 (lambda (v7) (k4 v7))))
> (meaning-c '(add1 10))
(lambda (k8) (k8 (add1 10)))
> (meaning-c '(+ 10 20))
(lambda (k9) (k9 (+ 10 20)))
> (meaning-c '((lambda (x) (* x 100)) 10))
(lambda (k10)
   ((lambda (a11 k12) (k12 (* a11 100))) 10 (lambda (v13) (k10 v13))))
> (meaning-c '(if #t 1 2))
(lambda (k14) (if #t (k14 1) (k14 2)))
> (meaning-c '(((lambda (fix)
                  (fix (lambda (fac)
                         (lambda (n)
                           (if (zero? n)
                               1
                               (* n (fac (- n 1))))))))
                (lambda (f)
                  ((lambda (g) (f (lambda (x) ((g g) x))))
                   (lambda (g) (f (lambda (x) ((g g) x)))))))
               5))
(lambda (k15)
   ((lambda (a16 k17)
       (a16 (lambda (a18 k19)
               (k19 (lambda (a20 k21)
                       (if (zero? a20)
                           (k21 1)
                           (a18 (- a20 1) (lambda (v22) (k21 (* a20 v22))))))))
            (lambda (v23) (k17 v23))))
    (lambda (a24 k25)
       ((lambda (a26 k27)
           (a24 (lambda (a28 k29)
                   (a26 a26 (lambda (v30) (v30 a28 (lambda (v31) (k29 v31))))))
                (lambda (v32) (k27 v32))))
        (lambda (a33 k34)
           (a24 (lambda (a35 k36)
                   (a33 a33 (lambda (v37) (v37 a35 (lambda (v38) (k36 v38))))))
                (lambda (v39) (k34 v39))))
        (lambda (v40) (k25 v40))))
    (lambda (v41) (v41 5 (lambda (v42) (k15 v42))))))
>
```

Exercise 6: Characterize the output of `meaning-c`.

Exercise 7: What happens is we apply the output of `meaning-c` to `(lambda (a) a)`? Why? Was it expectable?

4 Problems

4.1 Oyster of the week

Apply pecom to PE. What happens?

4.2 Delicious oyster of the week

Consider an programming-language interpreter

$$\text{Program} \times \text{Input} \rightarrow \text{Output}$$

and instantiate the diagrams of Section 2. What is the result of specializing the interpreter with respect to a program? What is the process of specializing the interpreter with respect to a program? How is the correctness criterion of partial evaluation instantiated? What is the result of specializing a partial evaluator with respect to an interpreter? What is the process of specializing a partial evaluator with respect to an interpreter? How is the correctness criterion of partial evaluation instantiated? What is the result of applying pecom to an interpreter? What is the process of applying pecom to an interpreter?

4.3 Spicy oyster of the week

"Currying" a function of type $\sigma_1 \times \sigma_2 \rightarrow \tau$ amounts to map it into a function of type $\sigma_1 \rightarrow \sigma_2 \rightarrow \tau$. This is achieved with a "curry" function (named after the logician Haskell Curry).[4]

Programming the curry function in Scheme is easy:

```
> (define curry
    (lambda (f)
      (lambda (x)
        (lambda (y)
          (f x y)))))
> (define adder (curry +))
> (define add1 (adder 1))
> (define add2 (adder 2))
> (add1 10)
11
> (add2 20)
22
>
```

What happens, however, in the presence of functions that take more than two arguments?

```
> (define curry3
    (lambda (f)
```

[4]The concept of currying, however, can be traced back to Manfred Schönfinkel, in 1924. More homework: what would be the name of this section if "schönfinkelization" was used in place of "currying"?

```
         (lambda (x)
           (lambda (y)
             (lambda (z)
               (f x y z))))))
> (define baz
    (lambda (x y z)
      (+ (* x y) (* x z))))
> (baz 10 2 3)
50
> (define bazer (curry3 baz))
> (((bazer 10) 2) 3)
50
>
```

The point of this exercise is to *generalize* curry (instead of to specialize it).

4.3.1 Scheme

Using Scheme, program a function `super-curry` taking two arguments: the arity of a function (an integer greater or equal to 2) and a function of that arity, and returning the corresponding super-curried function. (NB: you may or you may not use `reverse`.)

```
> (((super-curry 2 +) 1) 10)
11
> ((((super-curry 3 (lambda (x y z) (+ (* x y) (* x z)))) 10) 2) 3)
50
>
```

Could you extend `super-curry` to functions of arity 1? Of arity 0?[5] How would you specialize `super-curry` to get back `curry` or `curry3`, for example?

4.3.2 Standard ML

Transliterate `super-curry` in Standard ML. What do you observe? (If you are unfamiliar with Standard ML, transliterate `super-curry` in Michael Schwartzbach's tiny functional language [8] instead and exhibit its type.)

A Interpreter paraphernalia

```
(define constant?
  (lambda (e)
    (or (number? e) (boolean? e) (string? e))))

(define lambda?
  (lambda (e)
    (and (pair? e)
         (eq? (car e) 'lambda))))
```

[5]What *is* a function of arity 0?

```
(define lambda->formal caadr)
(define lambda->body caddr)

(define if?
  (lambda (e)
    (and (pair? e)
         (eq? (car e) 'if))))
(define if->test cadr)
(define if->then caddr)
(define if->else cadddr)

(define prim1?
  (lambda (e)
    (and (pair? e)
         (member (car e) '(car cdr null? add1 sub1 zero?)))))
(define prim1->prim car)
(define prim1->arg1 cadr)

(define prim2?
  (lambda (e)
    (and (pair? e)
         (member (car e) '(cons + - * /)))))
(define prim2->prim car)
(define prim2->arg1 cadr)
(define prim2->arg2 caddr)

(define call? pair?)
(define call->fun car)
(define call->arg cadr)

(define init-env
  (lambda (i)
    (error 'init-env "unbound variable: ~s" i)))

(define extend
  (lambda (i v r)
    (lambda (j)
      (if (equal? i j)
          v
          (r j)))))

(define do1
  (lambda (op)
    (case op
      [(car) car]
      [(cdr) cdr]
      [(null?) null?]
      [(add1) add1]
      [(sub1) sub1]
      [(zero?) zero?]
      [else (error 'do1 "unrecognized primop" op)])))
```

```
(define do2
  (lambda (op)
    (case op
      [(cons) cons]
      [(+) +]
      [(-) -]
      [(*) *]
      [(/) /]
      [else (error 'do2 "unrecognized primop" op)])))

(define gensym!
  (let ([gensym-count (vector -1)])
    (lambda (str)
      (begin
        (vector-set! gensym-count 0 (add1 (vector-ref gensym-count 0)))
        (string->symbol
          (string-append str
                         (number->string (vector-ref gensym-count 0))))))))
```

References

[1] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[2] Charles Consel and Olivier Danvy. *Partial Evaluation in Procedural Languages.* The MIT Press, 1995. To appear.

[3] Olivier Danvy. More about formatting. Lecture notes, December 1993.

[4] Yoshihito Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls 2, 5*, pages 45–50, 1971.

[5] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.

[6] Stephen C. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, New Jersey, 1952.

[7] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. Technical Report CS-95-178, Computer Science Department, Brandeis University, Waltham, Massachusetts, January 1995. An earlier version appeared in the proceedings of the 1994 ACM Conference on Lisp and Functional Programming.

[8] Michael Schwartzbach. Polymorphic type inference. Pearls of Theory PT-95-2, BRICS, Aarhus, Denmark, February 1995.