Higher-Order and Symbolic Computation,, 12, 203–212 (1999)
© 1999 Kluwer Academic Publishers. Manufactured in The Netherlands.

Partial Evaluation of the Euclidean Algorithm, Revisited

C.S. LEE leecs@cs.uwa.edu.au
Department of Computer Science, University of Western Australia, Nedlands, Western Australia 6009

Abstract. The usual formulation of the Euclidean Algorithm is not well-suited to be specialized with respect to one of its arguments, at least when using offline partial evaluation. This has led Danvy and Goldberg to reformulate it using bounded recursion. In this article, we show how *The Trick* can be used to obtain a formulation of the Euclidean Algorithm with good binding-time separation. This formulation of the Euclidean Algorithm specializes effectively using standard offline partial evaluation.

Keywords: Euclidean Algorithm, binding-time separation, offline partial evaluation, The Trick

1. Introduction

1.1. Background

Partial Evaluation is an *automatic* technique for specializing programs. Semantically, a partial evaluator must satisfy the following equation,

$$[\![[\![pe]\!]\,(prog, input_1)]\!] input_2 = [\![prog]\!]\,(input_1, input_2),$$

for any *prog*, $input_1$ and $input_2$, where *prog* is coded in a language suitable for the partial evaluator, *pe*. Informally, the semantic brackets around *pe* and *prog* map a program to the function it computes. In words, the above equation says: A program that is specialized with respect to some input, when given the remaining input, outputs the same result as the original program given all the input at once. It is hoped, naturally, that the specialized program will execute more efficiently than the original program on the same input. For an overview of partial evaluation, the reader is referred to Jones et al. [5].

1.2. Partial evaluation of the Euclidean Algorithm

The usual formulation of the Euclidean Algorithm for computing the greatest common divisor of two non-negative integers is given in figure 1 in Scheme [4, 6]. As it stands, the program is not amenable to standard offline partial-evaluation techniques [5]. Let us briefly analyze why.

Suppose we are given the first argument. By symbolically unfolding the recursive `gcd` invocation, we observe that after a single unfolding, the value of neither actual parameter

```
(define (gcd s d)
  (if (zero? s)
      d
      (gcd (modulo d s) s)))
```

Figure 1. The usual `gcd` definition.

can be determined, since each is bound to an expression containing the unspecified parameter. In partial-evaluation terminology, *partial-evaluation-time determinable* variables are described as *static*. A variable that is not *static* is said to be *dynamic*. The static or dynamic nature of a variable is known as its *binding time*. We can now state the problem as follows: The usual `gcd` program suffers from *bad binding-time separation*.

1.3. This work

In their article, Danvy and Goldberg [3] observe that the number of iterations of `gcd` is bounded by a number that can be computed statically, given a static argument. They thus reformulate the Euclidean algorithm using bounded recursion. Given a static argument, the reformulated algorithm is then specialized by unfolding it an appropriate number of times.

Danvy and Goldberg's solution is essentially algorithmic. We propose an alternative solution that is partial-evaluation-centric. Our solution is based on the observation that the result of the modulo primitive is of *bounded static variation* for static moduli. The standard method for taking advantage of parameters of bounded static variation is to apply a transformation known in the literature as The Trick [5]. The Trick leads us to a reformulation of the `gcd` function with improved binding-time separation. This formulation specializes effectively and represents a natural solution to the problem of specializing the Euclidean Algorithm.

Subsequently, we observe that the program generated by this specialization contains a case statement that can be implemented efficiently using tabulation. From this observation, we derive another formulation of the Euclidean Algorithm whose specialization yields residual code with improved efficiency. Our Tabulation Solution demonstrates effective use of available static information.

1.4. Overview

For simplicity, we will only consider the specialization of the Euclidean Algorithm with respect to the first parameter. We also assume both actual parameters will be non-negative integers.

Section 2 considers the specialization of the usual Euclidean Algorithm and explains why offline partial evaluation is ineffective here.

Section 3 presents the partial evaluation of the Euclidean Algorithm using The Trick.

Section 4 presents the Tabulation Solution.

Section 5 compares the Bounded Recursive Solution to the Tabulation Solution and discusses the Euclidean Algorithm in general.

Section 6 concludes.

```
(define (gcd-0 d_0)
  (define (gcd-0-2 s_0 d_1)
    (if (zero? s_0)
        d_1
        (gcd-0-2 (modulo d_1 s_0) s_0)))
  (let ((s_1 (modulo d_0 6)))
    (if (zero? s_1)
        6
        (gcd-0-2 (modulo 6 s_1) s_1))))
```

Figure 2. The gcd definition of figure 1 specialized with initial argument 6.

2. The problem

We now explain what happens when the gcd definition of figure 1 is specialized using standard offline partial evaluation as described in Jones et al. [5]. Let us suppose that the gcd function is called with only its first argument specified. This means that the parameter s is initially static while parameter d is dynamic. Alternatively, we say that the gcd function has initial *binding-time division* (S, D) (S = Static, D = Dynamic). In the presence of *polyvariant* binding-time analysis, each function is permitted to have more than one binding-time division. In this case, the recursive call, `(gcd (modulo d s) s)` induces a version of gcd with binding-time division (D, S). This version of gcd in turn induces another with binding-time division (D, D). We therefore expect the body of the gcd definition to be unrolled once in the specialized code, regardless of the value of the specialized parameter. In figure 2, we show the result of specializing the gcd definition of figure 1 with initial argument 6.

The problem, as noted before, is poor binding-time separation. Put naively, we need more static parameters to specialize against. The Bounded Recursive Solution proposed by Danvy and Goldberg achieves this by essentially creating an iteration counter whose range is computed from the static argument. We derive instead a formulation with improved binding-time separation, as detailed below.

3. The Trick

Our approach to the specialization of gcd uses a *binding-time improvement* technique known as The Trick. As we will see, The Trick indeed enables us to achieve improved binding-time separation in the gcd function. There is however an interesting variation in its application here.

3.1. A simple example

We begin this section with a simple example illustrating a typical application of The Trick. Consider the assoc function which associates a key to an item, given an association list of (key, item) pairs, returning -1 when key is not found.

```
(define (assoc key a-list)
  (cond ((null? a-list) -1†)
        ((equal? key (caar a-list)) (cdar a-list†)
        (else (assoc key (cdr a-list)))))
```

Suppose we wish to specialize the following expression,

```
(1+ (assoc d '((a . 0) (b . 1) (c . 2)))
```

where d is dynamic. As the return value of assoc is dynamic, it is clear that the function 1+ (in this expression) cannot be applied at partial-evaluation time. However, observe that the return value of assoc here is of bounded static variation: it must be a member of the set $\{-1, 0, 1, 2\}$. Thus, it has to be possible to coax the partial evaluator to perform the evaluation of 1+ with respect to these values.

The Trick is to move the static context (1+ ...) to the possible return values of assoc, namely the points in the program indicated with† in the definition of assoc above. We define a new version of assoc that incorporates the 1+ operation. (For the sake of clarity, we have refrained from simplifying (1+ -1) into 0.)

```
(define (assoc1 key a-list)
  (cond ((null? a-list) (1+ -1))
        ((equal? key (caar a-list)) (1+ (cdar a-list)))
        (else (assoc1 key (cdr a-list)))))
```

The expression to be specialized becomes the following.

```
(assoc1 d '((a . 0) (b . 1) (c . 2)))
```

Now, the expressions (1+ -1) and (1+ (cdar a-list)) in assoc1 are static, since a-list is static, and therefore reduces away completely at partial-evaluation time.

3.2. The Trick and the Euclidean Algorithm

Let us attempt to apply the same reasoning process to the gcd function.

As before, we observe readily that there exists an expression whose result is of bounded static variation. This is (modulo d s), since we know from the operation of modulo that for $s > 0$,

$$0 \leq \texttt{(modulo d s)} < \texttt{s}.$$

Next, we might attempt to propagate the static context (gcd ... s), in the same manner as the previous example. There would appear to be a problem with this since modulo is a primitive operation. But in fact, all that is required to enable The Trick is to create a static variable that loops across the possible return values of (modulo d s).

```
(define (gcd2 s d)
  (if (zero? s)
      d
      (gcd2 (my-modulo d s) s)‡))

(define (my-modulo d s)
  (check-range (modulo d s) (- s 1)))

(define (check-range d s)
  (cond ((= s 0)0†)
        ((= s d) s†)
        (else (check-range d (- s 1)))))
```

In a sense, function `my-modulo` calls `check-range` to "ensure" that `(modulo d s)` is indeed in the expected range, $0, \ldots, \mathtt{s} - 1$: it is easily seen that for $\mathtt{s} > 0$,

`(my-modulo d s)` = `(modulo d s)`.

The essence of The Trick is to move the `(gcd2 ... s)`‡ computation into the body of `my-modulo` or any function transitively invoked, so that the `(gcd2 ... s)`‡ computation acts *directly* on the possible return values of `my-modulo`. Specifically, the possible return values of `my-modulo` correspond to the `0` and the `s` in the definition of `check-range` (where these expressions have been marked with †). We now rephrase the `gcd2` definition as follows.

```
(define (gcd2 s d)
  (if (zero? s)
      d
      (my-modulo d s s)))

(define (my-modulo d s1 s)
  (check-range (modulo d s1) (- s1 1) s))

(define (check-range d s2 s)
  (cond ((= s2 0) (gcd2 0 s))
        ((= s2 d) (gcd2 s2 s))
        (else (check-range d (- s2 1) s))))
```

The parameter `s` of `gcd2` is passed to `my-modulo` and then to `check-range` so that the *continuation*, $\lambda r \cdot (gcd2\ r\ s)$ can be reconstituted at its use points (the points indicated by †). The benefit of all this development is clear: the recursive `gcd2` invocation now possesses binding-time division (S, S), promising effective specialization.

In the presence of polyvariant binding-time analysis, specialization behaves as though there are two versions of the `gcd2` definition, with binding-time divisions (S, D) and (S, S). The latter reduces completely at partial-evaluation time. If our partial evaluator uses monovariant binding-time analysis (like Similix), we may substitute the recursive `gcd2` invocation with a call to the in-built `gcd` primitive, thereby also ensuring effective specialization.

```
(define (gcd2-0 d_0)
  (let ((d_1 (modulo d_0 6)))
    (cond ((= 5 d_1) 1)
          ((= 4 d_1) 2)
          ((= 3 d_1) 3)
          ((= 2 d_1) 2)
          ((= 1 d_1) 1)
          (else 6))))
```

Figure 3. The gcd2 function specialized with initial argument 6 using Similix.

Specializing the suitably formulated gcd2 program with initial argument 6 using Similix [2] yields the code in figure 3. This is a natural solution to the specialization problem, since it is intuitively clear that after a single division step, the gcd of s and d reduces to the gcd of s and one of $0, \dots, \mathtt{s} - 1$.

4. The tabulation solution

The specialized code of figure 3 lends itself readily to a reformulation using tabulation. Since the variable d_1 of function gcd2-0 assumes a value from the set $\{0, \dots, \mathtt{s} - 1\}$, it can be used to index into a vector comprising $\{gcd_{0,\mathtt{s}}, \dots, gcd_{\mathtt{s}-1,\mathtt{s}}\}$. In this way, we reduce a non-constant-time conditional expression to a constant-time vector access invocation. To this end, we propose the following reformulation.

```
(define (gcd3 s d)
  (if (zero?  s)
      d
      (lookup d s)))

(define (lookup d s)
  (vector-ref (list->vector[1] (tabulate (- s 1) s))
              (- (- s 1) (modulo d s))))

(define (tabulate i s)
  (if (= i 0)
      (list (gcd 0 s))
      (cons (gcd i s) (tabulate (- i 1) s))))
```

It is not difficult to see that when gcd3 is specialized with binding-time division (S, D), and the static parameter is not 0, the resulting program consists of a single (dynamic) reference into a completely static vector. While the tabulation solution is not directly related to The Trick, we present it as yet another formulation that makes it possible to use partial evaluation effectively to solve the problem of specializing the gcd function. Partially evaluating gcd3 with initial argument 6 yields the program in figure 4.

```
(define (residual d)
  (vector-ref #(1 2 3 2 1 6) (- 5 (modulo d 6))))
```

Figure 4. The gcd3 function specialized with initial argument 6.

```
(define (gcd3-gen s)
  (let ((d (gensym! "d")))
    `(define (residual ,d)
       ,(if (zero? s)
            d
            `(vector-ref ,(list->vector (tabulate (- s 1) s))
                         (- ,(- s 1) (modulo ,d ,s)))))))

(define (tabulate i s)
  (if (= i 0)
      (list (gcd 0 s))
      (cons (gcd i s) (tabulate (- i 1) s))))
```

Figure 5. The gcd3 generating extension.

We provide also, for gcd3, a coding of its generating extension. This is given in figure 5. (The generating extension of a program p of two input arguments is a program p' such that the following holds,

$$[\![[\![p']\!]s]\!]d = [\![[\![pe]\!]\,(p, s)]\!]d = [\![p]\!]\,(s, d).$$

In words, p' produces a specialized version of p given p's first argument.)

Applying the gcd3 generating extension to value 6 would yield the specialization of gcd3 with respect to initial argument 6, i.e., the program of figure 4.

5. Assessment

5.1. *Practical assessment*

1. The performance of the Bounded Recursive Solution of Danvy and Goldberg is dependent on compiler/machine-related factors.

   (A) Assuming a naive compilation model, the unfolded Euclidean Algorithm saves a number of function calls, which are generally expensive. However, standard Scheme compilers compile tail-recursive definitions to while-loops. Thus, function calls are not an issue.

   (B) Knuth gives an implementation of the Euclidean Algorithm where trivial assignments are removed completely (see p. 466 of [7]). Bentley [1] describes the process by which this is achieved as *Transfer Driven Loop Unrolling*. We present in figure 6 the unrolled Euclidean Algorithm in pseudo machine language.

```
          Goto COND
LOOP:     $R_2 \leftarrow R_0$ mod $R_1$
          If $R_2 = 0$ Then Return $R_1$
          ⋮
          $R_n \leftarrow R_{n-2}$ mod $R_{n-1}$
          If $R_n = 0$ Then Return $R_{n-1}$
          $R_0 \leftarrow R_{n-1}$ mod $R_n$
          If $R_0 = 0$ Then Return $R_n$
          $R_1 \leftarrow R_n$ mod $R_0$
COND:     If $R_1 > 0$ Then Goto LOOP
          Return $R_0$
```

Figure 6. A good implementation of the Euclidean Algorithm. The R_i's are registers. Initially, R_0 and R_1 contains d and s, with $0 \leq s < d < 2^b$. This procedure returns $gcd_{s,d}$. A natural choice for n is 2.

It is indeed conceivable that a suitably strong compiler might produce the code of figure 6 from an unrolled `gcd` definition. On the other hand, certain machines use dedicated registers for division. In this case, transfer driven loop unrolling is not applicable.

2. The tabulation solution is superior efficiency wise, but note that it is (exponentially!) larger in size compared to the bounded recursive solution. We discuss this further next.

5.2. *Space versus time*

1. Compared to the bounded recursive solution, the tabulation solution trades space for an improvement in time complexity. The space cost of the bounded recursive solution is $O(\log n)$ bits, where $n = s + d$, whereas that of the tabulation solution is $O(s \log n)$ bits. On the other hand, the bounded recursive solution has $O(\log s)$ time complexity, while the tabulation solution is constant time.[2]
2. In partial-evaluation terminology, all mentioned gcd formulations are *non-oblivious*: the values of dynamic variables affect the sequence of values assigned to static variables. Code explosion (or even infinite specialization) is always a problem to be wary of when specializing non-oblivious programs.

5.3. *General observations*

The Euclidean Algorithm has been extensively analyzed in the literature (see, for instance, [7]), even though Danvy and Goldberg were apparently the first to discuss it in the context of specialization.

1. The bounded recursive solution implements loop unrolling using a natural value for the degree of unfolding. Even in the absence of static input, loop unrolling may be used to remove trivial assignments at the low level, as mentioned previously. In this case, an

unfolding degree of 2 (the minimum required to eliminate trivial assignments) seems to be a natural choice. This decision is, however, based on low-level considerations. Indeed, while unfolding is generally applicable, it is not always clear *how much* is suitable for a problem, as noted in Danvy and Goldberg's paper [3].

2. A common treatment of non-oblivious algorithms (even without static input) is to create specialized code for *small* sub-problems. This approach is particularly relevant for divide-and-conquer algorithms where the number of *small* sub-problems grow proportionately in the size of the main problem to be solved. In our case, we cannot expect any improvement due to caching *gcd* of small arguments to "scale" with the size of the input arguments. Nevertheless, for $0 \leq m < n < 2^{16}$, empirical evidence suggests that we save, on average, about 15% of division steps, if we pre-compute $gcd_{a,b}$ for $0 \leq a < b < 16$.

6. Conclusion

We have presented an approach to the specialization of the Euclidean Algorithm based on an application of The Trick.

It is easy to overlook the opportunity for applying The Trick with the `gcd` function. Firstly, the result of the `gcd`'s `modulo` invocation, whilst bounded, is not explicitly represented anywhere in the program or the static argument. We may therefore not be aware of the fact that it is statically available. Moreover, The Trick typically involves propagating a static context into the body of a user-defined function, which `modulo` is not. For these reasons, the specialization of `gcd` using The Trick is particularly instructive. The article has highlighted, through the specialization of a simple function, the `gcd`, that once an expression has been identified to be of bounded static variation, The Trick can always be enabled by creating a static variable that loops over its possible values.

The solution arrived at using The Trick has suggested a tabulation reformulation. Specializing the tabulation formulation yields residual code whose execution time is independent of the value of the static parameter. This is not true of the other solutions. Thus, in a sense, the tabulation approach has allowed us to utilize available static information effectively, in the context of specializing the Euclidean Algorithm.

The `gcd` example for illustrating The Trick has arisen in a broader investigation of partial evaluation aiming to automate The Trick, based on a static analysis operating jointly with binding-time analysis and collecting bounded static variables [8].

Acknowledgments

The author would like to thank Olivier Danvy for his technical encouragement, and also the reviewers for their suggestions and comments.

Notes

1. `list->vector` is a Scheme primitive to convert a list to a vector [6].
2. This is under the assumption of constant-time array lookups.

References

1. Bentley, J.L. *Writing Efficient Programs*. Prentice Hall, Englewood Cliffs, NJ, 1982.
2. Bondorf, A. *Similix Manual, System Version 4.0*. Technical report. DIKU, University of Copenhagen, Denmark, 1991.
3. Danvy, O. and Goldberg, M. Partial evaluation of the Euclidian algorithm. *Lisp and Symbolic Computation*, **10**(2):101–111, 1997.
4. Dybvig, R.K. *The Scheme Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1987.
5. Jones, N.D., Gomard, C.K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
6. Kelsey, R., Clinger, W., and Rees, J. (Eds.). Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, **11**(1):7–105, 1998.
7. Knuth, D.E. The art of computer programming, *Fundamental Algorithms*. vol. 1, 3rd edition. Addison-Wesley, Reading, MA, 1997.
8. Lee, C.S. Ph.D. Thesis. Department of Computer Science, University of Western Australia, forthcoming.