Towards Leakage Containment

Julia L. Lawall and Daniel P. Friedman[*]
Indiana University
Computer Science Department
Bloomington, IN 47405

Abstract

Functional programs are organized into procedures, each encapsulating a specific task. A procedure should not cause its callers to repeat its work. This forced repetition of work we call *leakage*. In this paper we describe several common instances of leakage, and show how they can be eliminated using an extension of continuation-passing style.

1 Introduction

A goal of programming is to divide a complex task into simpler parts. In a well organized program each of these simpler tasks is represented by a procedure. Each procedure should perform a distinct action. Its user should not have to be conscious about the details of its implementation and should not have to undo or repeat work performed by it. This excess interaction between procedures is called *leakage*. When leakage is eliminated, the clarity and correctness of programs is enhanced. Our goal in this paper is to show how some leakage can be removed by using a generalization of continuation-passing style. We emphasize the development of a clear programming style rather than efficiency.

Leakage is a problem in the interface between procedures. One example is when a procedure needs an accumulator as an argument. In this case the user is made aware of an implementation decision of the procedure, that it is written in accumulator style. In this case since the caller has no need to interact with accumulator it can be eliminated from the interface by using a help procedure. Another Leakage problem arises when the single return-value channel from a procedure to its caller is insufficient. Using this channel a procedure is only able to return the values it sees as appropriate in the format it chooses. But since the procedure is not the one that will use the information, this rigidity causes problems. The procedure must create a single structure that the caller must decode. This decoding is especially undesirable if the called procedure has already performed a similar task to generate the result. In this case both the caller and the procedure need to use the value, so we cannot eliminate this form of leakage as simply as we could eliminate the leakage caused by the accumulator. In this paper we show how to eliminate this form of leakage. Our approach shifts the emphasis from what the procedure sees as appropriate to what the caller actually wants. When the caller controls the content and format of the information returned, it is less likely to have to

[*]Julia Lawall (jll@iuvax.cs.indiana.edu) is supported by a NSF Graduate Fellowship and CCR 90-00597. Daniel Friedman (dfried@iuvax.cs.indiana.edu) by NSF grants CCR 87-02117, CCR 89-01919, and CCR 90-00597.

repeat the work of the called procedure. Our extension of continuation-passing style enables this shift to a caller-directed method of return values.

Continuation-passing style is a style of programming in which all procedures calls are in tail position. Rather than relying on the language implementation to remember what to do with the result, an abstraction of the context of the call is explicitly passed to the procedure. The called procedure is then responsible for passing its return value to this procedure. Algorithms exist to convert any sequential program to continuation-passing style without changing the program's extensional behavior [10].

There are two characteristics that define continuation-passing style. Because in most applicative languages procedures may only return one value, continuations always take just one argument. Continuations are always invoked in tail position. Otherwise the current value would not be the procedure's final answer, and thus it would be an inappropriate argument to the continuation.

Continuations are, however, just ordinary procedures and thus only continuation-passing style restricts how we build and use them. In this paper we remove these restrictions. The resulting programs come closer to this ideal of programs free of extraneous computation. We refer to this style as *continuation-constructing style*, to emphasize that care goes into the construction of a continuation as well as its use.

The next two sections each deal with a typical example of leakage. The fourth addresses leakage in a larger example and solves a problem arising from the naive application of the techniques developed in the previous two sections. The procedures used have been chosen because they best illustrate our discussion of the flow of information, although they may not use the best algorithms. In each case we develop a technique for using constructed continuations to close the action of a procedure within the procedure itself.

2 Flags

When a procedure has finished, it has to communicate to its caller a representation of what it did. If the caller is only interested in whether the procedure was able to complete its task successfully, an appropriate boolean can be used to communicate success or failure. The caller then has to test the result to determine what happened. This test is a repetition of work that was done inside the procedure. The procedure knew whether it succeeded, so the caller should not have to figure out that information again. Here the test seems harmless enough because it only involves a simple examination of a boolean value. Sometimes, however, more than two conditions need to be signalled. Arbitrary indicators can be chosen to represent each condition. Flags of any sort, however, are open to misinterpretation because they have no intrinsic meaning.

A more serious problem can arise when it is necessary to return an answer or an indication of failure. An example is the typical program for finding the value associated with a key in a table. Any value that the lookup procedure might return to indicate that the key is not in the table might also be the value associated with the key. One solution is to return an element of a union type, as might be done in a typed language such as ML [8]. Valid values are injected into one element of the union and the failure indicator is injected into another. This solution causes leakage in two ways. Not only is it necessary to test for the procedure's failure both inside and outside, but the injection tag, added to the returned value inside the procedure, then has to be removed by the caller before the value can be used.

Rather than returning separate flags for each condition and then decoding those flags to determine what to do next, we can construct and pass continuations describing what to do in each case. This solution is especially appealing when either a value or a failure indicator is returned. Now there is no problem about what value to choose for the flag, because no flag is needed. Instead what to do next is determined entirely by which continuation is invoked. Similarly once the called procedure determines it has failed it is not necessary for anyone else to test for a failure condition. A table lookup program written in this style is shown below. Because the caller wants to know the value, not just whether one was found, the value is an argument to the *success continuation* [13], `k`. Since in the failure case there is no information that needs to be communicated, the *failure continuation*, `q`, takes no arguments.

```
(define lookup
  (lambda (key table k q)
    (cond
      [(null? table) (q)]
      [(eq? key (key-of (first table))) (k (value-of (first table)))]
      [else (lookup key (rest table) k q)])))
```

A similar approach is to use exception handlers, or more generally Scheme's `call/cc`, and flag a different exception for each case. The exception handling code in the caller may, however, be convoluted.

3 Multiple Return Values

An obvious way the table lookup program violates continuation-passing style is that there are two continuations rather than only one. Another difference is more subtle. In continuation-passing style it only makes sense that every continuation should take exactly one argument. In the lookup program, however, we determined the number of arguments for each continuation by considering just how much information is actually needed by the continuation. Thus we end up with one continuation that has no arguments. The other indeed has exactly one, but that is just because the only piece of information needed by the continuation is the value associated with the key. The emphasis is, thus, not on returning a single value, but on passing to the rest of the computation the values it *needs*. In this section we discuss another example where the continuation needs a number of arguments other than one.

Some languages support multiple return values [14]. A possible solution in other functional languages when multiple return values are needed, is to package them into a list. The list is then returned to the caller who is responsible for decoding it to find the intended values. The list itself, however, conveys no information. Instead the receiver must be knowledgeable about what the called procedure intended. The following program, which determines the minimum and maximum elements of a list, is an example of this technique:

```
(define minmax
  (lambda (l)
    (if (null? (cdr l))
        (list (car l) (car l))
        (let ([mm (minmax (cdr l))])
          (let ([min (car mm)]
                [max (cadr mm)])
            (cond
              [(< (car l) min) (list (car l) max)]
              [(> (car l) max) (list min (car l))]
              [else mm]))))))
```

This program causes leakage. As is illustrated by the recursive call, the caller of minmax immediately has to undo the work minmax did to construct the returned data structure. It is also easy to forget which element of the list represents which value. Type constructors of a language like ML can partially alleviate this problem [8]. Such languages provide a special pattern matching syntax to extract the components of these structures in a concise way. The type must, however, be defined in the scope of all the procedures that might call the associated structure-returning procedure. Usually it must be defined globally. When there are many procedures that return structures this increases the possibility of name conflicts. Because the definitions of these procedures are near neither the call nor the return, the result can be hard to understand. Even when these procedures are used, structures are constructed only to be immediately decomposed by the caller.

Here again we can use our idea of passing to a constructed continuation all the values that it needs. Since the minmax program needs to return two values, it can take a constructed continuation of two arguments. Since the values are passed as individual arguments, the result does not need to be destructured. The parameters of the continuation can be named in a way that suggests their role, reducing the chance of error. Furthermore, there is no need to introduce a large number of global names. The minmax program rewritten in this style follows:

```
(define minmax
  (lambda (l k)
    (if (null? (cdr l))
        (k (car l) (car l))
        (minmax (cdr l)
          (lambda (min max)
            (cond
              [(< (car l) min) (k (car l) max)]
              [(> (car l) max) (k min (car l))]
              [else (k min max)]))))))
```

In the consequent a call to the continuation has replaced the call to list. In the other branch of the if expression there is a recursive call. Clearly its continuation must be a procedure of two arguments. In the original program the return value is first destructured and then tested in the cond statement. Now the destructuring is unnecessary. Instead only the cond is needed. In the original program in each line of the cond, list is called to return the two new answers. Just as in the consequent, we must now use k to return an answer.

In this example we turn to continuations, not because we have more than one return path, but because we have a non-standard number of values to return. Minmax is forced to be in something like continuation-passing style because its actual return value is determined completely by the continuation provided by the caller. Instead it can only reasonably use the values passed to the continuation. It is not exactly in continuation-passing style, however, because the continuation takes more than one argument.

We have identified two situations that may cause leakage - the need to signal several conditions and the need to return several values. The problem of signalling several conditions can be solved by passing in the corresponding number of continuations. The problem of returning several values can be solved by passing in a continuation of several arguments. Clearly it is possible for both of these problems to occur in a single procedure. In the next section we give such an example and address a problem that occurs in eliminating its leakage.

4 Multiple Continuations

Lookup in section 2 returns its answer to one of a number of continuations. Because it is a tail recursive program, the continuations passed in by the caller do not change on each recursive call. The original minmax is not tail-recursive and thus its continuation grows on each iteration. Because there is only one continuation this does not seem unreasonable. This section discusses a difficulty that can arise when multiple continuations are passed to a non-tail-recursive program in order to solve the leakage problems described above. To illustrate this difficulty we consider a third example, an extension of Scheme's lambda facility.

In Scheme, procedures created using lambda take either a fixed or a minimum number of arguments [11]. In a curried language, like Miranda [17], a function can be applied to any number of arguments. If not enough arguments are present a new function is built. If too many are present the body of the function is expected to return a function that will accept the remaining arguments. Of course, if exactly the right number of arguments are present the body is evaluated normally. This feature has been advocated as a way to avoid using the heap in some cases when returning higher order functions [7]. We examine the leakage problems that arise in adding this feature to a Scheme interpreter. We assume the existence of an interpreter and concentrate on two procedures, make-closure and extend. We have arbitrarily chosen to use flat association lists to represent the environment. The code to implement the traditional version of lambda is shown below.

```
(define make-closure
  (lambda (formals body env)
    (lambda (args)
      (Meval body (extend env formals args)))))

(define extend
  (lambda (env formals args)
    (cond
      [(null? formals) env]
      [else
        (let ([new (cons (car formals) (car args))])
          (cons new (extend env (cdr formals) (cdr args))))])))
```

Although we are extending an interpreter, the extension is *not* a solution to the leakage problem. Instead it serves as yet another example of how leakage can occur, and be eliminated.

These procedures assume that the number of formals is the same as the number of arguments. Thus make-closure can assume that extend returns a complete environment. Under the new definition of lambda this is no longer the case. Make-closure must know how the length of the list of formals compares to the length of the list of arguments. One approach would be for it to compute directly the length of each list. Extend, however, as we have written it, traverses the list just as length would. When it reaches the end of one list but not the other it knows which is longer. Because make-closure is not interested in the exact lengths, the calls to length are not needed. Having them would be a form of leakage. Since extend has the information make-closure needs, it should return, not just the new environment, but also an indication of which list is longer and what the leftovers are. Rather than return a list of several values, we recall minmax and construct a continuation in make-closure that it passes to extend. This is an example of the caller directing what values should be returned. The direction is made explicit by the continuation. A preliminary implementation of the new version of lambda appears below.

```
(define make-closure
  (lambda (formals body env)
    (lambda (args)
      (extend env formals args
        (lambda (new-env extra-formals extra-args)
          (cond
            [(and (null? extra-formals) (null? extra-args))
             (Meval body new-env)]
            [(null? extra-args)
             (make-closure extra-formals body new-env)]
            [(null? extra-formals)
             ((Meval body new-env) extra-args)]))))))

(define extend
  (lambda (env formals args k)
    (cond
      [(and (null? formals) (null? args)) (k env '() '())]
      [(null? args) (k env formals '())]
      [(null? formals) (k env '() args)]
      [else
        (let ([new (cons (car formals) (car args))])
          (extend env (cdr formals) (cdr args)
            (lambda (newenv extra-formals extra-args)
              (k (cons new newenv)
                 extra-formals extra-args))))])))
```

There is some redundancy in extend, introduced to emphasize what will be done with the values passed to the continuation. Clearly, the first three tests could be condensed into one. The empty list is used in each case to emphasize that the null value will be tested for.

Comparing make-closure and extend we see that there is a direct correspondence between the tests in the two procedures. The presence of this correspondence again indicates leakage. The variables extra-formals and extra-args each both encode the relationship between the number of formals and the number of arguments and serve as structures to hold the values left over. The dual role played by these two variables makes the tests doubly undesirable. As in lookup we can pass in a continuation for each condition rather than return a coded value. Each continuation need only take the arguments it uses. For example, the action when there are no extra formals and no extra arguments only uses the new environment, so just that value needs to be passed to it. Make-closure is rewritten in this new style below.

```
(define make-closure
  (lambda (formals body env)
    (lambda (args)
      (extend env formals args
        (lambda (new-env) (Meval body new-env))
        (lambda (new-env extra-formals)
          (make-closure extra-formals body new-env))
        (lambda (new-env extra-args)
          ((Meval body new-env) extra-args))))))
```

We now turn to the revision of extend. It must now handle three continuations instead of one. Let us call the continuations nk for the normal case (when there is an equal number of formals and arguments), xf for the case when there are extra formals, and xa for when there are extra arguments. Clearly in the first cond line nk should be applied to env. Similarly xf should be applied in the second to the environment and the current list of formals, and xa should be applied in the third to the environment and the current list of arguments. But just changing these lines is not enough. The recursive call in the last line is also a call to extend, and thus it needs three continuations. As before the only activity of these continuations is to extend the new environment. Any other arguments are just passed along. Extend rewritten accordingly appears below.

```
(define extend
  (lambda (env formals args nk xf xa)
    (cond
      [(and (null? formals) (null? args)) (nk env)]
      [(null? args) (xf env formals)]
      [(null? formals) (xa env args)]
      [else
        (let ([new (cons (car formals) (car args))])
          (extend env (cdr formals) (cdr args)
            (lambda (newenv)
              (nk (cons new newenv)))
            (lambda (newenv extra-formals)
              (xf (cons new newenv) extra-formals))
            (lambda (newenv extra-args)
              (xa (cons new newenv) extra-args))))])))
```

Here we see a difficulty with our technique for leakage elimination. On each recursive call we extend three continuations, and all are extended in the same way. We would rather do the computation of extend as before and then invoke one of nk, xf and xa at the very end. The problem is that what appears to be the end, the last test, is not the end of the computation at all. The continuation of each recursive call has to add a new association onto the front of the environment. The original `nk`, `xf`, and `xa` want the complete environment, not the one available at the time of the last test. This is the typical case when a non-tail-recursive program needs to signal a condition to its caller. Since the result of the recursive call can only be accessed in the continuation, we seem forced to copy the accumulation of this context in every continuation that might be invoked. In solving the leakage problem in a program that is not tail-recursive, we seem to have introduced another difficulty.

In `extend` the continuation of each recursive call only needs the value of the environment. If there is another value passed to it, as for `xf` and `xa`, that value is just passed on to the current continuation. But if we were to just return the environment, there would be no way to return the other necessary values "later." We need a way to remember the other values that should be returned. One possibility is to have a single continuation that takes care of adding to the environment. Besides the environment this continuation takes another argument to be invoked by the initial continuation. This second argument, which may be thought of as a continuation to the continuation, takes the final environment as an argument and then passes to one of the continuations provided by `make-closure` the values it expects. The new version of `extend` appears below:

```
(define extend
  (lambda (env formals args nk xf xa)
    (letrec
      ([loop
         (lambda (formals args k)
           (cond
             [(and (null? formals) (null? args)) (k env nk)]
             [(null? args)
              (k env (lambda (newenv) (xf newenv formals)))]
             [(null? formals)
              (k env (lambda (newenv) (xa newenv args)))]
             [else
               (let ([new (cons (car formals) (car args))])
                 (loop (cdr formals) (cdr args)
                   (lambda (env fn)
                     (k (cons new env) fn))))]))])
      (loop formals args (lambda (env fn) (fn env))))))
```

Because the continuation to `k` takes only one argument and is always applied to the final environment we can do the inverse of the continuation passing style transformation of the calls to `k`, making them non-tail-recursive. The code is show below. In leakage elimination it is often the case that the continuations do not take exactly one argument so this technique cannot generally be applied. When it can be used, however, the result can be rewritten in a more direct style using either Felleisen's $\mathcal{F}$ and `prompt` operators [4], or Danvy and Filinski's `shift` and `reset` [2].

```
(define extend
  (lambda (env formals args nk xf xa)
    (letrec
      ([loop
         (lambda (formals args k)
           (cond
             [(and (null? formals) (null? args))
              (nk (k env))]
             [(null? args) (xf (k env) formals)]
             [(null? formals) (xa (k env) args)]
             [else
               (let ([new (cons (car formals) (car args))])
                 (loop (cdr formals) (cdr args)
                   (lambda (env)
                     (k (cons new env)))))]))])
      (loop formals args (lambda (x) x)))))
```

5 Comparison with Related Work

The leakage problem has been addressed in a limited way before. One approach is to use a strongly typed language. In such a language the type of a procedure limits the range of values it may return. Values that look the same may be distinguished by union type tags. These features of a typed language lessen the chance of error, but do not eliminate the leakage problem.

A number of researchers have addressed the creation of unnecessary data structures within procedures. Wadler describes a compilation scheme that automatically converts programs that use many intermediate lists into a machine code that is more efficient [18]. The emphasis there is on allowing the programmer to write inefficient looking code, whereas ours is on providing him with a motivation to write better programs. Single threading analysis is another way of eliminating the unnecessary creation of internal data structures. It shows where assignments can be introduced [12]. Danvy explores the use of continuations to solve the intermediate list problem [1]. He further exploits the technique of using continuations in non-tail position. The emphasis of these approaches is different from ours. Our primary concern is with the interface between procedures, whereas they are interested in the flow of data within a procedure.

Dybvig and Hieb consider the creation of unnecessary data structures in the implementation of a `lambda` taking a variable number of arguments [3]. They allow a set of argument patterns with a different action for each case. These argument patterns and associated actions are similar to the continuations we pass in when multiple conditions need to be signalled. Their approach addresses a specific design problem, whereas ours is a general programming technique.

Our use of continuations in the interface between a procedure and its caller is related to Filinski's notion of a symmetric language [5]. In a symmetric language a procedure's value argument and continuation argument have equal importance and are equally accessible. Our solution to the leakage problem forces the caller to pass to the procedure not only values, but also continuations.

6 Conclusion

In denotational semantics continuation-passing style has made it possible to express complicated control structures that it would be clumsy to express in direct style [16]. It may be a major undertaking to convert an existing semantics to continuation-passing style [15], but once the conversion has been done the result is more expressive. Similarly the conversion from a direct style of building ordinary programs to the continuation-constructing style we have discussed, may initially seem complicated. Again the result is clearer and more extensible. As shown by the above examples, programs in this style may not be much longer than programs written in a more direct style. While others have addressed the issue of leakage in very specific cases we feel our approach can be applied to a wider range of leakage problems and has a greater influence on program style. The methods described in this paper will not, of course, eliminate leakage in every program. We have only addressed some very common programming constructs. Developing a better understanding of the nature of leakage may help solve these problems.

Acknowledgments

We would like to thank Olivier Danvy, Rhys Price Jones, Richard Salter, and George Springer for their careful reading and most helpful criticisms of this abstract.

References

[1] Danvy, O., Programming with tighter control. Special issue of the BIGRE journal *Putting the Scheme Language to Work*, Brest, France (July 1989) 10-29.

[2] Danvy, O. and Filinski, A., A functional abstraction of typed contexts. DIKU Report 89/12. DIKU Computer Science Department, University of Copenhagen (Aug. 1989).

[3] Dybvig, K., and Hieb R. A variable-arity procedural interface. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, Snowbird, Utah (July 1988), 106-115.

[4] Felleisen, M., The theory and practice of first-class prompts. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, San Diego, California (Jan. 1988) 180-190.

[5] Filinski, A., Declarative continuations and categorical duality. Master's thesis. DIKU, University of Copenhagen, July 1989.

[6] Fischer, M.J., Lambda Calculus Schemata. In *Proceedings of the ACM Conference Proving Assertions about Programs*, *SIGPLAN Notices 7*, 1 and *SIGACT News 14* (Jan. 1972) 104-109.

[7] Georgeff, M.P., A scheme for implementing functional values on a stack machine. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, Pittsburgh, Pennsylvania (August 1982), 188-195.

[8] Harper R., MacQueen D., Milner R., Standard ML. Laboratory for Foundations of Computer Science. University of Edinburgh. ECS-LFCS-86-2. 1986.

[9] Morris, J.H. Jr., A bonus from Van Wijngaarden's device. *Comm. of the ACM 15* (Aug. 1972) 773.

[10] Plotkin, G.D., Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci. 1*, 1975, 129-159.

[11] Rees, J.A., et al. Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices 21* (Dec. 1986).

[12] Schmidt, D.A., Detecting global variables in denotational specifications. *ACM Trans. on Prog. Lang. and Sys. 7* (1985) 299-310.

[13] Schmidt, D.A., *Denotational Semantics.* Allyn and Bacon, Boston, 1986. pp. 202-204.

[14] Steele, G.L. Jr, *Common Lisp: The Language.* Digital Press, 1984.

[15] Stoy, J.E., The congruence of two programming language definitions. *Theoretical Comp. Science 13* (1981) 151-174.

[16] Strachey, C. and Wadsworth, C.P., Continuations - a mathematical semantics for handling full jumps. Technical Monograph PRG-11, Programming Research Group, University of Oxford (1974).

[17] Turner, D.A., Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2nd International Conference on Functional Programming Languages and Computer Architecture LNCS 201*, Nancy, France (Sept. 1985) Springer-Verlag 1985, 1-16.

[18] Wadler, P., Listlessness is better than laziness: lazy evaluation and garbage collection at compile-time. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas (Aug. 1984), 45-52.