Higher Order Imperative Programming[*]

Aaron Kershenbaum [†] David Musser [‡] Alexander Stepanov [§]

Abstract

It is argued that a programming style based on higher order techniques—the use of procedures that have other procedures as arguments and/or results—can be most effectively employed if

- it is driven by abstraction from real algorithms, rather than attempting to work with a fixed set of functional forms; and
- the use of imperative forms and mutative procedures is permitted (even encouraged!), rather than restricting to a purely applicative style.

A sequence of examples is presented illustrating a number of higher order techniques—operators, iterators, accumulation, reduction, parallel reduction—and their effective use in conjunction with mutative procedures. The examples culminate with an interesting family of sorting algorithms, illustrating how higher order techniques can lead naturally to new algorithms.

1 Introduction

The central idea of higher order programming is the use of higher order procedures; i.e., procedures that take other procedures as arguments and/or return procedures as their results. Higher order techniques have been in use to a limited degree in Lisp programming for many years (e.g., `mapcar` and other functions that apply a function given as a parameter

[*]Rensselaer Polytechnic Institute Computer Science Department Technical Report 88-10, April 1988
[†]Computer Science Department, Polytechnic University, Brooklyn NY
[‡]Computer Science Department, Rensselaer Polytechnic Institute, Troy NY
[§]Computer Science Department, Polytechnic University, Brooklyn NY (presently with AT&T Bell Laboratories, Liberty Corner, NJ)

to each element of a given list), and more extensively in APL (e.g., the notion of reduction that produces summation over an array from $+$, product over an array from $\times$, etc.). APL especially promotes the notion that higher order techniques give the programmer significantly greater power, in terms of conciseness of expression, than most other languages.

The functional programming community has also been using higher order techniques (e.g., in languages such as FP [1], ML [5], and Miranda [12]). However, as with APL, there is an widespread emphasis in functional programming on working with a fixed, predefined set of functional forms. There is usually a further restriction to a purely applicative (no side-effects) style of programming. The applicative style is seen as providing the best foundation both for reasoning about programs, and, in the long run, most efficient use of computing resources, since program optimizations for parallel architectures can be more comprehensively applied in the absence of side-effects.

Our approach to higher order programming diverges from these trends in two significant ways. First, our higher order operators are derived from experience with real algorithms rather than by defining an *a priori* universal set of functional forms, as done for example in FP. Thus, we do not attempt to express algorithms in terms of already defined primitives, but instead, we examine real algorithms and derive useful operators from them via the process of algorithmic abstraction.

Second, we recognize that the use of higher order techniques is orthogonal to the question of applicative vs. non-applicative styles. Higher order techniques can in fact be used with great effectiveness with imperative forms and mutative[1] procedures, and hence can be put into much more widespread use *today*, in existing languages, without the need for new language or architectural advances. The usual objections to imperative forms and mutative procedures, that they are less easy to understand or to formally verify than applicative constructs, can be overcome by disciplining their use in higher order programming to a small number of well-behaved cases. The examples in this paper will, we hope, provide some initial evidence for these points.

Even for the long run we question whether a purely applicative style of programming is desirable. This is a controversial subject that we won't dwell on at any length in this paper, but in essense we argue, first, that imperative forms and mutative procedures capture natural aspects of many problems, and, secondly, many efficient algorithms are by nature mutative. The use of higher order techniques with mutative procedures is a highly effective way to express such algorithms; this is a key idea of what we refer to as higher order programming.

[1]The perjorative term "destructive" is frequently used, but "mutative" more closely captures the notion that the *purpose* of a procedure can be to have an effect (not a "side-effect") on its arguments.

2 The Scheme Language

Some languages have better support than others for higher order programming techniques. Most of our work to date has been carried out in Scheme [9], Ada [8], and C++ [11]. We will use Scheme for the examples in this paper, since Scheme supports higher order techniques in a simple, natural way, and since readers with a modest familiarity with Lisp should be able to read the subset of Scheme used in the examples. However, the techniques can be adapted to other languages, and our use of them in Ada will be discussed briefly in the conclusions section.

We will use only the following special forms of Scheme:

`(lambda` ($parameter_1 \ldots\ parameter_n$`)`
  $form_1 \ldots\ form_k$`)`

for creating a procedural object with parameters;

`(let ((`$var_1\ \ vform_1$`)` ... `(`$var_n\ \ vform_n$`))`
  $form_1\ \ldots form_k$`)`

for creating local, initialized variables;

`(if` *condition* $form_1\ \ form_2$`)`

for conditional execution of $form_1$ or $form_2$ based on whether *condition* evaluates to true or false;

`(begin` $form_1\ \ldots form_k$`)`

for sequential execution of $form_1\ \ldots form_k$;

`(set!` *identifier form*`)`

for changing the value bound to a variable;

`(define` *identifier form*`)`

for binding a variable to a value of a form; and

`(define (`*identifier* $parameter_1 \ldots\ parameter_n$`)`
  $form_1 \ldots form_k$`)`

which is shorthand for

```
(define identifier
   (lambda (parameter_1 ... parameter_n)
      form_1 ...form_k))
```

For list-manipulation, Scheme retains the traditional Lisp names `car`, `cdr` and `cons`. Using traditional Lisp terminology, we refer to the halves of a "cons cell," or a "pair," as its "car" and "cdr". For mutating the car of a pair, Scheme provides `set-car!` instead of the more traditional `rplaca`, and, similarly, `set-cdr!` instead of `rplacd`.

The procedures `set-car!` and `set-cdr!` and the special form `set!` illustrate the convention usually followed in Scheme of syntactically distinguishing the names of procedures and special forms that mutate their arguments by ending them with an exclamation point. These forms are executed primarily, or in some cases solely, for their effect rather than for any value they return. We follow this convention in the examples below.

One of the main characteristics of Scheme that makes it well-suited to higher order programming is that it treats procedures as first-class values: they can be assigned to variables, passed as arguments to other procedures, and even returned as values by other procedures. Scheme makes this even more convenient than in other Lisp dialects such as Common Lisp, since in a procedure application,

```
(form_1 form_2 ... form_n)
```

each of the component forms are evaluated in the same way: *form*$_1$ is not given any special treatment. Thus procedures can be passed as parameters and used as values with exactly the same syntax as any other values; the necessity for `funcall` is eliminated.

Although most examples in this paper use lists for data representation, we should point out that this is in no way essential to higher order programming. We have also developed a large number of examples using arrays and other fundamental data structures.

3 Operators

An operator is a procedure which takes one or more other procedures as arguments. Among other benefits, operators allow us to simplify both informal understanding and formal verification of programs by encapsulating iterative constructs. We begin by considering a simple example to illustrate these ideas. As we progress, we will see increasingly more substantial examples of the expressive power of programs written using operators.

The following procedure, `until`, takes as its arguments two procedures `done?` and `compute-next`, each of which takes a single argument. `Done?` is assumed to return true or false, and is treated as a stopping condition for an iteration, namely the computation of `value`, `(compute-next value)`, `(compute-next (compute-next value))`, etc., until the result satisfies `done?`. The result of `until` is a new procedure which takes a single argument, `value`, and performs this iterative computation.

```
(define (until done? compute-next)
  (define (new-procedure value)
    (if (done? value)
        value
        (new-procedure (compute-next value))))
  new-procedure)
```

For example, to compute the smallest power of 3 greater than 1000, we could write:

```
(define (done? x) (> x 1000))
(define (compute-next x) (* x 3))
((until done? compute-next) 3)
```

Note that the call of `until` produces a procedure, and that procedure is what is applied to 3. In writing `until`, we are using the feature of Scheme that allows a procedure definition to contain nested definitions. A value (usually a procedure) thus created may be referenced by its associated identifier only within the outer `define`. Such a value may however be returned as the value of the outer `define`, as is `new-procedure` in `until`. These nested definitions may reference any formal parameters of the outer definition and also any other definitions at the same level (lexical scoping is used in Scheme).

Another example of the use of `until` is the following definition of a square root procedure using a Newton iteration:

```
(define (sqrt x epsilon)
   (define (acceptable? y)
      (< (abs (- x (* y y))) epsilon))
   (define (compute-next-approximation y)
      (* .5 (+ y (/ x y))))
   (define iteration (until acceptable? compute-next-approximation))
   (iteration x))
```

Another important feature of Scheme that we are making use of in defining `new-procedure` within `until` is the fact that Scheme executes tail-recursive procedure calls (those calls of the procedure being defined that are not followed by any computation) without using stack space; they become, in effect, efficient iterative loops. In most other languages, one would program operators like `until` using explicit iteration rather than recursion. As we shall see, however, we can build other iterative operators out of `until`, so that the need for either tail-recursion or explicit iteration is minimal. In general, we believe that explicit iteration and recursion should be avoided in favor of higher order procedures when possible, so that most code is "straight-line."

An alternative, simpler, way to define `until` would be

```
(define (until done? compute-next value)
  (if (done? value)
     value
     (until done? compute-next (compute-next value))))
```

i.e., rather than producing a procedure that operates on a data value to produce another value, it works directly with data values as well as procedural arguments. However, we have generally chosen to write higher order procedures in the form of our previous definition of `until`, so that it is possible to use the definition first to define a new procedure, then separately apply that procedure to data values. This style increases the possibilities for compile time evaluation of higher order procedures and allows our techniques to be applicable to languages that do not treat procedures as first class objects, but do have generics (e.g., Ada) or type polymorphism (e.g., CLU [4]).

4 Iterators

`Until` is an example of one kind of operator called an iterator; i.e., a procedure that allows us to repeat another procedure. Another example of an iterator is the procedure `for-each-pair` given below. `For-each-pair` is the basis for many other important operators.

```
(define (for-each-pair operate-on)
   (define (new-procedure list)
      (if (pair? list)
         (begin (operate-on list)
                (new-procedure (cdr list)))))
   new-procedure)
```

`For-each-pair` takes a procedure, `operate-on`, and produces a procedure which applies `operate-on` repeatedly, each time to a different argument. The first time, the entire list is used as the argument. On each successive application, `cdr` is used to advance by one element until finally only the empty list is left (detected when `pair?` returns false) and `for-each-pair` terminates. As an example,

```
((for-each-pair print) '(1 2 3))
```

will print:

```
(1 2 3)
(2 3)
(3)
```

For-each-pair returns a value, but we are not interested in what it returns. The purpose of invoking for-each-pair is to cause the procedure operate-on to be executed. Thus, we expect operate-on to be mutative or, as in the case of print, to cause some effect such as output.[2]

For-each-pair is really a special case of the type of iteration we have already encapsulated in the until iterator. In order to take advantage of until in defining for-each-pair, and for many other uses, we define the following procedure combinators:

```
(define (compose f g)
   (lambda (x) (f (g x))))

(define (apply-and-return operate-on)
   (lambda (x) (operate-on x) x))
```

There is little to say about these procedures that is not implied by their names; the motivation for them is shown by their usefulness in many subsequent examples. For-each-pair can be now be written:

```
(define (for-each-pair operate-on)
   (until null? (compose cdr (apply-and-return operate-on))))
```

Using for-each-pair we can define the iterator for-each which iteratively invokes a procedure successively using each element of a given list as the procedure's argument. (A slightly different version of for-each is actually a built-in procedure in Scheme.)

```
(define (for-each operate-on)
  (for-each-pair (compose operate-on car)))
```

As with for-each-pair, procedures produced using for-each are executed for their effect rather than to return a value. For example,

```
((for-each print) '(1 2 3))
1
2
3
```

5 Accumulation

We often desire not just to execute a procedure iteratively, but also to accumulate the results computed. We might, for example, wish to find the sum of the values of all the elements in a list. Rather than writing a specific piece of code to do this in each instance, it is possible to write a single operator which will do the job in all cases:

[2]The case in which operate-on mutates the cdr of its argument is discussed in Section 6.

```
(define (accumulate iterate combine)
  (define (new-procedure structure accumulator)
    (define (reset-accumulator! item)
      (set! accumulator (combine item accumulator)))
    ((iterate reset-accumulator!) structure)
    accumulator)
  new-procedure)
```

`Accumulate` takes as input an iterator, `iterate`, and another procedure, `combine`, and produces a new procedure which takes a structure (not necessarily a list) and an initial value. The result returned by the new procedure is result of using `iterate` over `structure` with a procedure that mutates `accumulator` by `combine`-ing it with its argument. `Combine` is assumed to be a binary operation. A simple example is:

```
((accumulate for-each +) '(3 1 -5 8) 0)
```

which returns 7, the sum of the elements in the list. `Accumulate` is a very powerful operator. The structure need not be a list; the only requirement is that the iterator passed as an argument must work on the structure passed. For example:

```
((accumulate for-each-in-file string-append) file "")
```

will accumulate the contents of a file into a character string in memory, given the iterator `for-each-in-file` which reads records from the file, and the built-in procedure `string-append` which concatenates strings. Even more powerful procedures can be built by passing as the second argument a procedure which selectively accumulates based on some property of elements of the structure. Thus, for example,

```
((accumulate for-each
  (lambda (x y)
    (if (positive? x) (+ x y) y)))
 '(3 -8 9 -4)
 0)
```

returns 12, the sum of just the positive numbers in a list.

The examples just shown illustrate that higher order programming is simplified by the dynamic typing of Scheme and other Lisp dialects; strong-typing would require multiple versions of `accumulate` or else would require types also to be passed as parameters and used to declare the operations so that types matched. In Ada, generics provide a partial solution. In general we believe that the advantages of strong-typing, particularly the way it allows many errors to be detected at compile-time, outweigh the inconveniences.

We can also use `accumulate` to produce `reverse` (also a built-in procedure in Scheme), which returns a list whose elements are in the reverse of the order of those in the list passed to it as an argument. We first define `reverse-append`, which takes two lists as arguments and produces a list whose elements are those of the first argument in reverse order followed by those of the second argument in order. In terms of `accumulate`, it is simply:

```
(define reverse-append (accumulate for-each cons))
```

Then

```
(define (reverse list)
   (reverse-append list '()))
```

where `'()` denotes an empty list.

6 Mutative Procedures and Operators

Note that `reverse` produces a new list and does not alter its argument. This is often desirable, but it is wasteful if the original list will no longer be needed, as it involves creating new cons cells, which takes both time and space. Even more important is that in some cases the intent is to rearrange the original list, not to produce a copy. There may be other variables pointing at this list, and it could be important that these all point at cells of the rearranged list.

Thus, it is important also to be able to work with mutative procedures in higher order programming. As an example we now define `reverse!`, a mutative version of `reverse`. The built-in `set-cdr!` procedure of Scheme returns an unspecified value, so we create the following procedure which mutates the cdr of a pair and also returns the new cdr value (all characters on a line following a semicolon are comments):

```
(define (replace-cdr! pair x) ; also known as rplacd in Lisp
   (set-cdr! pair x)
   pair)
```

Let us also define another procedure combinator,

```
(define (apply-and-return-old-cdr operate-on)
   (define (new-procedure pair)
      (let ((old-cdr (cdr pair)))
         (operate-on pair)
         old-cdr))
   new-procedure)
```

with which we can define a useful new iterator analogous to `for-each-pair`:

```
(define (for-each-original-pair operate-on)
   (until null? (apply-and-return-old-cdr operate-on)))
```

This iterator is intended for use with mutative procedures, in particular those which alter structure of the list passed as an argument. Note that `for-each-original-pair` produces a procedure with one argument, a list, which saves the cdr of the current list before calling `operate-on` so that even if `operate-on` mutates the cdr of the list, the original list will still be traversed, whereas `for-each-pair` would traverse the altered list. In different situations, each behavior could be appropriate. When `operate-on` is applicative with respect to its list argument, the procedures produced by `for-each-pair` and `for-each-original-pair` have the same effect.

The definitions of `reverse-append!` and `reverse!` now follow naturally from the above definitions:

```
(define reverse-append!
  (accumulate for-each-original-pair replace-cdr!))

(define (reverse! list)
  (reverse-append! list '()))
```

We thus see that it is possible to create a mutative analog to an applicative procedure. Another important example of this is `map!`. Scheme has a built-in procedure, `map`, which applies a procedure to the elements of a list, forming a list of the results and leaving the original list unaltered. It is often desirable to replace the elements of the original list with the results of the procedure applications. The `map!` procedure we define is analogous to `map`, except that we make it a combinator. First we define two additional combinators for working with mutative procedures:

```
(define (change-car! operate-on)
  (define (new-procedure! pair)
    (set-car! pair (operate-on (car pair))))
  new-procedure!)

(define (change-cdr! operate-on)
  (define (new-procedure! pair)
    (set-cdr! pair (operate-on (cdr pair))))
  new-procedure!)
```

(The second of these will be used later.) Now

```
(define (map! operate-on)
  (apply-and-return (for-each-pair (change-car! operate-on))))
```

In general, we prefer to use `map!` when we can, rather than `map`, as the former is more efficient in that it reuses the original list. The question then arises, when are we free to replace an applicative operator by a mutative one? Clearly, we can when we can prove that the list passed as an argument is not used again.

Another closely related question is how to show that a procedure, `p!`, is the mutative analog of another procedure, `p`. Informally, we say the procedures are analogs if the only difference between them is their effect on the argument; i.e., if:

```
(p x)  ==  (p! (copy x))
```

where `(copy x)` makes a copy of the entire data structure `x` and `==` signifies that not only do both procedures return that same values, but also that they both have the same effects. Note in this regard that since the value returned by `(copy x)` is not bound to any variable, that the fact that `p!` mutates it is not considered a side effect.

We thus see that it is possible to create an applicative analog of a mutative procedure in a mechanical way. Often we need only create the mutative version of a procedure, which we will use in most cases, and pass a copy of the argument in the remaining cases where an applicative procedure is required. If the mutative procedure only mutates a restricted part of the representation of its arguments, then only a partial, more efficient copy operation can be used. For example, if we only had `reverse!` and wanted to create an applicative `reverse` procedure, we could implement it as `(compose reverse! copy-top-level)`, where `copy-top-level` creates new cells only for the top level cells in the representation of its argument, sharing any other cells.[3] Such analysis can be extended to other cases of producing an applicative procedure by composition of a mutative procedure with an applicative procedure that does not share, or shares in only a restricted way, the representation of its result with that of its argument; e.g., `(compose (map! operate-on) reverse)`.

Strictly speaking, however, there is more to the issue of one procedure being the mutative analog of another. By the above definition, a procedure could be its own mutative analog. This is not our intent. When we speak of a mutative procedure, we expect it to actually change its argument. Thus, we require that (unless `p!` is an identity function) that `x` change after `(p! x)` is evaluated. An even stronger requirement would be to require that after executing `(p! x)` that `x` is changed to be the result of `(p! x)`. Unfortunately, it is not always possible to guarantee this and so we rely on the less stringent requirement that `p!` mutate its argument at least for some inputs.

[3] In this case it is undoubtably more efficient to program `reverse` independently, as is done in Section 5. This independence also would permit us to implement `copy-top-level` as `(compose reverse! reverse)`.

7 Reduction

Reduction operators were first introduced in APL [3]. Given a binary operation and a structure, we define the reduction of that operation over the structure as the result of successively combining all elements of the structure with the operation. This notion is very similar to the discussion of `accumulate` above and, indeed, we define `reduce` for lists in terms of `accumulate`:

```
(define (reduce combine default)
  (define (new-procedure list)
    (if (null? list)
      default
      ((accumulate for-each combine) (cdr list)
         (car list))))
   new-procedure)
```

Here `default` is a value that is to be returned if `list` is empty.[4] For example,

```
((reduce * 1) '(1 2 3 4 5))
```

returns 5!.

There are many different possible forms of reduction. The above implementation accumulates the result using the elements of the list from left to right. It is also possible to define a right-to-left reduction:

```
(define (right-reduce! combine default)
   (compose (reduce combine default) reverse!))
```

If the `combine` operation is associative and commutative, the order of reduction does not matter, but in many cases we will want to apply reduction to operations which do not have these properties. Indeed, in some cases, as we will see below, the order of reduction is very important and by choosing different forms, algorithms with different efficiency characteristics are created.

8 Parallel Reduction

Reduction can also be done "in parallel" by combining pairs of elements.[5] We will use parallel reduction to define an interesting class of sorting algorithms in the following section.

[4]Although it would allow a slightly simpler definition of `reduce`, we are *not* assuming that there is an an identity value for `combine` (a value `e` such that `(combine e x) = (combine x e) = x`, since in some of the examples of sorting procedures given later we use a `combine` that has no identity value. Note that when `list` has only one element, `new-procedure` always returns that element.

[5]To the best of our knowledge, the notion of parallel reduction as an operator was introduced in [6].

In order to define parallel reduction, we first define the following combinator which, given a binary procedure `combine`, returns a procedure which replaces the first element of a non-empty list by the result of applying `combine` to the first two elements of the list; the second element is eliminated.

```
(define singleton? (compose null? cdr))
(define (combine-first-two! combine)
  (define (new-procedure! list)
     (if (not (singleton? list))
        (begin
           (set-car! list
              (combine (car list) (car (cdr list))))
           (set-cdr! list (cdr (cdr list))))))
  new-procedure!)
```

For example, `((combine-first-two! +) '(3 4 5 6))` transforms its argument into `(7 5 6)`.

The following combinator yields a procedure that replaces the elements in a non-empty list by the results of applying an operation to each pair of elements:

```
(define (pairwise-combine! combine)
   (apply-and-return (for-each-pair (combine-first-two! combine))))
```

For example,

```
(define pair-min! (pairwise-combine! min))
```

binds `pair-min!` to a mutative procedure which replaces the first and second element of a list by their minimum, and similarly for the third and fourth, the fifth and sixth, etc. Thus

```
(pair-min! '(7 4 2 9 3))
```

returns `(4 2 3)`.

We can now define `parallel-reduce!` as a procedure which keeps applying `pairwise-combine!` until the result is a singleton list:

```
(define (parallel-reduce! combine default)
  (define (new-procedure! list)
    (if (null? list)
        default
        (car ((until singleton? (pairwise-combine! combine))
              list))))
  new-procedure!)
```

Note that the application of the `until` iterator to `list` returns a singleton list whose element is the desired result; we therefore return the car of this list. For example,

```
((parallel-reduce! pair-min! 0) '(7 4 2 9 3))
```

first computes `(4 2 3)`, then `(2 3)`, then `(2)`, finally returning `2` as the result.

Also note that in this case it is essential that the `new-procedure!` produced by `parallel-reduce!` is mutative, because it is making repeated passes over its argument; if it had to allocate storage for its result every time, a great deal of unnecessary memory management overhead would be incurred.

9 Sorting Algorithms Based on Reduction

We are now ready to apply higher order programming techniques to the problem of implementing sorting algorithms. We limit the discussion to algorithms which sort numbers into increasing order. By passing a predicate as an argument to the sort, it is possible to sort in descending order or, in fact, to sort a list containing any type of objects using the criterion embodied in the predicate.

We can easily implement *merge-sorting* efficiently in terms of `parallel-reduce`. We assume a `merge` procedure is given, and define `listify!` as a mutative procedure that replaces each element of a list by a singleton list containing that element:

```
(define (list-one x) (cons x '()))
(define listify! (map! list-one))
```

e.g.,

```
(listify! '(1 2 3))
```

yields `((1) (2) (3))`. Merge sorting is then achieved simply by

```
(define merge-sort! (compose (parallel-reduce! merge '()) listify!))
```

This algorithm begins by merging singleton lists into ordered two-element lists, which are then merged into ordered four-element lists, etc., until a single ordered list is obtained. Since `merge` is a linear time algorithm, each stage of merging (each call to `pairwise-combine!` within `parallel-reduce!`) is linear in the length N of the original list, and since there are $\log N$ stages, `merge-sort!` is an efficient $N \log N$ algorithm.[6]

[6]Merge sorting could also be accomplished by traversing the input list in order to divide it in two, and so on recursively, but this approach is both clumsy and inefficient (neither of which has prevented it from appearing in some textbooks).

```
          1
        / | \
        3  2  6
     / |  |  |
     8  5  4  9
              |
              10
```

Figure 1 - A Tournament Tree

A less familiar approach to sorting, and one which produces a number of sorting algorithms with unusual properties, can be defined in terms of a data structure called a *tournament tree* which captures the results of comparisons done among the elements of the list to be sorted. By remembering the results of previous comparisons, tournament trees help to create highly efficient sorting procedures. The algorithms that will be presented were discovered while attempting to find a higher order representation of Floyd's Treesort algorithm [2].

A tournament tree is an ordered tree whose nodes satisfy the relation

$$\text{parent} \leq \text{child}$$

An example is given in Figure 1. Tournament trees are generalizations of heaps, as used in heapsorting, in that they embody the same relationship between parent and child nodes, but do not restrict the number of children of a node. Tournament trees are created by comparing node values and making the node with the larger value a child of the node with the smaller one. The nodes compared are the roots of two tournament trees and so when we compare them, we are actually merging the two trees by making the root of one a child of the root of the other. The tree in Figure 1 could have been formed, for example, by comparing the root of a tree rooted at 3 (containing 3, 8, and 5) and a tree rooted at 1 (containing 1, 2, 4, 6, 9, and 10).

We represent tournament trees as lists. The tree in Figure 1 is represented as:

```
(1 (3 (8) (5)) (2 (4)) (6 (9 (10))))
```

Note that such lists have the structure:

$$(\textit{parent}\ \textit{child}_1\ \ldots \textit{child}_k)$$

where each child is itself a tournament tree. The leaves are singleton lists, the simplest possible tournament trees. To facilitate the creation and manipulation of tournament trees we define the procedure `adopt!`, which adds a (leftmost) child to a parent's list of children:

```
(define (adopt! parent child)
  (replace-cdr! parent (cons child (cdr parent))))
```

Using `adopt!`, we define `play!`, the basic comparison procedure which all our sorts use. `Play!` compares two nodes and makes the larger the leftmost child of the smaller:

```
(define (play! x y)
   (if (<= (car x) (car y))
       (adopt! x y)
       (adopt! y x)))
```

We refer to a list of tournament trees as a forest. If we apply `play!` to (the roots of) two tournament trees in a forest, they will be merged as described above. If we pass `play!` and a forest to any reduction operator, `reduction1`, say, it will return a tournament tree.

The smallest element is now at the root of the tree. We may output it and remove it from further consideration. This leaves us with a forest of children of the root which can be passed to another reduction operator, `reduction2`, which will return a tournament tree. In a manner analogous to pulling a tangled rope through one's fist, we thus have Tournament Sort:

```
(define (make-tournament-sort! reduction1 reduction2)
  (define (new-sort! list)
    (let ((tournament-tree
            ((reduction1 play! '()) (listify! list))))
      ((for-each-pair
        (lambda (pair)
          (set-cdr! pair ((reduction2 play! '()) (cdr pair)))))
        tournament-tree)   ; the tournament-tree is transformed
      tournament-tree))    ; step by step into a linear list
  new-sort!)
```

Note that `make-tournament-sort!` is not a sorting procedure, but a a combinator which constructs a sorting procedure, given two reduction operators. If we pass it `parallel-reduce!` and `right-reduce!` (in that order), it creates a sorting procedure which does exactly the same comparisons (in the same order) as Floyd's Treesort [2], but using a different data structure.

```
1  3  2            1            2  3  5             2
|  |             / | \             |                |
5  4            2  3  5            4                3
                   |                              / |
                   4                             5  4

Fig. 2A         Fig. 2B         Fig. 2C         Fig. 2D
```

```
   1        1, 2     1, 2, 3     1, 2, 3, 4    1, 2, 3, 4, 5
 / | \         |         / |              |
2  3  5        3        5  4              5
   |         / |
   4        5  4

                         Fig. 3
```

```
(define parallel-sequential-tournament-sort!
  (make-tournament-sort! parallel-reduce! right-reduce!))
```

Figure 2 illustrates how `parallel-sequential-tournament-sort!` works, given the sequence of values 5, 1, 3, 4, 2. Fig. 2A shows the initial forest after the first call to `pairwise-combine!` within `parallel-reduce!`. Fig. 2B shows the initial tournament tree formed after completing the call to `parallel-reduce!`. Fig. 2C shows the forest formed by outputting node 1. Finally, Fig. 2D shows the tournament tree formed by the first call to `right-reduce!`. In Figure 3 we show the sequence of tournament-trees generated and the way in which the original tournament tree is transformed into a linear list with the elements in order. By passing `parallel-reduce!` for both reduction parameters, we obtain an entirely new sorting algorithm [10]:

```
(define parallel-tournament-sort!
  (make-tournament-sort! parallel-reduce! parallel-reduce!))
```

which sorts N elements by making slightly more than $N \log N$ comparisons when the elements are randomly ordered and only slightly more than $2N$ comparisons when they start out nearly in order or nearly in reverse order. No other algorithm currently known has all these properties.

By passing `right-reduce!` twice we obtain a sort

```
(define sequential-sequential-tournament-sort!
   (make-tournament-sort! right-reduce! right-reduce!))
```

which does the same comparisons as Insertion Sort, but has the added advantage that it functions as a priority queue; i.e., it returns the elements in order one at a time so that the sort can be halted after finding the k smallest elements, if that is what is desired. Actually, all versions of Tournament Sort function as priority queues. Note that all these sorts are mutative but could be used as applicative sorts by passing a copy of the list in place of the list to be sorted.

The Tournament Sort family of algorithms demonstrates a key point regarding our experience with the higher order imperative programming approach: it often allows us not only to find highly concise expressions of known algorithms, but also to derive entirely new algorithms in the process.

10 Conclusions

Operators such as `until`, `for-each-pair`, `reduce`, `parallel-reduce!`, etc., allow us to control complexity by creating procedures from simpler ones without constantly having to keep in mind details of the implementation of the lower level procedures. While to some extent the same can be said of any procedure called from inside another, operators are particularly effective in this regard because they allow us to extend a programming language directly to express algorithms more clearly.

Of course, the line between programming and language design could easily become blurred. It is easy to create a Tower of Babel where each person writes programs in a different language or, worse yet, where every program a person writes is expressed in a language which was never used before and will never be used again. To avoid this, the operators used to extend a language must be carefully chosen and their semantics must be very clear.

Higher order programming techniques deal directly with this issue. Using a comparatively small number of operators, including a few that are designed to work with mutative procedures, it is possible to dramatically increase the expressive power of a language and to create substantial and practically useful algorithms.

In this paper, to keep our examples simple for publication purposes, we did not use some features available in most Scheme implementations, such as `(define-integrable ... )` for directing the compiler to replace certain procedure calls by the procedure definition inline. In practice, one would want to take advantage of the efficiency improvements afforded by these features.

Many of the techniques illustrated in this paper can be used in Ada programming via generics, and, in principle, can result in highly efficient code, since no run-time type checking is required and much of the layering of procedure calls can be removed by inlining.[7] Although some techniques are not supported, such as procedures with second order procedure parameters (e.g., `make-tournament-sort!`), other techniques not provided in Scheme become available via generic packages. These techniques are particularly suitable as the basis for development of generic software libraries [7].

Acknowledgements. Earlier drafts of this paper were written while the second author was at General Electric Research and Development Center, Schenectady, NY. We are grateful to Leon Levy and Martin Shannon of AT&T Bell Labs for insightful comments and suggestions.

References

[1] J. Backus, "Can Programming Be Liberated from the von Neumann Style," *Communications of ACM*, Vol. 21, No.8, August 1978.

[2] R. W. Floyd, "Treesort (Algorithm 113)", *Communications of ACM*, December 1964, 701.

[3] K. E. Iverson, "Operators," TOPLAS 1 (2), October 1979.

[4] B. Liskov, et al., "CLU Reference Manual," Springer-Verlag, 1984.

[5] R. Milner, "A Proposal for Standard ML," *1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[6] D. Kapur, D. R. Musser, and A. A. Stepanov, "Operators and Algebraic Structures," *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire, October 1981.

[7] D.R. Musser and A.A. Stepanov, "A Library of Generic Algorithms in Ada," *Proc. ACM SIGAda Conference*, Boston, December 9-11, 1987.

[8] *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, U. S. Department of Defense, January 1983.

[7] "In principle" because currently many Ada compilers do not handle complex combinations of layered generics and inlining directives at all.

[9] J.A. Rees and W. Clinger, eds., "The Revised^3 Report on the Algorithmic Language Scheme," *SIGPLAN Notices 21*, 12, December 1986.

[10] A. Stepanov and A. Kershenbaum, "Using Tournament Trees to Sort," Technical Report 86-13, Center for Advanced Technology In Telecommunications, Polytechnic University.

[11] B. Stoustrup, *The C++ Programming Language*, Addison-Wesley, December 1986.

[12] D. A. Turner, "Miranda: A non-strict functional language with polymorphic types," in J.-P. Jouannaud (ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* 201, Springer Verlag, 1985.