Partial Evaluation of the Euclidian Algorithm

(Extended Version)

Olivier Danvy
Mayer Goldberg

BRICS Report Series RS-97-1

ISSN 0909-0878 January 1997

Partial Evaluation of the Euclidian Algorithm [*] (extended version)

Olivier Danvy Mayer Goldberg

BRICS [†]

Department of Computer Science

Aarhus University [‡]

January 1997

Abstract

Some programs are easily amenable to partial evaluation because their control flow clearly depends on one of their parameters. Specializing such programs with respect to this parameter eliminates the associated interpretive overhead. Some other programs, however, do not exhibit this interpreter-like behavior. Each of them presents a challenge for partial evaluation. The Euclidian algorithm is one of them, and in this article, we make it amenable to partial evaluation.

We observe that the number of iterations in the Euclidian algorithm is bounded by a number that can be computed given either of the two arguments. We thus rephrase this algorithm using bounded recursion. The resulting program is better suited for automatic unfolding and thus for partial evaluation. Its specialization is efficient.

Keywords: partial evaluation, scientific computation.

The programs described in this article are available at the URL

`http://www.brics.dk/~danvy/Programs/{power,euclid}.scm`

[*]To appear in LISP and Symbolic Computation.

[†]Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

[‡]Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
Phone: (+45) 89 42 33 69. Fax: (+45) 89 42 32 55
E-mail: `{danvy,mayer}@brics.dk`
Home pages: `http://www.brics.dk/~{danvy,mayer}`

1 Partial Evaluation in Principle

Partial evaluation is a program-transformation technique for specializing programs [5, 9]. A partial evaluator typically specializes a source program with respect to parts of its input and yields a residual program. Running this residual program on the remaining input yields the same result as the source program on the complete input — but (it is hoped) more efficiently.

$$\begin{tikzcd}
\text{Source program} \arrow[r, "\substack{\text{partial evaluation}\\ \text{(some input)}}"] \arrow[dr, "\substack{\text{evaluation}\\ \text{(complete input)}}"'] & \text{Residual program} \arrow[d, "\substack{\text{evaluation}\\ \text{(remaining input)}}"] \\
 & \text{Result}
\end{tikzcd}$$

2 Partial Evaluation in Practice

Some programs lend themselves well to partial evaluation — for example, when the initial values of their induction variables are known. A partial evaluator can simply unfold the corresponding recursive calls.

2.1 An effective example

For example, Figure 1 displays the source procedure computing the power function. It is written using the programming language Scheme [4]. Figure 2 displays its associated generating extension [9] when the exponent is known — the generating extension of a program p is a (stand-alone) program carrying out the specialization of p. It is also known as a "backquote interpreter" in the literature [7, 11].

Figure 3 displays the result of specializing power with respect to the exponent 10 (i.e., of running the generating extension of Figure 2). In the base case, the residual code could be simplified a bit further, but this would only clutter Figure 2. The residual code is represented both in compact form and in linearized form, i.e., using straight-line code, which is an asset

```
(define sqr
  (lambda (x) (* x x)))

(define power
  (lambda (x n)
    (letrec ([loop (lambda (n)
                     (cond
                       [(zero? n)
                        1]
                       [(odd? n)
                        (* x (sqr (loop (/ (1- n) 2))))]
                       [else
                        (sqr (loop (/ n 2)))]))])
      (loop n))))
```

Figure 1: Source procedure computing the power function

```
(define power-gen
  (lambda (n)
    (let ([x (gensym! "x")])
      `(lambda (,x)
         ,(letrec ([loop (lambda (n)
                           (cond
                             [(zero? n)
                              `1]
                             [(odd? n)
                              `(* ,x (sqr ,(loop (/ (1- n) 2))))]
                             [else
                              `(sqr ,(loop (/ n 2)))]))])
            (loop n))))))
```

Figure 2: Generating extension for the power function

```
(define power-10
  (lambda (x8)
    (sqr (* x8 (sqr (sqr (* x8 (sqr 1))))))))

;;; for all x, (power x 10) = (power-10 x) = (power-10-linearized x)

(define power-10-linearized
  (lambda (x9)
    (let* ([x10 (sqr 1)]
           [x11 (* x9 x10)]
           [x12 (sqr x11)]
           [x13 (sqr x12)]
           [x14 (* x9 x13)]
           [x15 (sqr x14)])
      x15)))
```

Figure 3: Specialized versions of the power function

```
(define GCD
  (lambda (m n)
    (if (zero? m)
        n
        (GCD (remainder n m) m))))
```

Figure 4: Source procedure computing the GCD function (Euclidian algorithm)

for compiling scientific code efficiently [2, 3]. It is simple to modify Figure 2 to make it generate the let expressions, and again to simplify the residual code a bit in the base case (see appendix).

2.2 An ineffective example

Other programs, however, are less suited for partial evaluation because their control flow does not uniquely depend on one of their parameters. This is the case for the Euclidian algorithm, which computes the greatest common divisor of two numbers (see Procedure GCD in Figure 4). Both arguments act as induction variables, and furthermore, the induction is mixed. Therefore, GCD has no static induction variable and a partial evaluator, given any one of the two arguments, cannot specialize GCD as simply as in Figure 2.

2.3 Call unfolding and bounded recursion

As illustrated in Section 2.1, specializing the power procedure with respect to a given exponent amounts to unfolding its recursive calls. Indeed, the power function is defined inductively over the exponent, and correspondingly, the power procedure is defined recursively — which is actually too heavy handed. We can instead consider a bounded recursive definition of the power procedure and the corresponding generating extension.

As a first step, let us make the unbounded recursive definition explicit. We first exhibit the use of the (applicative-order) fixed-point operator in the definition of the power procedure (Figure 5), and then we factor the functional out through uncurrying and eta-reduction. The result, displayed in Figure 6, is a traditional recursive definition using a fixed-point operator: Procedure `power` is the fixed-point, i.e., the potentially unbounded composition of Functional `(make-power-functional x)`, for any `x`.

Since the exponent determines the number of recursive calls, we can simplify this unbounded recursive definition into a bounded recursive one. We do this by defining `power` as the m-th composition of `(make-power-functional x)`, for any `x`. The operator taking a function and returning its m-th composition reads as follows.

$$\lambda f.\lambda x.\underbrace{(f \cdots (f}_{m \text{ times}}\ x) \cdots)$$

This operator is the m-th Church numeral [1]. Figure 7 displays the Scheme procedure `integer->Church`, which takes an integer m, and returns the m-th Church numeral.

Now we only need the number of iterations of the exponentiation algorithm. Given an exponent n, the algorithm divides it by two until it hits zero. The number of iterations is thus the number of digits of the binary representation of n plus one, i.e., $\lfloor \log_2(n) \rfloor + 2$.

Figure 8 displays the definition of the exponentiation algorithm in bounded form. It behaves exactly as the code of Figures 1, 5, and 6, with the added guarantee that by construction, it cannot loop. The corresponding generating extension is shown in Figure 9. It behaves exactly as the code of Figure 2, also with the added guarantee that by construction, it cannot loop.

The problem with the Euclidian algorithm is that we cannot define it inductively over one of its parameters, and thus we cannot specialize the corresponding program merely using bounded unfolding.

```
(define unbounded-power
  (lambda (x n)
    ((fix1 (lambda (loop)
             (lambda (n)
               (cond
                 [(zero? n)
                  1]
                 [(odd? n)
                  (* x (sqr (loop (/ (1- n) 2))))]
                 [else
                  (sqr (loop (/ n 2)))])))) n)))
;;; where fix1 is an applicative-order fixed-point operator
;;; for unary functions.
```

Figure 5: Inner functional associated to the exponentiation algorithm

```
(define make-power-functional
  (lambda (x)
    (lambda (loop)
      (lambda (n)
        (cond
          [(zero? n)
           1]
          [(odd? n)
           (* x (sqr (loop (/ (1- n) 2))))]
          [else
           (sqr (loop (/ n 2)))])))))

(define unbounded-power
  (lambda (x n)
    ((fix1 (make-power-functional x)) n)))
;;; where fix1 is an applicative-order fixed-point operator
;;; for unary functions.
```

Figure 6: Outer functional associated to the exponentiation algorithm

```
(define integer->Church
  (lambda (m)
    (if (zero? m)
        (lambda (f)
          (lambda (x)
            x))
        (let ([m-1 (integer->Church (1- m))])
          (lambda (f)
            (lambda (x)
              (f ((m-1 f) x))))))))
```

Figure 7: Procedure generating a Church numeral

```
(define bottom
  (lambda (f)
    (lambda xs
      (error f "out of gas"))))

(define float->integer
  (lambda (x)
    (inexact->exact (truncate x))))

(define log2
  (lambda (n)
    (/ (log n) (log 2))))

(define bounded-power
  (lambda (x n)
    ((((integer->Church (+ (float->integer (log2 n)) 2))
        (make-power-functional x))
      (bottom 'power))
     n)))
```

Figure 8: The exponentiation algorithm in bounded form

```
(define bounded-power-gen
  (lambda (n)
    (let ([x (gensym! "x")])
      `(lambda (,x)
         ,((((integer->Church (+ (float->integer (log2 n)) 2))
             (lambda (loop)
               (lambda (n)
                 (cond
                   [(zero? n)
                    `1]
                   [(odd? n)
                    `(* ,x ,(loop (1- n)))]
                   [else
                    `(sqr ,(loop (/ n 2)))]))))
           (lambda (n)
             `((bottom 'power) ,n)))
          n)))))
```

Figure 9: Generating extension for the exponentiation algorithm in bounded form

3 Making the Euclidian Algorithm Amenable to Partial Evaluation

The number of recursive calls in `GCD`, however, is bounded. Given any one of `GCD`'s arguments, this bound can be computed independently of the other one. This property enables us to put the Euclidian algorithm into the partial-evaluation groove.

We first consider an unbounded recursive definition of `GCD` (Section 3.1). We then establish an upper bound for the Euclidian algorithm (Section 3.2). We then state the corresponding unbounded recursive definition of `GCD` and its associated generating extension (Section 3.3).

3.1 The unbounded recursive definition of the Euclidian algorithm

As a first step, let us make the unbounded recursive definition explicit. We do this by applying an applicative-order fixed-point operator to the functional associated to Euclid's algorithm, as displayed in Figure 10.

Procedure `GCD` is the fixed-point, i.e., the potentially unbounded compo-

```
(define Euclid-functional
  (lambda (GCD)
    (lambda (m n)
      (if (zero? m)
          n
          (GCD (remainder n m) m)))))

(define unbounded-GCD
  (fix2 Euclid-functional))
;;; where fix2 is an applicative-order fixed-point operator
;;; for binary functions.
```

Figure 10: Functional associated to the Euclidian algorithm

sition of Functional `Euclid-functional`.

3.2 An upper bound for the Euclidian algorithm

Proposition 1 *For any $a \in \mathbb{N}$, the number of iterations needed for computing* $\gcd(a, b)$ *or* $\gcd(b, a)$ *for all $b \in \mathbb{N}$ is bounded above by the following expression.*

$$\left\lceil \frac{\log(a\sqrt{5}-1)}{\log(\frac{1+\sqrt{5}}{2})} \right\rceil + 2$$

Proof: We use Lamé's result, as stated in Knuth's Art of Computer Programming [10, Page 79]. Given two integers a and b such that $a \leq b$, it takes at most $n_0 + 1$ iterations to compute $\gcd(b,\, a)$ whenever $b < F_{n_0+1}$, where F_n denotes the n-th Fibonacci number.

For our purposes, we do not know whether $b \geq a$. After one iteration, however, $\gcd(b,\, a)$ reduces to $\gcd(a,\, b \bmod a)$, whose computation is bounded by $n_0 + 1$. Therefore, given $a \in \mathbb{N}$ such that $a < F_{n_0}$ for some $n_0 \in \mathbb{N}$, computing either $\gcd(a,\, b)$ or $\gcd(b,\, a)$ for all $b \in \mathbb{N}$ requires at most $n_0 + 2$ steps.

Let us now find a closed expression for n_0. De Moivre's equation [10] gives us a closed form for the n-th Fibonacci term

$$F_n \;=\; \frac{1}{\sqrt{5}}\,(\varphi^n - \widehat{\varphi}^n)$$

where φ and $\widehat{\varphi}$ respectively denote the golden ratio $\frac{1+\sqrt{5}}{2}$ and its conjugate $\frac{1-\sqrt{5}}{2}$.

Since $|\widehat{\varphi}|$ is less than 1 and thus $\lim_{n\to\infty} \widehat{\varphi}^n = 0$, replacing $|\widehat{\varphi}|$ by 1 yields the following inequality:

$$\varphi^{n_0} - 1 \;\leq\; a\sqrt{5} \;\leq\; \varphi^{n_0} + 1$$

for any $a \in \mathbb{N}$.

Since we aim for an upper bound, we concentrate on the right-hand inequality, and obtain:

$$n_0 \;=\; \left\lceil \log_{\varphi}(a\sqrt{5} - 1) \right\rceil \;=\; \left\lceil \frac{\log(a\sqrt{5} - 1)}{\log(\frac{1+\sqrt{5}}{2})} \right\rceil$$

Hence the result. □

Figure 11 displays the function mapping a number into the number of iterations needed to compute the greatest common divisor of this number and any other number. As can be noted, this function increases logarithmically (for example, it maps 10^{10} to 50). We can also compare it with Lamé's upper bound, i.e., the number of iterations given the smaller of the two arguments of the Euclidian algorithm: five times the number of digits in base 10. Lamé's upper bounds exceed ours in all but five cases (from 5 to 9).

3.3 A bounded recursive definition of the Euclidian algorithm

Figure 12 displays the definition of the Euclidian algorithm in bounded form. The corresponding generating extension is shown in Figure 13. Given a number, it computes the upper bound on the number of calls to the GCD function, and unfolds the code of the GCD procedure accordingly. Figure 14 displays the result of specializing `GCD` with respect to 3. (As in Figure 3, the first test in the residual code could be simplified away by unfolding the functional once, as done in appendix.)

4 Practical Assessment

As illustrated above, partial evaluation inductively inlines the exponentiation and the Euclidian procedures. For the exponentiation procedure, a

```
(define upper-bound
  (let* ([sqrt-5 (sqrt 5.0)]
         [log-of-golden-ratio (log (/ (+ 1.0 sqrt-5) 2.0))])
    (lambda (a)
      (+ 2 (float->integer (/ (log (* sqrt-5 a))
                              log-of-golden-ratio))))))
```

Figure 11: Upper bound on the number of iterations in the Euclidian algorithm

```
(define bounded-GCD
  (lambda (m n)
    ((((integer->Church (upper-bound m))
       Euclid-functional)
      (bottom 'GCD))
     m n)))
```

Figure 12: The Euclidian algorithm in bounded form

```
(define bounded-GCD-gen
  (lambda (m)
    (let ([n (gensym! "n")])
      `(lambda (,n)
         ,((((integer->Church (upper-bound m))
             (lambda (GCD)
               (lambda (m n)
                 `(if (zero? ,m)
                      ,n
                      ,(let ([x (gensym! "n")])
                         `(let ([,x (remainder ,n ,m)])
                            ,(GCD x m)))))))
            (lambda (m n)
              `((bottom 'GCD) ,m ,n)))
           m n)))))
```

Figure 13: Generating extension for the Euclidian algorithm in bounded form

```
(define GCD-3   ;;; for all x, (GCD 3 x) = (GCD-3 x)
  (lambda (n0)
    (if (zero? 3)
        n0
        (let ([n1 (remainder n0 3)])
          (if (zero? n1)
              3
              (let ([n2 (remainder 3 n1)])
                (if (zero? n2)
                    n1
                    (let ([n3 (remainder n1 n2)])
                      (if (zero? n3)
                          n2
                          (let ([n4 (remainder n2 n3)])
                            (if (zero? n4)
                                n3
                                (let ([n5 (remainder n3 n4)])
                                  ((bottom 'GCD) n5 n4))))))))))))))
```

Figure 14: Specialized version of the GCD function

tight loop is replaced by a series of squarings and multiplications. For the Euclidian procedure, a tight loop is replaced by a series of conditional expressions. The former is clearly a win. The latter may or may not yield a more efficient program on modern architectures (cf. branch-delay slots, instruction cache, etc.).

In any case, if there were a static computation associated with the loop, it would be performed at partial-evaluation time and the resulting value would be either consumed at partial-evaluation time or inlined (cached) in the specialized program. Again, on a modern architecture, the latter could be a mixed blessing (cf. register allocation, etc.), unless the static computation is sizeable.

Nevertheless, these issues are more conjectural than structural. They depend on both the language processor and on the computer at hand. In our experience with high-level languages such as Scheme, ML, and C, the specialized code usually performs better than the source code, particularly when the Euclidian algorithm is enriched with a static computation at each iteration of the loop. (For example, we could equip it to return a pair: the greatest common divisor as usual, plus the result of some iterative arithmetic computation over the static argument.)

But the problem we were facing in this work was “can we specialize Euclid’s algorithm at all?” — and the answer is yes.

5 Conclusion

The Euclidian algorithm has no clear binding-time division and thus it cannot be specialized using offline partial-evaluation methods, as the exponentiation algorithm [5, 9]. By rephrasing it using bounded recursion, we have made it more amenable to automatic unfolding. Specializing it with respect to either of its arguments terminates and yields a finite, flat sequence of tests and remainder operations. This residual code corresponds to having unfolded the source procedure finitely many times.

Following this method, other algorithms with no clear binding-time division should be amenable to partial evaluation, when part of their input makes it possible to find an upper bound for the depth of their call graph. Similarly, if a probable upper bound n could be established statistically for a program, then a partial evaluator could unfold calls n times and then retain a call to the original code. The correctness of this transformation for a functional f is stated as follows:

$$\mathrm{fix}\, f \quad =_{\beta\eta} \quad \ulcorner n \urcorner\, f\, (\mathrm{fix}\, f)$$

where $\ulcorner n \urcorner$ denotes the n-th Church numeral. Such blind unfoldings are customary in most optimizing compilers and conventional partial evaluators. However, blind unfoldings were explicitly ruled out in partial evaluation and mixed computation, ten years ago [8], because not only do they most often lead to residual code explosion but also because they raise the thorny problem of finding a satisfactory n. We make do without them here, thus providing a concrete example of resource-bounded partial evaluation [6].

Acknowledgements

Grateful thanks to Rowan Davies, Laurie Hendren, Julia L. Lawall, Karoline Malmkjær, and René Vestergaard for comments and discussions, to Dany Breslauer and Peter Kornerup for bibliographic pointers, and to the reviewers and the editor for lucid comments.

The diagram of Section 1 was drawn with Kristoffer Rose’s XY-pic package.

A Optimized Versions of Power and GCD

As pointed out in the body of this article, the specialized versions of power and of GCD could be simplified a bit further, by folding the squaring of 1 in power, and the test over the constant argument in GCD. The former is achieved in Figure 15 and the latter in Figure 16.

References

[1] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, 1984.

[2] Andrew A. Berlin. Partial evaluation applied to numerical computation. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 139–150, Nice, France, June 1990. ACM Press.

[3] Andrew A. Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[4] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[5] 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.

[6] Olivier Danvy, Nevin C. Heintze, and Karoline Malmkjær. Resource-bounded partial evaluation. *ACM Computing Surveys*, 28(2):329–332, June 1996.

[7] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* The MIT Press and McGraw-Hill, 1991.

[8] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 1–14. North-Holland, 1988.

```
(define power
  (lambda (x n)
    (letrec ([loop (lambda (n)
                     (cond
                       [(even? n)
                        (sqr (loop (/ n 2)))]
                       [(= n 1)
                        x]
                       [else
                        (* x (sqr (loop (/ (1- n) 2))))]))])
      (if (zero? n)
          1
          (loop n)))))

(define bounded-power-gen
  (lambda (n)
    (let ([x (gensym! "x")])
      `(lambda (,x)
         ,(if (zero? n)
              1
              ((((integer->Church (1+ (float->integer (log2 n))))
                 (lambda (loop)
                   (lambda (n)
                     (cond
                       [(even? n)
                        `(sqr ,(loop (/ n 2)))]
                       [(= n 1)
                        x]
                       [else
                        `(* ,x (sqr ,(loop (/ (1- n) 2))))]))))
                (lambda (n)
                  `((bottom 'power) ,n)))
               n))))))
```

Figure 15: Optimized versions of Figures 1 and 9

[9] 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.

[10] Donald E. Knuth. *The Art of Computer Algorithms, Volume 1, Fundamental Algorithms.* Addison-Wesley, 1968.

```
(define GCD
  (lambda (m n)
    (letrec ([loop (lambda (m n)
                     (if (zero? m)
                         n
                         (loop (remainder n m) m)))])
      (if (zero? m)
          n
          (loop (remainder n m) m)))))

(define bounded-GCD-gen
  (lambda (m)
    (let ([n (gensym! "n")])
      `(lambda (,n)
         ,(if (zero? m)
              n
              (let ([x (gensym! "n")])
                `(let ([,x (remainder ,n ,m)])
                   ,((((integer->Church (1- (upper-bound m)))
                       (lambda (GCD)
                         (lambda (m n)
                           `(if (zero? ,m)
                                ,n
                                ,(let ([x (gensym! "n")])
                                   `(let ([,x (remainder ,n ,m)])
                                      ,(GCD x m)))))))
                      (lambda (m n)
                        `((bottom 'GCD) ,m ,n)))
                     x m))))))))
```

Figure 16: Optimized versions of Figures 4 and 13

[11] Mitchell Wand. From interpreter to compiler: a representational derivation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, pages 306–324, Copenhagen, Denmark, October 1985.

Recent BRICS Report Series Publications