More about Formatting

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

December 1993

Abstract

This is an extension of the "format" example, in our POPL tutorial [2]. The description is polished, an analysis is added, and a stand-alone formatter is presented.

1 Printing formatted text

We consider a formatting procedure, as can be found in most programming languages (*e.g.*, `format` in Lisp and `printf` in C). Figure 1 displays our formatting procedure. It is written in Scheme [1].

- Given a channel, a control string, and a list of values, this procedure interprets the control string to determine how to format the list of values, and outputs text in the channel.
- For conciseness, only three formatting directives are treated: `~N`, `~S` and `~%`. The first directive specifies that the corresponding element in the list of values must be output as a number. The second specifies that it must be output as a string. The third directive specifies an end-of-line character. Any other character is output verbatim.
- For simplicity, we assume that the control string matches the list of values.

2 Partial evaluation of the formatting procedure

In practice, and most of the time, `format` is called with a constant control string. This situation is ideal for partial evaluation, particularly when `format` is repeatedly called with the same constant control string. We can interpret the control string only once by specializing `format` with respect to this control string. The specialized procedure takes a channel and a list of values and returns an updated channel. It is built as a dedicated combination of printing operations.

Figure 2 presents a version of `format` specialized with respect to the control string `"~N is not ~S~%"`, where the interpretive overhead of `format` has been entirely removed. The

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

```
;;; format: Port * ControlString * List(Value) -> Port
(define format
  (lambda (port control-string values)
    (let ([end (string-length control-string)])
      (letrec ([traverse
                (lambda (port offset values)
                  (if (= offset end)
                      port
                      (case (string-ref control-string offset)
                        [(#\~)
                         (case (string-ref control-string (+ offset 1))
                           [(#\N)
                            (traverse (write-number port (car values))
                                      (+ offset 2)
                                      (cdr values))]
                           [(#\S)
                            (traverse (write-string port (car values))
                                      (+ offset 2)
                                      (cdr values))]
                           [(#\%)
                            (traverse (write-newline port)
                                      (+ offset 2)
                                      values)]
                           [else
                            (error 'format
                                   "illegal control string: ~S" control-string)])]
                        [else (let ([new-offset
                                     (letrec ([upto-tilde
                                               (lambda (new)
                                                 (cond
                                                   [(= new end)
                                                    new]
                                                   [(equal? (string-ref control-string
                                                                        new)
                                                            #\~)
                                                    new]
                                                   [else
                                                    (upto-tilde (+ new 1))]))])
                                       (upto-tilde offset))])
                                (traverse (write-string port (substring control-string
                                                                        offset
                                                                        new-offset))
                                          new-offset
                                          values))]))))])
        (traverse port 0 values)))))

;;; given write-string:    Port * String -> Port
;;;       write-number:    Port * Number -> Port
;;;       write-newline:   Port          -> Port
```

Figure 1: A formatting procedure

```
;;; for any port p and list of two values vs,
;;; (format.1 p vs) = (format p "~N is not ~S~%" vs)

;;; format.1: Port * List(Value) -> Port
(define format.1
  (lambda (port values)
    (write-newline
      (write-string
        (write-string
          (write-number port (car values))
          " is not ")
        (car (cdr values))))))
```

Figure 2: Specialized version of Figure 1

partial evaluator has performed all the operations manipulating the control string. No references to the control string are left in the residual program. The specialized procedure only consists of operations manipulating the unknown arguments, *i.e.*, the channel and the list of values to be formatted.

This example illustrates the essential purpose of partial evaluation: eliminating interpretive overhead — here the interpretation of the control string.

In the two following sections, we analyze the formatting example further. First, how can we assess which interpretive overhead will partial evaluation remove? Second, what exactly does a partial evaluator to remove the interpretive overhead? The answer to the first question is provided by analyzing the binding times of the source formatting procedure. We answer to the second question by deriving a formatter from the source procedure and the binding-time information.

3 Binding-time analysis of the formatting procedure

Essentially, the formatting procedure traverses the control string. This traversal is carried out by the tail-recursive procedure `traverse`.

The control string is static, and this has the following consequences:

- The application of `string-length` can be performed statically. It yields a static number. Therefore the identifier `end` denotes a static value.
- Procedure `traverse` is first called with a dynamic argument (the port), a static argument (the number 0), and a dynamic argument (the list of values). In all the recursive calls to `traverse`, the second argument remains static:
  - If the current character is a control character, the offset is incremented by 2 — a static operation.
  - Otherwise, a new offset is computed with procedure `upto-tilde`. This procedure is applied to the current offset, which is static, uses the free variable `end`, which is static, and traverses the control string, which is also static. Therefore it yields a static result.

The else branch of the case expression is also static — given an illegal control string, we expect the partial evaluator to raise the corresponding error, statically.

- Finally, the two tests in procedure `traverse` are static: the first one depends on the offset and on `end`, which are static, and the second depends on the control string and the offset, which are static.

The port and the list of values are dynamic. Thus all the operations involving them are dynamic as well.

Let us summarize our analysis by displaying the *binding-time signatures* of all the user-defined procedures involved: their type, annotated with binding-time information.

$$\begin{aligned} \texttt{format} &: [\underline{\texttt{Port}} \times \overline{\texttt{ControlString}} \times \underline{\texttt{List(Value)}}] \Rightarrow \underline{\texttt{Port}} \\ \texttt{traverse} &: [\underline{\texttt{Port}} \times \overline{\texttt{Integer}} \times \underline{\texttt{List(Value)}}] \Rightarrow \underline{\texttt{Port}} \\ \texttt{new-offset} &: \overline{\texttt{Integer}} \Rightarrow \overline{\texttt{Integer}} \end{aligned}$$

As for the predefined procedures (`string-ref`, `write-number`, etc.), if any of their argument is dynamic, their result is dynamic.

In the next section, we use this global binding-time information to direct a series of local transformations.

4 Specialization of the formatting procedure

The strategy is to compute all that has been classified as static, and to reconstruct everything else. For example, we compute the expression `(string-length control-string)` and we reconstruct the expression `(write-newline port)`. We also reconstruct a mixed term such as `(write-string port (car values))`, inserting the value of `(car value)` in the residual program. More globally, since the induction variables of `traverse` and of `upto-tilde` are static, we unfold all the calls to these two procedures.

Finally, we construct a residual procedure expecting the port and the list of values.

Next section substantiates the specialization by constructing a formatter, based on the binding-time information.

5 Construction of a formatter

Our formatter expects a control string and constructs a specialized program. We write it in Scheme.

Scheme, as a Lisp-like language, makes it easy to treat programs as data: one represents them as lists. Further, list constructions can be written very concisely using quasiquote (backquote) and unquote (comma). Here is an example.

The following procedure returns the representation of a program as a list:

```
(define adder
  (lambda (x)
    (list 'lambda (list 'y) (list '+ 'y (* x 10)))))

;;; (adder 2) --> (lambda (y) (+ y 20))
```

It can be written more concisely with quasiquote and unquote:

```
(define adder
  (lambda (x)
    `(lambda (y) (+ y ,(* x 10)))))
```

The quasiquote-unquote notation is exactly what we need here: we are going to construct the formatter by sticking a few quasiquotes and unquotes in the text of `format`, guided by binding-time information.

We want our formatter to construct the specialized formatting procedure. This procedure will receive a control string and construct a residual lambda-abstraction:

```
(define formatter
  (lambda (control-string)
    `(lambda (port values)
       ,...)))
```

Then essentially we compute all the expressions that were classified static, and we reconstruct all the other parts.

The complete formatter is displayed in Figure 3. Applying procedure `formatter` to the control string `"~N is not ~S~%"` yields the lambda-abstraction of Figure 2.

6 A more realistic formatter

The formatter from Figure 3 is a bit naïve in that it allows dynamic expressions to be duplicated. For example, applying it to a control string with more than two control characters will cause multiple occurrences of the expression `(cdr values)`. If the string is `"~N~N~N~N"`, the specialized program looks as follows.

```
(lambda (port values)
   (write-number
      (write-number
         (write-number
            (write-number
               port
               (car values))
            (car (cdr values)))
         (car (cdr (cdr values))))
      (car (cdr (cdr (cdr values))))))
```

Against this backdrop, one usually inserts let expressions, which forces one to use a generator of new names, as illustrated below.

```
(lambda (port0 values1)
   (let ([values2 (cdr values1)])
      (let ([values3 (cdr values2)])
         (let ([values4 (cdr values3)])
            (let ([values5 (cdr values4)])
               (write-number
                  (write-number
                     (write-number
                        (write-number
```

```
;;; formatter: ControlString -> ResidualProgram
;;; where ResidualProgram: Port * List(Value) -> Port
(define formatter
  (lambda (control-string)
    `(lambda (port values)
       ,(let ([end (string-length control-string)])
          (letrec ([traverse
                    (lambda (port offset values)
                      (if (= offset end)
                          port
                          (case (string-ref control-string offset)
                            [(#\~)
                             (case (string-ref control-string (+ offset 1))
                               [(#\N)
                                (traverse `(write-number ,port (car ,values))
                                          (+ offset 2)
                                          `(cdr ,values))]
                               [(#\S)
                                (traverse `(write-string ,port (car ,values))
                                          (+ offset 2)
                                          `(cdr ,values))]
                               [(#\%)
                                (traverse `(write-newline ,port)
                                          (+ offset 2)
                                          values)]
                               [else
                                (error 'formatter
                                       "illegal control string: ~S" control-string)])]
                            [else (let ([new-offset
                                         (letrec ([upto-tilde
                                                   (lambda (new)
                                                     (cond
                                                       [(= new end)
                                                        new]
                                                       [(equal? (string-ref control-string
                                                                            new)
                                                                #\~)
                                                        new]
                                                       [else
                                                        (upto-tilde (+ new 1))]))])
                                           (upto-tilde offset))])
                                    (traverse `(write-string ,port
                                                             ,(substring control-string
                                                                         offset
                                                                         new-offset))
                                              new-offset
                                              values))]))))])
            (traverse 'port 0 'values))))))
```

Figure 3: A specializer for Figure 1

```
                    port0
                    (car values1))
                (car values2))
            (car values3))
        (car values4)))))))
```

This residual program is an instance of `format`, specialized with respect to the control string `"~N~N~N~N"`. Every time a recursive call to `traverse` has been unfolded, a let-expression has been inserted. In other words, the clause

```
[(#\N)
 (traverse `(write-number ,port (car ,values))
           (+ offset 2)
           `(cdr ,values))]
```

in Figure 3 has been replaced by the clause

```
[(#\N)
 (let ([rest (gensym! "values")])
  `(let ([,rest (cdr ,values)])
    ,(traverse `(write-number ,port (car ,values))
               (+ offset 2)
               rest)))]
```

and similarly in the clause for `(#\S)`.

Writing a formatter this way is a straightforward task — since the global binding-time information guides each local transformation — but it can be a bit frustrating at times: there seems to be always one more thing to optimize. For example, in the last residual program, there is no point for the inner let-expression. It could be avoided by spotting the last control character and not generating any let-expression, for example. Alternatively the residual program could be post-processed.

Also, our source procedure is purposefully simple — we assume that the list of values matches the control string, *i.e.*, that `format` does not run out of values nor has too many values. Often, the length of the list is known as well as the control string, so this match could be checked by the partial evaluator en passant.

In any case, we would like to conclude that while writing a formatter is a tedious task, it is a straightforward one — best left for an automatic tool such as a partial evaluator.

7 Related work

The formatter can be seen as a "backquote interpreter" *à la* Friedman, Wand, and Haynes [3]. The essential point here is that we derive it explicitly from binding-time information.

The formatter could be also obtained by self-application, *i.e.*, by specializing the partial evaluator with respect to the source formatting procedure.[1] In fact, this is what we have done by hand, using quasiquotes and unquotes for readability.

Writing programs that construct other programs is however not an exclusivity of Lisp or Scheme, and short of quasiquotes and unquotes, other data-type constructors are necessary. Two schools are currently competing in that direction, in the area of macros: Kolbecker

[1] In partial-evaluation parlance, the formatter is called a "generating extension" — a term due to the late Academician Andrei Ershov.

proposes a pattern-directed approach in Scheme [4] and Weise proposes a language of syntax constructors in C [**?**].

8 Summary

A procedure formatting output with a control string can be specialized with respect to this string. A successful partial evaluator can remove the entire interpretive overhead of this string. A simple partial-evaluation strategy works in two steps: (1) binding-time analysis and (2) specialization. The binding-time analysis provides global information about the parts of the source program that solely depend on the static input. This information can direct a series of local program transformations where static expressions are reduced and dynamic ones are reconstructed. It can also be used to derive a specializer dedicated to the source program.

References

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

[2] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[3] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1991.

[4] E. E. Kolhlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, Bloomington, Indiana, 1986.