Programming Techniques for Partial Evaluation *

Olivier Danvy

BRICS †
Department of Computer Science
University of Aarhus ‡

January 2000

Abstract

These lecture notes describe how to write generating extensions, i.e., dedicated program specializers. The focus is on compositional programs and their associated fold functions. Each generating extension is expressed as an instance of this fold function. A number of examples are considered and pointers to related work are provided.

*Lecture notes for Marktoberdorf'99, extended version.

†Basic Research in Computer Science (`http://www.brics.dk/`),
Centre of the Danish National Research Foundation.

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

1 Introduction

Partial evaluation is a program-transformation method for specializing programs with respect to part of their data. It is motivated by the fact that dedicated programs operate more efficiently than general-purpose ones. To specialize programs, one typically uses techniques also found in optimizing compilers: constant propagation, constant folding, call unfolding / inlining, dead-code elimination, memoization, folding, etc. A wide variety of literature on partial evaluation is available today, including a textbook by Neil Jones, Carsten Gomard, and Peter Sestoft [31], a number of overviews [8, 24, 30], at least three encyclopedia entries [14, 35, 36], the ACM SIGPLAN PEPM series,[1] several special issues of journals [15, 12, 29, 37], and the proceedings of occasional meetings and summer schools [4, 11, 20, 27]. More information is available at the web site of the PEPT mailing list.[2]

The goal of these lecture notes is to document the programming techniques for writing *generating extensions* (a term due to Andrei P. Ershov). A generating extension for a program p is a program p' which, given some data s, (1) performs the operations of p that depend on s and (2) constructs residual code for the remaining operations. Running p' on s thus returns a version of p that is specialized with respect to s.

> *Correctness criterion for specialization*
>
> Given a program p and its generating extension p', running p' on some data s yields a residual program p_s such that running p_s on the remaining data d yields the same result as running p on both s and d, provided p', p_s, and p all terminate.

The means of these lecture notes are fairly minimal: rather than relying on an existing partial evaluator, we have chosen to use the code-generating facilities of Scheme. To this end, the reader is provided with a handful of vanilla Scheme functions. We choose to use the Scheme language for its simplicity and the Petite Chez Scheme implementation for its availability and excellence.[3] All the following Scheme programs are available online from the author's home page, together with sample solutions for the exercises.

For the purpose of these lecture notes, we focus on compositional pro-

[1] `http://www.acm.org/pubs/contents/proceedings/series/pepm/`

[2] `http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pept/`

[3] Petite Chez Scheme is freely available at `http://www.scheme.com/`.

grams performing a recursive descent over their known input. To enforce this requirement, we express these programs with the fold function associated with the known input. The generating extension is thus obtained by passing code-generating functions to each fold function. The lecture notes are composed as a series of simple examples, setting up themes and inviting the reader to exercise some variations.

1.1 Prerequisites and notation

A passing familiarity with Scheme and quasiquotation will definitely help the reader. The most up-to-date reference for Scheme is the R^{5}RS [32]. Alan Bawden presented a survey of quasiquotation in Scheme at PEPM'99 [2]. Further information is available in Kent Dybvig's textbook [18] and in the Chez Scheme user's guide [19]. All these documents are freely available online.

In the rest of these notes, all efficiency measurements are carried out with Chez Scheme's timing facilities [19, Section 10.6]. Occasionally, to illustrate a point or an exercise, we display the transcript of an interactive Scheme session. In such sessions, "`> `" is the interactive prompt, preceding an expression to be evaluated; what follows is the result of the evaluation.

Throughout, we only consider exact integers [32], and occasionally, we rely on a simple record facility. We use `define-record` to define a new record and `case-record` to dispatch among possible records, much like the pattern matching of ML.

We also assume a basic familiarity with fold, as can be gathered from Graham Hutton's recent tutorial [28].

1.2 Overview

The rest of these lecture notes is organized as follows. We first describe a simple toolkit for program-generating programs in Scheme (Section 2) and then treat the power function, which is a canonical example in partial evaluation (Sections 3 and 4). We then consider a multiplication function (Section 5), the Euclidean algorithm (Section 6), and regular expressions (Section 7), before concluding (Section 8).

2 A Toolkit for Program-Generating Programs

We want to write programs that construct other programs. In Scheme, where programs are represented as S-expressions, constructing a residual program boils down to constructing an S-expression conforming to the syntax of Scheme. The rest of this section addresses the issue of fresh names (Section 2.1), common idioms for block structure and predefined functions (Section 2.2), advanced idioms (Section 2.3), and common errors (Section 2.4).

2.1 Fresh names

We want to generate residual programs with scope. Therefore it is essential that no name clashes occur. A name clash occurs if a residual program attempts to use the same variable with more than one purpose in the same scope. Names clashes are traditionally avoided by systematically generating fresh variables during the construction of residual programs, using a generator of new symbols (traditionally named "gensym"). We adopt this solution here, since alternatives such as combinators, de Bruijn indices, or de Bruijn levels are not as practical.

For readability as well as for debugging purposes, we parameterize the gensym function with a name stub. We also reset the gensym function prior to any specialization, so that performing the same specialization several times yields the same textual result.

2.2 Standard idioms

Let us review how to construct the usual syntactic forms of Scheme: function application, function declaration, conditional expressions, and lexical blocks.

Constructing a residual application is straightforward: given the operator and the operands, simply construct a flat list. We choose to inline this construction everywhere. However, we factor out the construction of residual applications of a predefined function when this function may give rise to algebraic simplifications (Section 2.2.1).

Constructing residual lambda-abstractions requires fresh names. The same goes for constructing residual let- or letrec-expressions. We choose to factor out these constructions (Sections 2.2.2 and 2.2.3).

Residual programs typically include nested let-expressions (`let` in Scheme), which can be written more concisely using the sequential `let*` construct

of Scheme. We thus provide the corresponding syntax constructor (Section 2.2.4).

2.2.1 Residualizing versions of predefined functions

Let us consider multiplication. Its simplest residualizing version takes two residual expressions and constructs the corresponding residual multiplication:

```
(define mk-*
  (lambda (x y)
    `(* ,x ,y)))
```

This simple residualizing function can be refined, since multiplying any number by 1 yields that number unchanged. A more refined version thus reads as follows:

```
(define mk-*
  (lambda (x y)
    (cond
      [(equal? x 1)
       y]
      [(equal? y 1)
       x]
      [else
       `(* ,x ,y)])))
```

Exercise 2.1: One might be tempted to further refine `mk-*` by taking into account that multiplying any number by 0 yields 0. Is this a good idea in general?

Hint—the non-zero argument may represent a non-trivial computation. □

Exercise 2.2: What happens if `mk-*` is applied to two residual literals? Does this suggest yet another refinement? □

Other residualizing functions can be found in the appendix.

2.2.2 Bindings

Constructing residual lambda-abstractions requires gensym. We factor it out in constructors such as `mk-lambda` and `mk-lambda2`. Given a name stub and a function expecting a fresh name and yielding a residual expression, `mk-lambda` constructs a unary residual lambda-abstraction.

Example 2.3:

```
> (mk-lambda "x" (lambda (x)
                   (mk-* x 100)))
(lambda (x0)
  (* x0 100))
```

□

Similarly, `mk-lambda2` constructs a binary residual lambda-abstraction, given two name stubs and a function expecting two fresh names and yielding a residual expression.

Example 2.4:

```
> (mk-lambda2 "x" "k" (lambda (x k)
                        `(,k ,x)))
(lambda (x0 k1)
  (k1 x0))
```

□

`mk-lambda` and `mk-lambda2` are defined in the appendix.

2.2.3 Block structure

Constructing residual blocks (`let` and `letrec`) also requires gensym. Similarly, we factor it out in two constructors `mk-let` and `mk-letrec`. Given a name stub, a residual expression, and a function expecting a fresh name and yielding a residual expression, `mk-let` constructs a unary residual let-expression.

Example 2.5:

```
> (mk-lambda "f"
             (lambda (f)
               (mk-let "x"
                       `(,f 1000)
                       (lambda (x)
                         (mk-let "y"
                                 `(,f 2000)
                                 (lambda (y)
                                   (mk-* x y)))))))
(lambda (f0)
  (let ([x1 (f0 1000)])
    (let ([y2 (f0 2000)])
      (* x1 y2))))
```

□

Similarly, `mk-letrec` constructs a unary residual letrec-expression, given a name stub, a function expecting a fresh name and yielding a residual lambda-abstraction, and a function expecting a fresh name and yielding a residual expression.

`mk-let` and `mk-letrec` are defined in the appendix.

2.2.4 Naming and sequentialization

The previous example is typical of residual code: it contains nested let-expressions, which would be more concisely expressed with one sequential let-expression. Such a let-expression is obtained using the `mk-let*` constructor defined in the appendix. Using `mk-let*` instead of `mk-let` in the previous example yields the following residual expression:

```
(lambda (f0)
  (let* ([x1 (f0 1000)]
         [y2 (f0 2000)])
    (* x1 y2)))
```

For simplicity, we define `mk-let` to be `mk-let*` in the rest of these lecture notes.

2.3 Advanced idioms

Higher-order program-generating programs typically require one to write functions mapping residual expressions to residual expressions. Two particular cases stand out.

2.3.1 Reflection: from expressions to functions

Let us go back to the first version of `mk-*` from Section 2.2.1. In essence, it maps two residual expressions (the operands of a residual multiplication) into another residual expression (a residual multiplication). This idiom recurs frequently, in that it will appear in the definition of every residualizing function where there is no opportunity to simplify.

Let us abstract this idiom:

```
(define reflect-fun  ;;; exp -> (exp -> exp)
  (lambda (e)
    (lambda (x)
      `(,e ,x))))

(define reflect-fun2 ;;; exp -> (exp * exp -> exp)
  (lambda (e)
    (lambda (x y)
      `(,e ,x ,y))))
```

We can now instantiate it at will:

```
> ((reflect-fun2 '*) 'a 'b)
(* a b)
> ((reflect-fun 'car) '...)
(car ...)
```

2.3.2 Reification: from functions to expressions

Symmetrically, one often needs to residualize a function mapping an expression to an expression—for example, a function obtained by reflection.

The corresponding idiom is abstracted as follows:

```
(define reify-fun    ;;; (exp -> exp) -> exp
  (lambda (f)
    (mk-lambda "x" f)))

(define reify-fun2   ;;; (exp * exp -> exp) -> exp
  (lambda (f)
    (mk-lambda2 "x" "x" f)))
```

For example, we can reify the residualizing version of multiplication:

```
> (reify-fun2 mk-*)
(lambda (x0 x1)
  (* x0 x1))
```

We can also reify the identity function:

```
> (reify-fun (lambda (x) x))
(lambda (x0)
  x0)
```

2.3.3 Summary and conclusion

We have described two functions arising frequently in higher-order generating extensions: `reflect-fun` and `reify-fun`.

- `reflect-fun` maps a residual expression to a residualizing function.
- `reify-fun` maps a residualizing function into a residual expression.

Further applications of reification and reflection are mentioned in Section 8.1.

2.4 Common errors

The following two common errors often arise when one programs with quasiquote and unquote in general and often with `reify-fun` and `reflect-fun` in particular:

- An expression is used as a value (e.g., an integer or a function), making Scheme signal a run-time error.
- A compound value (such as a Scheme closure) is used as an expression, yielding an ill-formed residual program.

These type errors would be detected earlier in a statically typed language such as ML or Haskell.

2.5 Summary and conclusion

We have briefly presented a toolkit for program-generating programs using Scheme, using quasiquote and unquote and factoring out the generation of fresh names. In the rest of these lecture notes, we occasionally use the Scheme function `eval` to evaluate residual programs. This function essentially provides a form of run-time code generation.

Exercise 2.6: Write a similar toolkit in ML or in Haskell.

Hint #1—define a datatype of residual expressions, and use a pretty-printer to obtain a residual program in concrete syntax.

Hint #2—to evaluate a residual program, one can either cut and paste it at the toplevel loop, or save it in a file and load this file. □

3 The Power Function

We begin with a canonical example of partial evaluation: the power function (Section 3.1). We first consider its naive generating extension (Section 3.2). The power function, however, implicitly uses the fold function for natural numbers (Section 3.3). We thus revisit both the power function (Section 3.4) and its generating extension (Section 3.5) using this fold function. We then consider how the fold function makes it possible to stage specialization (Section 3.6). Finally, we assess the overall approach of specializing the power function (Section 3.7).

3.1 The power function

We consider the linear power function. (A logarithmic version is considered in Section 4.)

```
(define power (lambda (x n)
                (letrec ([walk (lambda (n)
                                 (if (zero? n)
                                     1
                                     (* x (walk (- n 1)))))])
                  (walk n))))
```

3.2 Specializing the power function

Specializing `power` with respect to a given value for its induction variable makes it possible to unroll all the recursive calls, leaving a trail of residual multiplications. To this end, one instruments the power function by (1) making it generate a residual lambda-abstraction, and (2) using a residualizing multiplication.

```
(define power-gen
  (lambda (n)
    (mk-lambda "x" (lambda (x)
                     (letrec ([walk (lambda (n)
                                      (if (zero? n)
                                          1
                                          (mk-* x (walk (- n 1)))))])
                       (walk n))))))
```

And indeed applying `power-gen` to `5` yields the following simplified and non-recursive residual program:

```
(lambda (x0)
  (* x0 (* x0 (* x0 (* x0 x0)))))
```

The correctness criterion of partial evaluation reads as follows here: applying the function denoted by this residual program to any number x yields the same result as applying `power` to x and `5`.

For all `x` and `n`,

`((eval (power-gen n)) x)` = `(power x n)`

Exercise 3.1: Modify `power-gen` so that passing it `5` yields the following residual program:

```
(lambda (x0)
  (let* ([v1 (* x0 x0)]
         [v2 (* x0 v1)]
         [v3 (* x0 v2)])
    (* x0 v3)))
```

Hint—equip `walk` with a continuation. □

3.3 A fold function for the power function

There is nothing magical, however, about the power function. It is an instance of the following fold function for natural numbers:

```
(define nat
  (lambda (bc is)
    (letrec ([walk (lambda (n)
                     (if (zero? n)
                         bc
                         (is (walk (- n 1)))))])
      walk)))
```

`nat` corresponds to primitive iteration. It is parameterized with two functions: one for the base case and one for the induction step. We can instantiate `nat` to define the power function (Section 3.4) and its generating extension (Section 3.5). Conversely, we can specialize `nat` with respect to a natural number prior to instantiating it for obtaining a specialized power function or a specialized generating extension (Section 3.6).

3.4 The power function, revisited

We express the power function in terms of `nat` as follows. For the base case, we pass it the literal `1`,[4] and for the induction step, we pass it a curried multiplication function, partially applied to the number to exponentiate.

```
(define power-alt
  (lambda (x n)
    ((nat 1 (lambda (a) (* x a))) n)))
```

For all `x` and `n`,

`(power-alt x n)` = `(power x n)`

3.5 Specializing the power function, revisited

Similarly, we can express the generating extension of the power function in terms of `nat` as follows. For the base case, we pass it the residual literal `1`,[4] and for the induction step, we pass it a code-generating function residualizing a multiplication:

[4] In Scheme, the literal `1` and the residual literal `1` coincide, but that would not be the case in ML or Haskell: one would have an integer type, and the other would have the type of residual expressions.

```
(define power-gen-alt
  (lambda (n)
    (mk-lambda "x" (lambda (x)
                     ((nat 1 (lambda (a) (mk-* x a))) n)))))
```

And indeed applying `power-gen-alt` to `5` yields the same residual program as in Section 3.2.

For all `n`,

`(power-gen-alt n)` = `(power-gen n)`

Exercise 3.2: Modify `power-gen-alt` so that passing it `5` yields the same residual program as in Exercise 3.1. □

3.6 A generating extension for the fold function

Sections 3.4 and 3.5 make it clear that both the power function and its generating extension are instances of the fold function for natural numbers. Therefore, we can shift perspectives: instead of instantiating the fold function prior to specialization, we can specialize it prior to instantiation. To this end, let us write the corresponding generating extension. This generating extension is given a natural number and invokes the fold function with code-generating functions.

```
(define nat-gen
  (lambda (n)
    (mk-lambda2 "bc" "is" (lambda (bc is)
                            ((nat bc (reflect-fun is)) n)))))
```

N.B. In the definition of `nat-gen`, `reflect-fun` maps the residual identifier denoted by `is` into a residualizing function.

For example, applying `nat-gen` to `5` yields the following residual program:

```
(lambda (bc0 is1)
  (is1 (is1 (is1 (is1 (is1 bc0))))))
```

This residual program can be instantiated either to compute the power function (as in Section 3.4) or to specialize it (as in Section 3.5) with an exponent fixed to be `5`. We come back to this point in Section 6.4.

For all `n`, `bc`, and `is`,

```
((eval (nat-gen n)) bc is) = ((nat bc is) n)
```

Exercise 3.3: Write an alternative version of `nat-gen` so that applying it to `4` yields the following residual program:

```
(lambda (bc0 is1)
  (let* ([v2 (is1 bc0)]
         [v3 (is1 v2)]
         [v4 (is1 v3)]
         [v5 (is1 v4)])
    (is1 v5)))
```

Hint—instantiate `nat` with continuation-passing functions. □

3.7 A critical assessment

Let us now compare the efficiency of the power function and of its specialized instances. In Petite Chez Scheme, the arithmetic operators are overloaded in that they work in two modes: over fixed numbers (represented as simple data objects and thus incurring a fixed allocation cost) and over big numbers (represented as heap-allocated "bignums" and thus incurring a variable allocation cost). We are thus led to distinguish two cases:

For fixed numbers:

- The specialized instances consume virtually no space whereas the original function does.
- The specialized instances are about twice as fast as the original function.

For big numbers: there is no perceptible difference between the original function and its specialized instances.

The second item indicates that the real cost of the power function is not the interpretive overhead of the exponent, but the cost of the multiplications over bignums. The linear power function is thus mostly a pedagogical example of partial evaluation.

3.8 Summary and conclusion

We have treated the power function, which is a canonical example in partial evaluation. Specializing it with respect to a given exponent amounts to unrolling all its recursive calls and to constructing residual multiplications. We have shown that this specialization strategy is not ad hoc: it is an instance of the fold function for natural numbers, like the power function itself. This observation makes it clear that the generating extension, like the power function, terminates. It also suggests that one can specialize the fold function prior to instantiating it. Finally, we have measured the effect of partial evaluation on the linear power function in Scheme, and found it to be minor.

4 The Binary Power Function

We continue with the power function in its Russian-peasant form, taking the same steps as in Section 3, but in the form of exercises.

4.1 The power function

This version of the power function works by dividing the exponent by two rather than decrementing it. (N.B. We have short-cut one recursive call in the odd case, compared to traditional presentations of the Russian-peasant algorithm.)

```
(define power2
  (lambda (x n)
    (letrec ([walk (lambda (n)
                     (cond
                       [(zero? n)
                        1]
                       [(even? n)
                        (sqr (walk (quotient n 2)))]
                       [else
                        (* x (sqr (walk (quotient n 2))))]))])
      (walk n))))

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

4.2 Specializing the power function

Exercise 4.1: As in Section 3.2, instrument the power function and write a function `power2-gen` such that (1) it generates a residual lambda-abstraction, and (2) it uses a residualizing multiplication and a residualizing squaring function. □

Question 4.2: Suppose that the residualizing squaring function is defined in terms of the residualizing multiplication:

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

What is the effect of this change on the residual code? □

Exercise 4.3: As in Section 3.2, modify `power2-gen` to make it generate residual let-expressions. □

Question 4.4: If `power2-gen` generates residual let-expressions, does it matter that the residualizing squaring function is defined in terms of the residualizing multiplication? □

4.3 A fold function for the power function

Again, there is nothing magical about the binary power function. It is an instance of the following binary fold function for natural numbers:

```
(define nat2
  (lambda (bc ec oc)
    (letrec ([walk (lambda (n)
                     (cond
                       [(zero? n)
                        bc]
                       [(even? n)
                        (ec (walk (quotient n 2)))]
                       [else
                        (oc (walk (quotient n 2)))]))])
      walk)))
```

4.4 The power function, revisited

Exercise 4.5: As in Section 3.4, write a function `power2-alt` using the binary fold function defined just above. □

4.5 Specializing the power function, revisited

Exercise 4.6: As in Section 3.5, write a function `power2-alt-gen` using the binary fold function. □

4.6 A generating extension for the fold function

Exercise 4.7: As in Section 3.6, write a function `nat2-gen` using the binary fold function, and test it to compute the binary power function and to specialize it with respect to a fixed exponent. □

4.7 A critical assessment

Exercise 4.8: As in Section 3.7, compare the efficiency of the binary power function and of its specialized instances. □

4.8 Summary and conclusion

We have visited the binary power function, which is another canonical example in partial evaluation. Specializing it with respect to a given exponent amounts to unrolling all its recursive calls and to constructing residual multiplications and squaring functions. As in Section 3, we have shown that this specialization strategy is not ad hoc either: it is an instance of a fold function for natural numbers, like the power function itself. Therefore specialization, like the power function, terminates.

5 The Multiplication Function (Outline)

The goal of this section is to specialize the multiplication function into a combination of left-shifts and additions, given one of its arguments. For example, `left-shift` could be defined as follows:

```
(define left-shift
  (lambda (n x)
    (* x (expt 2 n))))
```

Usually, though, `left-shift` is implemented more efficiently (cf. `fxsll` in Petite Chez Scheme).

Exercise 5.1: Define the multiplication function so that specializing it with respect to `25` yields the following residual program:

```
(lambda (y0)
  (+ y0 (left-shift 3 (+ y0 (left-shift 1 y0)))))
```

□

Exercise 5.2: Define the multiplication function so that specializing it with respect to `25` yields the following residual program:

```
(lambda (y0)
  (let* ([v1 (left-shift 1 y0)]
         [v2 (+ y0 v1)]
         [v3 (left-shift 3 v2)])
    (+ y0 v3)))
```

Hint—use continuation-passing style. □

6 The Euclidean Algorithm

We consider the gcd function because of its interesting recursion pattern. At each recursive call, its two arguments are intertwined so that, unlike in the power function, there is no clear binding-time separation between these two arguments, making it a challenge to specialize the gcd function with respect to either of them (Section 6.1). The gcd function is an instance of the Euclidean algorithm. We review a few other such instances (Section 6.2) before turning to the fold function associated to the Euclidean algorithm (Section 6.3). We then stage its various instantiations (Section 6.4).

6.1 The Euclidean algorithm

The gcd function is standard, and presented, e.g., in Knuth's Volume 2 [33, Section 4.5.2]. We name it `gcd-std` rather than `gcd` in order not to redefine the resident gcd function in Petite Chez Scheme.

```
(define gcd-std
  (lambda (i j)
    (letrec ([walk (lambda (i j)
                     (if (zero? i)
                         j
                         (walk (remainder j i) i)))])
      (walk i j))))
```

In Sections 3 and 4, the induction variable of the power functions denotes a natural number. In the gcd function, however, it denotes a pair of integers, which presents the following challenge for partial evaluation.

Say that we want to specialize the gcd function with respect to one of its two arguments. This knowledge may enable us to unfold one recursive call. However, and in contrast to the power function, the zero test cannot be resolved. Therefore, short of speculative evaluation across both branches of the if, we are stuck. (Besides, should we wish to evaluate speculatively, how could we ensure that unfolding terminates?) Fortunately, either argument of gcd actually determines an upper bound on the number of recursive calls [33, page 360]. This upper bound makes it possible to unroll all the recursive calls [3, 13, 34].

Let us first write the gcd function in terms of the fold function `nat` defined in Section 3.3, using Finck's upper bound:

```
(define upper-bound
  (lambda (n)
    (if (< n 2)
        2
        (+ 1 (inexact->exact (ceiling (* 2 (log n))))))))
```

The upper bound provides the number of iterations. Since the base-case function should never be reached, we define it as an error-raising function. The induction-step function, which is naturally continuation-passing, carries out an iteration step.

```
(define nat-gcd
  (lambda (i j)
    (((nat (lambda (i j)
             (error 'nat-gcd "out of steam"))
           (lambda (k)
             (lambda (i j)
               (if (zero? i)
                   j
```

```
                (k (remainder j i) i)))))
  (upper-bound (min i j)))
 i j)))
```

Like the power function in Section 3.5, we can obtain a generating extension of the gcd function using the same fold function. For the base case, we pass it a constant residual function raising an error, and for the induction step, we pass it a code-generating function residualizing an iteration step.

```
(define nat-gcd-gen
  (lambda (i)
    (mk-lambda "j" (lambda (j)
                     (((nat (lambda (i j)
                              (mk-error 'nat-gcd "out of steam"))
                            (lambda (k)
                              (lambda (i j)
                                (mk-if (mk-zero? i)
                                       j
                                       (mk-let "r"
                                               (mk-remainder j i)
                                               (lambda (r)
                                                 (k r i)))))))
                       (upper-bound i))
                      i j)))))
```

The outcome is straightforward: all the iteration steps are unrolled. Here is a sample result:

```
> (nat-gcd-gen 4)
(lambda (j0)
  (let ([r1 (remainder j0 5)])
    (if (zero? r1)
        5
        (let ([r2 (remainder 5 r1)])
          (if (zero? r2)
              r1
              (let ([r3 (remainder r1 r2)])
                (if (zero? r3)
                    r2
                    (let ([r4 (remainder r2 r3)])
                      (error 'nat-gcd "out of steam")))))))))
```

The usefulness of the residual let-expressions is now plain, since the result of each remainder (except the innermost one) may be used twice.

For all `i` and `j`,

```
((eval (nat-gcd-gen i)) j)  =  ((eval (nat-gcd-gen j)) i)
                            =  (gcd-std i j)
```

Exercise 6.1: Compare the efficiency of `gcd-std` and of its specialized versions. Does it match your expectations? □

6.2 Instances of the Euclidean algorithm

We consider three more instances of the Euclidean algorithm: computing the partial quotients of a rational number (Section 6.2.1), computing the Euclidean coefficients recursively (Section 6.2.2), and computing them iteratively (Section 6.2.3).

6.2.1 Finite simple continued fractions

The partial quotients q_0, q_1, q_2, ..., q_n of a rational number i/j are defined by the following equality:

$$\frac{i}{j} = q_0 + \frac{1}{q_1 + \frac{1}{q_2 + \ldots \frac{1}{q_n}}}$$

They can be computed by the following variant of the Euclidean algorithm [33, Section 4.5.3]:

```
(define pq
  (lambda (i j)
    (letrec ([walk (lambda (i j)
                     (if (zero? i)
                         '()
                         (cons (quotient j i)
                               (walk (remainder j i) i))))])
      (walk i j))))
```

Given two natural numbers representing the numerator and the denominator of a rational number, `pq` constructs a list of its partial quotients.

6.2.2 The Euclidean coefficients (a recursive solution)

The Euclidean coefficients of any natural numbers i and j are two integers a and b such that $ai + bj = \gcd(i, j)$. Let us calculate two such ones.[5]

Base case: if $i = 0$ then $a = 0$ and $b = 1$ since, by definition of gcd, $\gcd(0, j) = j = 0i + 1j$.

Induction case: if $i \neq 0$ then by definition of gcd, $\gcd(i, j) = \gcd(j \operatorname{rem} i, i)$. By induction hypothesis, two integers a and b exist such that $\gcd(j \operatorname{rem} i, i) = a(j \operatorname{rem} i) + bi$. Therefore we are looking for two integers a' and b' such that $a'i + b'j = a(j \operatorname{rem} i) + bi$.

Since $i(j \operatorname{quo} i) + (j \operatorname{rem} i) = j$ a simple calculation leads one to $a(j \operatorname{rem} i) + bi = (b - a(j \operatorname{quo} i))i + aj$. Therefore $a' = b - a(j \operatorname{quo} i)$ and $b' = a$.

The Euclidean coefficients are thus computed as follows:

```
(define gcd-coeff
  (lambda (i j)
    (letrec ([walk (lambda (i j)
                     (if (zero? i)
                         '(0 . 1)
                         (let ([p (walk (remainder j i) i)])
                           (let ([a (car p)]
                                 [b (cdr p)])
                             (cons (- b (* a (quotient j i)))
                                   a)))))])
      (walk i j))))
```

6.2.3 The Euclidean coefficients (an iterative solution)

In Section 6.2.2, `gcd-coeff` is recursive: `walk` dives to the base case and computes a series of intermediate pairs all the way back. An iterative solution, however, exists, based on the observation that the intermediate pairs are obtained by a linear transformation. Such a transformation is canonically represented by a 2×2 matrix, and multiplying this (easily calculated) matrix with an intermediate pair yields the next pair in the series:

$$\begin{pmatrix} a' \\ b' \end{pmatrix} = \begin{bmatrix} -(j \operatorname{quo} i) & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

[5] Euclidean coefficients are not unique, but this issue is out of the scope of these lecture notes.

Since matrix multiplication is associative, we can write an iterative version of `gcd-coeff` using an accumulator initialized to the identity matrix. The result reads as follows:

```
(define gcd-coeff-acc
  (lambda (i j)
    (letrec ([walk (lambda (i j x11 x12 x21 x22)
                     (if (zero? i)
                         (cons x12 x22)
                         (let ([q (quotient j i)])
                           (walk (remainder j i)
                                 i
                                 (- x12 (* x11 q))
                                 x11
                                 (- x22 (* x21 q))
                                 x21))))])
      (walk i j 1 0 0 1))))
```

6.3 A fold function for the Euclidean algorithm

Along the same lines as Sections 3 and 4, let us now turn to a fold function associated to the Euclidean algorithm. The fold function we consider here is parameterized with two functions: one for the base case and one for the induction step.

```
(define euclid
  (lambda (bc is)
    (letrec ([walk (lambda (i j)
                     (if (zero? i)
                         (bc j)
                         (is i j (walk (remainder j i) i))))])
      walk)))
```

During its course of values, `euclid` passes intermediate values to `bc` and `is`. It is thus more akin to primitive recursion than to primitive iteration.

Exercise 6.2: Write a function `gcd-std-alt` computing the gcd function, in terms of `euclid`. □

Exercise 6.3: Write a function `gcd-noi` computing the number of iterations of the Euclidean algorithm, in terms of `euclid`. □

Exercise 6.4: Write a function `pq-alt` computing the partial quotients of a rational number represented as a pair of integers, in terms of `euclid`. □

Exercise 6.5: Write a function `gcd-coeff-alt` computing the Euclidean coefficients recursively, in terms of `euclid`. □

Exercise 6.6: Write a function `gcd-coeff-acc-alt` computing the Euclidean coefficients iteratively, in terms of `euclid`. □

Exercise 6.7: Write a generating extension `euclid-gen` for `euclid`. □

Example 6.8: Evaluating `(euclid-gen 5 8)` yields

```
(lambda (bc0 is1)
  (is1 5 8 (is1 3 5 (is1 2 3 (is1 1 2 (bc0 1))))))
```

This residual program illustrates the course of values of the Euclidean algorithm for the two Fibonacci numbers 5 and 8. □

For all `i`, `j`, `bc`, and `is`,

`((eval (euclid-gen i j)) bc is)` = `((euclid bc is) i j)`

We are now in position to stage the Euclidean algorithm, following the same approach used to stage the power function in Section 3.6.

6.4 Staging the Euclidean algorithm

When performing a Euclidean computation, we now have the choice of doing it either directly, as in Section 6.2, or generically through a fold function, as in Section 6.3. This section compares the performance of non-unfolded, dedicated code such as

```
(gcd-coeff 5 8)
```

with unfolded, non-dedicated code such as

```
(euclid-5-8 (lambda (d) ...) (lambda (i j) ...))
```

where `euclid-5-8` denotes the value of

```
(lambda (bc0 is1)
  (is1 5 8 (is1 3 5 (is1 2 3 (is1 1 2 (bc0 1))))))
```

which is the result of evaluating `(euclid-gen 5 8)`.

The purpose of this comparison is only pedagogical, since we disregard the cost of evaluating `(euclid-gen 5 8)` and its result.

6.4.1 Experiments

We conduct five comparisons, corresponding to the exercises of Section 6.3:

GCD-STD: calculating the gcd of two numbers. This is the base experiment: we compare the non-unfolded, dedicated code and the unfolded, non-dedicated code (where the result is already computed).

GCD-NOI: calculating the number of iterations of the Euclidean algorithm. This is a witness experiment.

- The non-unfolded, dedicated code iterates and increments an integer.
- The unfolded, non-dedicated code instantiates the base case and the induction step; at each induction step, it increments an integer.

PQ: calculating the partial quotients of a rational number.

- The non-unfolded, dedicated code iterates and computes a quotient and a remainder at each iteration.
- The unfolded, non-dedicated code instantiates the base case and the induction step; at each induction step, it computes one quotient.

GCD-COEFF: calculating Euclidean coefficients, recursively.

- The non-unfolded, dedicated code computes a remainder at each call; at each return, it computes a quotient and allocates a pair.
- The unfolded, non-dedicated code instantiates the base case and the induction step; at each induction step, it computes one quotient and allocates one pair.

GCD-COEFF-ACC: calculating Euclidean coefficients, iteratively.

- The non-unfolded, dedicated code iterates and computes a quotient, a remainder, two subtractions, and two multiplications at each iteration.
- The unfolded, non-dedicated code instantiates the base case and the induction step; at each induction step, it computes one quotient, one subtraction, and one multiplication.

We chose to use the Chez Scheme compiler instead of the Petite Chez Scheme interpreter, to avoid interpretive noise. The measurements were conducted on a Pentium 150MHz running Linux. In each case, we performed 5000 runs of the non-unfolded, dedicated code and 5000 runs of the unfolded, non-dedicated code, and we measured their performance with Chez Scheme's `time` function [19, Section 10.6]. Throughout, the Chez Scheme process occupied a handful of megabytes and thus incurred no swapping costs.

6.4.2 Analysis of the results

Figures 1 and 2 display the results. The left columns correspond to CPU time, including garbage collection, and the right columns correspond to allocated memory. Each line groups two graphs and corresponds to one experiment (GCD-STD, GCD-NOI, PQ, GCD-COEFF, and GCD-COEFF-ACC).

In each graph, the horizontal axis accounts for the number of iterations of the Euclidean algorithm, and the vertical axis accounts for time (left column) and space (right column). Figure 1 displays up to 50 iterations and Figure 2 displays up to 250 iterations. (Further experiments up to 500 iterations yield curves that are similar to the ones in Figure 2.)

Each graph contains two curves: one for an unfolded, non-dedicated function and one for a non-unfolded, dedicated function.

GCD-STD and GCD-NOI: as could be expected, the unfolded, non-dedicated code incurs essentially no cost in time and space, whereas the non-unfolded, dedicated code pays for iterating and computing quotients.

PQ: Up to 45 iterations, the unfolded, non-dedicated code is about twice as fast as the non-unfolded, dedicated code, for about the same memory usage. Beyond this threshold, the curves diverge, reflecting the increased costs of bignum calculations. The PQ figures match the

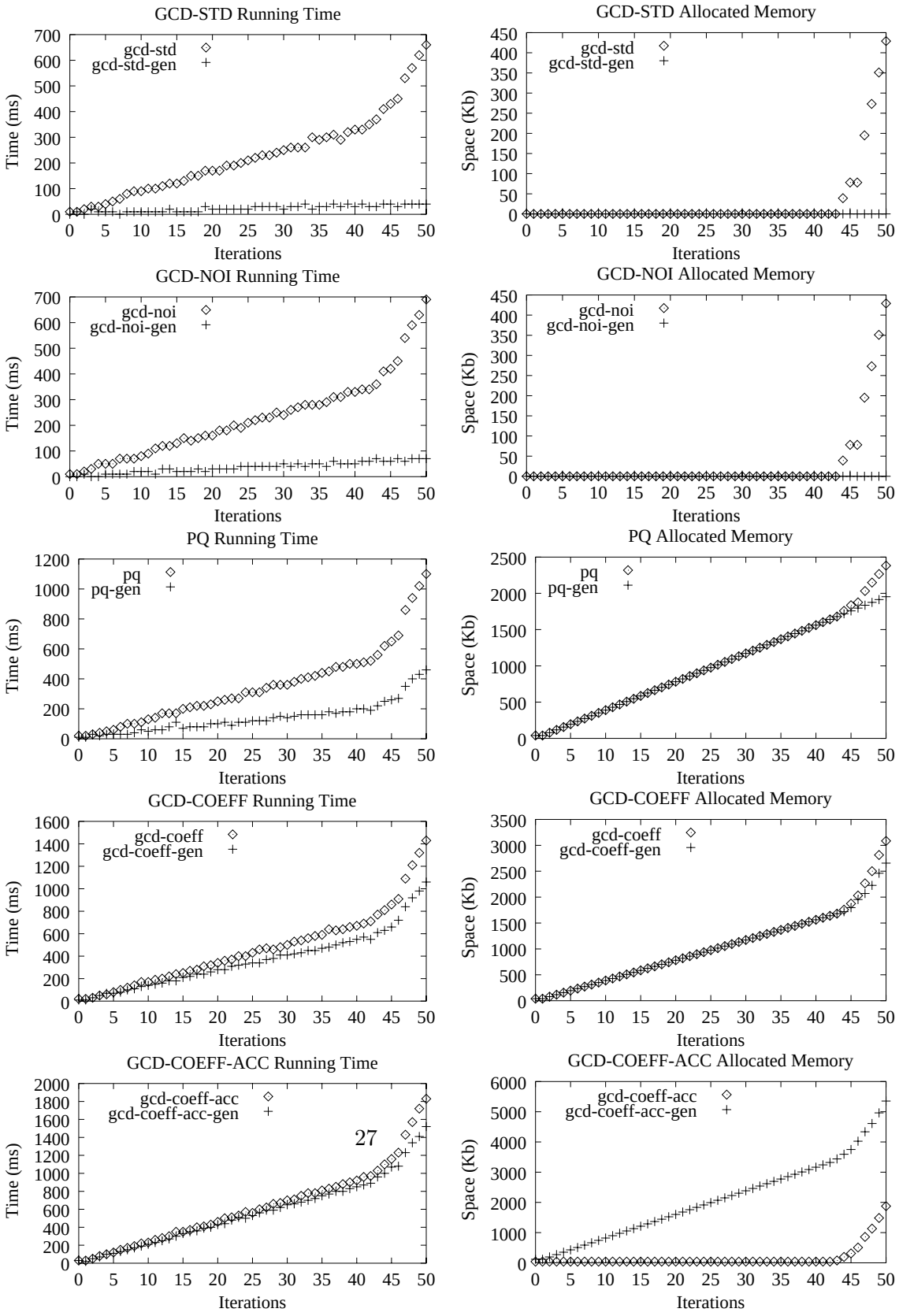

Figure 1: Benchmarks (up to 50 iterations)

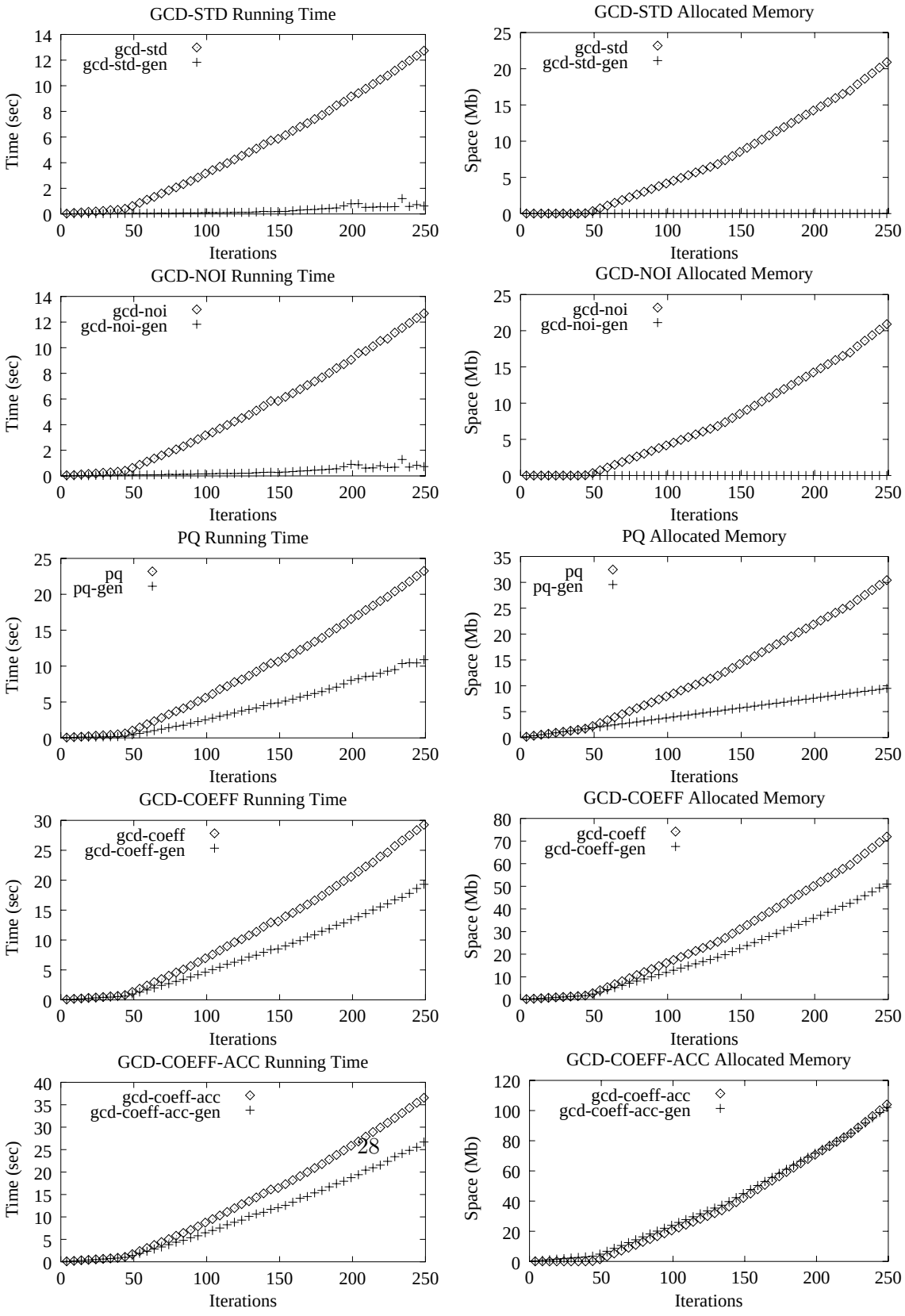

Figure 2: Benchmarks (up to 250 iterations)

GCD-STD figures in that adding the GCD-STD curve to the lower PQ curve yields the higher PQ curve.

GCD-COEFF: Up to 45 iterations, the time and space curves essentially match. Beyond this threshold, the curves diverge: the unfolded, non-dedicated code is about 50% faster than the non-unfolded, dedicated code, while allocating about 70% less memory.

GCD-COEFF-ACC: Up to 45 iterations, the time curves essentially match, but the non-unfolded, dedicated code allocates virtually no space, probably reflecting a better register-allocation policy of the accumulator. Beyond this threshold, the memory usages essentially match, while the unfolded, non-dedicated code slowly gets to be faster than the non-unfolded, dedicated code.

Both the memory curve and the time curve suggest that the threshold between fixed-number and big-number arithmetic is reached after about 45 iterations.

6.5 Summary and conclusion

Somewhat recreationally, we have explored the Euclidean algorithm in the light of partial evaluation, unfolding it given an upper bound on the number of iterations, and writing it generically and unfolding the generic version prior to instantiating it. We also have measured the relative performances of unfolded, non-dedicated code and non-unfolded, dedicated code; as contrived as the overall experiment may be—the Euclidean algorithm is already very efficient and we are using Chez Scheme's bignum arithmetic off-the-shelf—the results are as could be expected, showing that unfolding calls and propagating and folding constants can pay off.

Exercise 6.9: Write the fold function associated to the binary Euclidean algorithm [33, page 338]. □

7 Regular Expressions

The goal of this section is to write an interpreter for regular expressions and its associated generating extension. The interpreter takes a regular expression and a list of tokens, and decides whether the list of tokens belongs

to the set represented by the regular expression. The generating extension compiles a regular expression into a dedicated decision procedure written in Scheme.

We first define the regular expressions considered here (Section 7.1), and then the corresponding interpreter (Section 7.2) and generating extension (Section 7.3). We also consider some variations and extensions (Section 7.4).

7.1 Regular expressions and the associated fold function

The regular expressions we consider here have four constructors:

```
(define-record (TOKEN token))
(define-record (SEQ regexp regexp))
(define-record (ALT regexp regexp))
(define-record (STAR regexp))
```

`TOKEN` defines a fixed token, `SEQ` defines a sequence of regular expressions, `ALT` defines two alternative regular expressions, and `STAR` defines a Kleene closure.

For example, writing `TOKEN` without tag, `SEQ` with infix $\cdot$, `ALT` with infix $|$, and `STAR` with superscripted $*$, the regular expression $10 \cdot 20$ represents the set $\{$`(10 20)`$\}$; the regular expression $10 \cdot (20 \mid 30) \cdot 40$ represents the set $\{$`(10 20 40)`, `(10 30 40)`$\}$; and the regular expression $10^* \cdot 20$ represents the set $\{$`(20)`, `(10 20)`, `(10 10 20)`, ...$\}$.

The fold function for the data type of regular expressions is defined as follows.

```
(define fold-regexp
  (lambda (token seq alt star)
    (letrec ([walk (lambda (e)
                     (case-record e
                       [(TOKEN s)
                        (token s)]
                       [(SEQ e1 e2)
                        (seq (walk e1) (walk e2))]
                       [(ALT e1 e2)
                        (alt (walk e1) (walk e2))]
                       [(STAR e)
                        (star (walk e))]

                       [else
                        (error 'fold-regexp "not a regexp ~s" e)]))])
      walk)))
```

7.2 An interpreter for regular expressions

An interpreter for regular expressions takes a regular expression and a list of tokens, and determines whether the list belongs to the set represented by the regular expression. If it does, then we say that the regular expression 'accepts' the list, and otherwise that it 'rejects' it.

Exercise 7.1: Write an interpreter `ire` for regular expressions using `fold-regexp`. The interpreter should return `#t` if the list is accepted, and `#f` otherwise.

Hint—the type of the instance of `walk`, in `fold-regexp`, should be as follows:

```
regexp -> list-of-tokens * (list-of-tokens -> boolean) -> boolean
```

For example, here is a continuation-passing function for testing the first element of a list of tokens. If the test is positive, the rest of the list is sent to the continuation.

```
(define test-token
  (lambda (a ts k)
    (and (not (null? ts))
         (equal? a (car ts))
         (k (cdr ts)))))
```

N.B. Section 7.3 provides some hints about how to write `ire`. □

7.3 A generating extension for regular expressions

Exercise 7.2: Write a generating extension `ire-gen` for regular expressions using `fold-regexp`. The generating extension should take a regular expression and compile it into Scheme code, and this Scheme code should satisfy the correctness criterion of specialization.

For all `re` and `ts`,

```
((eval (ire-gen re)) ts) = (ire re ts)
```

For example, using the same convention as above, $10 \cdot 20$ could be compiled into:

```
(lambda (ts0)
  (test-token 10 ts0 (lambda (ts1)
                       (test-token 20 ts1 (lambda (ts2)
                                            (null? ts2))))))
```

Similarly, $10 \cdot (20 \mid 30) \cdot 40$ could be compiled into:

```
(lambda (ts0)
  (test-token 10 ts0 (lambda (ts1)
                       (let ([k4 (lambda (ts2)
                                   (test-token 40 ts2 (lambda (ts3)
                                                        (null? ts3))))])
                         (or (test-token 20 ts1 (lambda (ts5)
                                                  (k4 ts5)))
                             (test-token 30 ts1 (lambda (ts6)
                                                  (k4 ts6)))))))))
```

And finally, $10^* \cdot 20$ could be compiled into:

```
(lambda (ts0)
  (letrec ([star1
            (lambda (ts2)
              (or (test-token 20 ts2 (lambda (ts3)
                                       (null? ts3)))
                  (test-token 10 ts2 (lambda (ts4)
                                       (star1 ts4)))))])
    (star1 ts0)))
```

N.B. $(42^*)^*$ should be compiled into a non-diverging residual program such as the following one:

```
(lambda (ts0)
  (letrec ([star1
            (lambda (ts2)
              (or (null? ts2)
                  (letrec ([star3
                            (lambda (ts4)
                              (or (and (not (equal? ts2 ts4))
                                       (star1 ts4))
                                  (test-token 42
                                              ts4
                                              (lambda (ts5)
                                                (star3 ts5)))))])
                    (star3 ts2))))])
    (star1 ts0)))
```

□

Question 7.3: The second example suggests that the compiler should name the continuation of an `ALT` expression. Why? □

Exercise 7.4: Observing that the compiled regular expressions contain many eta-redexes, how could `mk-lambda` be modified to avoid them? For example, $10^* \cdot (20 \mid 30)$ could be compiled into:

```
(lambda (ts0)
  (letrec ([star1 (lambda (ts2)
                    (or (or (test-token 20 ts2 null?)
                            (test-token 30 ts2 null?))
                        (test-token 10 ts2 star1)))])
    (star1 ts0)))
```

□

Question 7.5: In Scheme, or-expressions can take arbitrarily many arguments, not just two. How could this be used so that the compiled code just above reads instead as follows?

```
(lambda (ts0)
  (letrec ([star1 (lambda (ts2)
                    (or (test-token 20 ts2 null?)
                        (test-token 30 ts2 null?)
                        (test-token 10 ts2 star1)))])
    (star1 ts0)))
```

□

7.4 Variations and extensions

7.4.1 Efficiency

The compiled version of $(10 \cdot 20) \mid (10 \cdot 30)$, or of $10 \mid 10$ for that matter, is sub-optimal, since the token `10` is tested repeatedly.

Food for thought 7.6: How could the situation be improved?

Hint—think of the *nullable*, *firstpos* and *followpos* functions in the Dragon book [1, page 135]. □

7.4.2 A wild card

Let us extend the language of regular expressions with a wild card, i.e., a constructor that matches any token.

```
(define-record (ANY))
```

Exercise 7.7: Extend the fold function, the interpreter, and its generating extension to cater for this new constructor.

Hint—define a variant of `test-token`. □

7.4.3 Variables

Let us extend the language of regular expressions with non-linear variables, i.e., with a constructor introducing a variable such that all other occurrences of this variable must match the same token.

```
(define-record (VAR symbol))
```

For example, the regular expression $x \cdot 20 \cdot x$ accepts all the lists `(v1 v2 v3)` such that `v2` = `20` and `v1` = `v3`.

The goal of this section is to extend the interpreter and its generating extension to cater for this new constructor. For example, given a function for variables such as the following one,

```
(define name-token
  (lambda (ts k)
    (and (not (null? ts))
         (k (car ts) (cdr ts)))))
```

the regular expression $x \cdot 20 \cdot x$ could be compiled into the following residual code:

```
(lambda (ts0)
  (name-token ts0 (lambda (v1 ts2)
                    (test-token 20 ts2 (lambda (ts3)
                                         (test-token v1 ts3 null?))))))
```

The key point of this residual code is that the first element of the list of tokens is named `v1`, using `name-token`, and the second call to `test-token` checks whether `v1` and the third element of the list denote the same token.

Exercise 7.8: Extend the interpreter to handle variables. To this end, equip it with an *environment* mapping variables to tokens. □

Exercise 7.9: Extend the generating extension to handle variables, so that all references to the environment are resolved at compile time. (Indeed, there is no track of any environment in the example just above.) □

Food for thought 7.10: Alternatively, the interpreter could be equipped with a run-time stack of tokens. Named tokens would be pushed on that stack and the environment, which would live at compile time, would map compile-time variables to run-time stack offsets. □

7.4.4 Accepting prefixes

Suppose that we want to check whether the *prefix* of a list of tokens (instead of the whole list) is accepted by a regular expression.

Exercise 7.11: Modify the interpreter so that it yields either a rejection value or the suffix of the input list of tokens, using the two following records to distinguish between failure and success:

```
(define-record (ACCEPT list-of-tokens))
(define-record (REJECT))
```

Hint—start by modifying `test-token`. □

7.4.5 Multiple answers

There may be several ways for a list of tokens to be accepted by a regular expression.

Exercise 7.12: Modify the interpreter so that it yields the number of ways in which a regular expression accepts a list of tokens—0 signifying failure. □

7.4.6 Collecting suffixes

Building on Sections 7.4.4 and 7.4.5, we want to construct the list of suffixes of the input list of tokens such that the prefix of each of these lists is accepted by the regular expression.

Exercise 7.13: Modify the interpreter to make it construct a list of accepted suffixes.

Hint—add an extra continuation. □

7.5 Summary and conclusion

We have gone through a classical example of partial evaluation, namely an interpreter for regular expressions. Specializing it with respect to a given regular expression amounts to unrolling all of its recursive calls and constructing residual calls to atomic tests. As in the previous sections, we have shown that this specialization strategy is not ad hoc: the compiler is an instance of a fold function which is defined inductively on the structure of regular expressions and therefore it is guaranteed to terminate.

Further references about interpreters for regular expressions include Neil Jones, Carsten Gomard, and Peter Sestoft's textbook [31], Bob Harper's article on proof-directed debugging [25], and Andrzej Filinski and the author's work on continuations [10].

The programming techniques reviewed in Section 7.4 apply directly to writing programming-language interpreters and specializing them with respect to a program, which is one of the most celebrated applications of partial evaluation. Further references about programming-language interpreters include the above-mentioned textbook, Yoshihiko Futamura's seminal article on self-applicable partial evaluation [23], and the author's joint work on imperative languages [7, 16].

8 Conclusion and Issues

Through a series of examples and exercises, we have illustrated some programming techniques for partial evaluation, focusing on generating extensions and structuring them as instances of fold functions. We conclude these lecture notes with some further issues.

8.1 Type-directed partial evaluation

The functions `reify-fun` and `reflect-fun` of Section 2.3 can be generalized to work at arbitrary type rather than at `exp -> exp`. The resulting type-indexed reify function implements a practical and efficient normalization function for the λ-calculus. This normalization function is efficient because—like the generating extensions considered here—it requires

no translation overhead and operates at native speed. It is also practical because, for example, it enables one to obtain the effect of all the generating extensions considered here without writing a single quasiquote and unquote: these are all factored out in the normalization function. More detail and further bibliographic references can be found in the author's 1998 lecture notes on type-directed partial evaluation [9] and in Andrzej Filinski's semantic account [22].

8.2 Sharing residual code

The following observation applies to all the examples considered here: when specializing a compositional program with respect to a given data structure for its induction variable, we map this data structure into the control structure of the residual program, homomorphically. This means that when the data structure is tree-shaped, the control-flow graph of the specialized program is also tree-shaped. In particular, when the source program is recursive rather than inductive over a part of the data structure (e.g., a Kleene star in regular expressions or a while loop in an imperative program), the residual program contains a recursive definition (e.g., an instance of `star` in Section 7).

Exercise 8.1: In this light, revisit how regular expressions are traditionally compiled into finite automata in, e.g., the Dragon Book [1, Sections 3.7, 3.8, and 3.9]. □

DAG-shaped, rather than tree-shaped, residual programs can be obtained as follows.

By construction, each residual program point is a version of a source program point, specialized with respect to some static values. Suppose that we index some source program points with some static values, together with a pointer to the corresponding residual code. This indexing makes it possible, at specialization time, to *share* residual computations. If, later in the specialization, the same source program point needs to be specialized with respect to the same static values, there is no need to duplicate the specialization and the residual code. This strategy is known as *polyvariant program-point specialization*, and has proven instrumental, e.g., to derive Knuth, Morris, and Pratt's linear string matcher out of a naive quadratic one [6]. More detail can be found in Jones, Gomard, and Sestoft's textbook,

including a study of the program speedups enabled by partial-evaluation techniques [31].

8.3 Data specialization

There is ample room for variation in the area of partial evaluation. One is *data specialization*. The issue is crystallized in the following example. Rather than generating a residual program such as

```
(lambda (bc0 is1)
  (is1 3 5 (is1 2 3 (is1 1 2 (bc0 1)))))
```

one could store the static values 3, 5, etc. in a residual table and generate a more compact residual program accessing this table at run time. Such data specialization has proven to be practically useful in scientific computing, where it allows one to share and reuse the result of expensive computations. More detail and further bibliographic references can be found in Sandrine Chirokoff, Charles Consel and Renaud Marlet's recent article on program and data specialization [5].

8.4 Binding times

Let us conclude with the following quote from Jerome Feldman's PhD thesis, written in the early sixties.

> "One of the most difficult concepts in translator writing is the distinction between actions done at translation time and those done at run time. Anyone who has mastered this difference has taken the basic step towards gaining an understanding of computer languages." [21]

This quote is visionary in that the issue it raises occurs both in Yoshihiko Futamura's statement of the problem of self-applicable partial evaluation [23] and in Neil Jones's solution [31]. Jones's solution hinges on a *binding-time analysis* occurring prior to specialization and determining which parts of a source program can be safely performed at specialization time and which parts should be reconstructed. This solution spawned the so-called offline approach to partial evaluation, which is a topic of active research today [17].

Feldman's point is as valid today as it was almost 40 years ago: even though binding-time analyses provide automated support to distinguish, e.g., between compile time and run time, this support does not dispense one from understanding personally the distinction between binding times and its consequences for programming.[6] This distinction is the underlying theme of these lecture notes.

Acknowledgments: Grateful thanks to the organizers of Marktoberdorf'99 for the invitation, to the staff, for a friendly and flawless organization, and to the other participants, for the pleasant and studious atmosphere which seems to make Marktoberdorf, year after year, such a successful scientific retreat. I would also like to thank Ralf Steinbrueggen for outstanding editorship, Andrzej Filinski for comments and sound encouragement, Daniel Damian and Morten Rhiger for comments and editorial assistance, Torben Amtoft, Anindya Banerjee, Peter Chang, Charles Consel, Kent Dybvig, Mayer Goldberg, Hanne Gottliebsen, John Hatcliff, Niels O. Jensen, Julia Lawall, Gilles Muller, David Schmidt, Ulrik Schultz, Sebastian Skalberg, Charles Stewart, René Vestergaard, and Oscar Waddell for timely comments, and Sandrine Chirokoff, Irène Danvy, Bernd Grobauer, Karoline Malmkjær, and Zhe Yang for discussions and feedback.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Alan Bawden. Quasiquotation in Lisp. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, pages 4–12, San Antonio, Texas, January 1999. Available online at `http://www.brics.dk/~pepm99/programme.html`

[3] Jon Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982.

[4] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[6] A similar point can be made for static type-checking.

[5] Sandrine Chirokoff, Charles Consel, and Renaud Marlet. Combining program and data specialization. *Higher-Order and Symbolic Computation*, 12(4):309–335, 1999.

[6] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.

[7] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[8] 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.

[9] Olivier Danvy. Type-directed partial evaluation. In Hatcliff et al. [27], pages 367–411. Extended version available as the lecture notes BRICS LN-98-3.

[10] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[11] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, Dagstuhl, Germany, February 1996. Springer-Verlag.

[12] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Symposium on Partial Evaluation*, volume 30, number 3es of *Computing Surveys*. ACM Press, September 1998.

[13] Olivier Danvy and Mayer Goldberg. Partial evaluation of the Euclidean algorithm. *Lisp and Symbolic Computation*, 10(2):101–111, 1997.

[14] Olivier Danvy and John Hatcliff. Partial evaluation. In David Hemmendinger, Anthony Ralston, and Edwin Reilly, editors, *Encyclopedia of Computer Science (Fourth Edition)*. Macmillan Reference/Grove Dictionaries, 1999.

[15] Olivier Danvy and Carolyn L. Talcott, editors. *Special Issue on Partial Evaluation, Part I*, Higher-Order and Symbolic Computation, Vol. 12, No. 4, 1999.

[16] Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996. Springer-Verlag. Extended version available as the technical report BRICS-RS-96-13.

[17] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*. To appear.

[18] R. Kent Dybvig. *The Scheme Programming Language, Second Edition*. Prentice Hall, 1996. Available online at `http://www.scheme.com/tspl2d/`.

[19] R. Kent Dybvig. *Chez Scheme User's Guide*. Cadence Research Systems, 1998. Available online at `http://www.scheme.com/csug/`.

[20] Andrei P. Ershov, Dines Bjørner, Yoshihiko Futamura, K. Furukawa, Anders Haraldsson, and William Scherlis, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987*, New Generation Computing, Vol. 6, No. 2-3. Ohmsha Ltd. and Springer-Verlag, 1988.

[21] Jerome A. Feldman. *A formal semantics for computer oriented languages*. PhD thesis, Department of Mathematics, Carnegie Institute of Technology, Pittsburgh, Pennsylvania, 1964.

[22] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag. Extended version available as the technical report BRICS RS-99-17.

[23] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Compu-*

tation, 12(4), 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.

[24] Robert Glück and Neil D. Jones. Automatic program specialization by partial evaluation: an introduction. In W. Mackens and S.M. Rump, editors, *Software Engineering in Scientific Computing*, pages 70–77. Vieweg, 1996.

[25] Robert Harper. Proof-directed debugging. *Journal of Functional Programming*. To appear.

[26] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, pages 507–541, 1997. Extended version available as the technical report BRICS RS-96-34.

[27] John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, Copenhagen, Denmark, July 1998. Springer-Verlag.

[28] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

[29] Neil D. Jones, editor. *Special issue on Partial Evaluation*, Journal of Functional Programming, Vol. 3, Part 3, July 1993.

[30] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–504, 1996.

[31] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`

[32] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available online at `http://www.wkap.nl/sampletoc.htm?1388-3690+11+1+1998`

[33] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms – Third Edition*. Addison-Wesley, 1988.

[34] Chin-Soon Lee. Partial evaluation of the Euclidean algorithm, revisited. *Higher-Order and Symbolic Computation*, 12(2):203–212, 1999.

[35] Torben Æ. Mogensen and Peter Sestoft. Partial evaluation. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. Marcel Dekker, 270 Madison Avenue, New York, New York 10016, 1997.

[36] Gilles Muller, Calton Pu, and Charles Consel. Specialization. In J. Urban and P. Dasgupta, editors, *Encyclopedia of Distributed Computing '99*. Kluwer Academic Publisher, 1999.

[37] Peter Sestoft and Harald Søndergaard, editors. *Special issue on partial evaluation*, Lisp and Symbolic Computation, Vol. 8, No. 3, 1993.

A Some residual-code constructors

A.1 Section 2.2.1

```
(define mk-zero?
  (lambda (n)
    (if (number? n)
        (zero? n)
        `(zero? ,n))))

(define mk-if
  (lambda (test then else)
    (if (boolean? test)
        (if test then else)
        `(if ,test ,then ,else))))
```

```
(define mk-error
  (lambda (f s . vs)
    `(error ',f ,s ,@vs)))


(define mk-remainder
  (lambda (i j)
    (if (and (number? i)
             (number? j))
        (remainder i j)
        `(remainder ,i ,j))))
```

Question A.1: Why is this definition of `mk-if` unsatisfying? □

A.2 Section 2.2.2

```
(define mk-lambda
  (lambda (stub f)
    (let ([x (gensym! stub)])
      `(lambda (,x)
         ,(f x)))))
```

```
(define mk-lambda2
  (lambda (stub1 stub2 f)
    (let* ([x1 (gensym! stub1)]
           [x2 (gensym! stub2)])
      `(lambda (,x1 ,x2)
         ,(f x1 x2)))))
```

A.3 Section 2.2.3

```
(define mk-let
  (lambda (stub e f)
    (let ([x (gensym! stub)])
      `(let ([,x ,e])
         ,(f x)))))
```

```
(define mk-letrec
  (lambda (stub header body)
    (let ([x (gensym! stub)])
      `(letrec ([,x ,(header x)])
         ,(body x)))))
```

A.4 Section 2.2.4

```
(define mk-let*
  (lambda (stub e f)
    (let* ([x (gensym! stub)]
           [body (f x)])
      (cond
        [(equal? x body)
         e]
        [(and (pair? body) (equal? (car body) 'let*))
         `(let* ([,x ,e] ,@(cadr body)) ,(caddr body))]
        [else
         `(let* ([,x ,e])
            ,body)]))))

(define mk-let mk-let*)
```

Exercise A.2: Motivate the clause `(equal? x body)` in the definition of `mk-let*`. □

Exercise A.3: What happens if the second argument of `mk-let*` is atomic, i.e., a symbol, a number, a string, or a boolean? Modify `mk-let*` to cater for this case. □

B An alternative to continuations

In several exercises, we have hinted at continuations for constructing flat residual let-expressions. Alternatively, one could modify `mk-let*` to exploit the following equation [26]:

```
(let* ([x (let* ([x1 e1] [x2 e2] ...) e)]) body)
= (let* ([x1 e1] [x2 e2] ... [x e]) body)
```

(This equation is valid if there is no name clash, which is the case here because of gensym.)

Question B.1: What else should be modified in the toolkit to enable this alternative version of `mk-let*`? □