CSci 450: Organization of Programming Languages
Developing Functional Programs

H. Conrad Cunningham

16 September 2017

Contents

6 Developing Functional Programs **2**
6.1 Chapter Introduction 2
6.2 Developing a Cosequential Processing Family 2
6.2.1 Function generalization 2
6.2.2 Scope 2
6.2.3 Frozen spots 3
6.2.4 Hot spots 4
6.2.5 Hot spot #1: Variability in total ordering 4
6.2.6 Hot spot #2: Variability in record format 5
6.2.7 Hot spot #3: Independent variability of sequences 6
6.2.8 Hot spot #4: Variability in sequence transformations . . 7
6.2.9 Hot spot #5 :Variability of sequence source/destination . 11
6.2.10 Bag and set operation implementations 11
6.2.11 Sequential file update algorithm (TODO) 13
6.2.12 Recap 14
6.3 Exercises 14
6.4 References 14
6.5 Terms and Concepts 15

Acknowledgements: In Summer 2016, I adapted and revised the discussion of function generalization from my and Pallavi Tadepalli's paper "Using Function Generalization to Design a Cosequential Processing Framework" (*Proceedings of the 39th Hawaii International Conference on System Sciences*, January 2006),

In 2017, I continue to develop this module.

I maintain these notes as text in Pandoc's dialect of MarkDown using embedded LaTeX markup for the mathematical formulas and translate them to HTML and

PDF

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of Septeber 2017 is a recent version of Firefox from Mozilla.

TODO:

- Add chapter goals and outcomes
- Cleanup function generalization
- Add eager evaluated version of `merge4b`, perhaps rename `coseq`
- Add sequential file update example
- Add more content – e.g., function pipeline?
- Add Exercises
- Add Terms and Concepts

6 Developing Functional Programs

6.1 Chapter Introduction

TODO

6.2 Developing a Cosequential Processing Family

6.2.1 Function generalization

In a previous chapter, we examined *families* of related functions to define generic, higher-order functions to capture the computational pattern for each family. In this chapter, we approach *function generalization* more systematically.

The function generalization approach begins with a prototype member of the family. As in similar techniques for building object-oriented software frameworks, we apply Scope-Commonality-Variability (SCV) analysis to the family represented by this prototype. We identify the:

- *scope* – what is in the family and what is not.
- *commonalities* – the common, reusable aspects of the family. We call these the *frozen spots*.
- *variabilities* – the aspects that are likely to vary among the different members of the family. We call these the *hot spots*.

Once we have the above, we incrementally transform the prototype function for each of the hot spots.

A generalizing transformation may replace specific values or data types at a hot spot by parameters. We may make a type more abstract, perhaps making it polymorphic. Or we may break a type into several types if it plays potentially different roles.

Similarly, a generalizing transformation may replace fixed, specialized operations at a hot spot by abstract operations. We may make an abstract operation a higher-order parameter of the generalized function.

We ensure that the new, more general function can implement the previous, more specialized function.

6.2.2 Scope

Cosequential processing concerns the coordinated processing of two ordered sequences to produce some result, often a third ordered sequence. Key requirements include:

- Both input sequences must be ordered according to the same total ordering.
- The processing should be incremental, where only a few elements of each sequence (perhaps just one) are examined at a time.

This important family includes the ascending merge needed in merge sort, set and bag operations, and sequential file update applications.

Consider a function `merge0` that takes two ascending sequences of integers and merges them together to form a third ascending sequence.

```
merge0 :: [Int] -> [Int] -> [Int]  -- xs, ys
merge0 [] ys = ys
merge0 xs [] = xs
merge0 xs@(x:xs') ys@(y:ys')
  | x < y    = x : merge0 xs' ys
  | x == y   = x : merge0 xs' ys'
  | x > y    = y : merge0 xs  ys'
```

This function takes two lists of integers and returns a list of integers.

The `merge0` function must satisfy a number of properties:

- *Precondition*: The two input lists must be in ascending order.
- *Postcondition*: The output list must also be in ascending order. The number of times an element appears in the output list is the maximum number of times it appears within one of the two input lists.
- *Termination*: The sum of the lengths of the two input sequences must decrease by at least one for each call of the recursive function.

For the cosequential processing family, let take function `merge0` as the prototype member.

Aside: The `merge0` function differs from the merge function we used in merge sort in a previous chapter. For merge sort, the `x == y` leg would need to remove the head of only one of the input lists, i.e., be defined as either `x : merge0 xs' ys` or `x : merge0 xs ys'`.

6.2.3 Frozen spots

Considering the scope and examining the prototype function `merge0`, we identify the following frozen spots for the family of functions:

1. The input consists of two sequences ordered by the same total ordering.
2. The output consists of a sequence ordered by the same total ordering as the input sequences.
3. The processing is incremental. Each step examines the current element from each input sequence and advances at least one of the input sequences for subsequent steps.
4. Each step compares the current elements from the input sequences to determine what action to take at that step.

The merge function represents the frozen spots of the family. It gives the common behavior of family members and the relationships among the various elements of the hot spot subsystems. A hot spot subsystem consists of a set of Haskell functions, types, and class definitions that add the desired variability into the merge function.

6.2.4 Hot spots

Again considering the scope and examining the prototype function `merge0`, we can identify the following hot spots:

1. Variability in the total ordering used for the input and output sequences, i.e., of the comparison operators and input sequence type.
2. The ability to have more complex data entities in the input and output sequences, i.e., variability in “record” format.
3. The ability to vary the input and output sequence structures independently of each other.
4. Variability in the transformations applied to the data as it passes into the output.
5. Variability in the sources of the input sequences and destination of the output sequence.

We need to be careful to avoid enumerating hot spots that are unlikely to be needed in an application.

Now let's analyze each hot spot, design a hot spot subsystem, and carry out the appropriate transformations to generalize the Haskell program.

6.2.5 Hot spot #1: Variability in total ordering

In the function `merge0`, the input and output sequences are restricted to elements of type `Int` and the comparison operations, hence, to the integer comparisons.

The responsibility associated with hot spot #1 is to enable the base type of the sequences to be any type upon which an appropriate ordering is defined. In this transformation, we still consider all three sequences as containing simple values of the same type.

We can generalize the function to take and return sequences of any ordered type by making the type of the list polymorphic. Using a type variable `a`, we can redefine the type signature to be `[a]`. However, we need to constrain type `a` to be a type for which an appropriate total ordering is defined. We do this by requiring that the type be restricted to those in the predefined Haskell type class `Ord`. This class consists of the group of types for which all six relational operators are defined.

The function resulting from generalization step is `merge1`.

```
merge1 :: Ord a => [a] -> [a] -> [a]  -- xs, ys
merge1 []  ys = ys
merge1 xs  [] = xs
merge1 xs@(x:xs') ys@(y:ys')
  | x <  y    = x : merge1 xs' ys
  | x == y    = x : merge1 xs' ys'
  | x >  y    = y : merge1 xs  ys'
```

This function represents the frozen spots of the cosequential processing framework. The implementation of class `Ord` used in a program is hot spot #1. To satisfy the requirement represented by frozen spot #1, we require that the two lists `xs` and `ys` be in ascending order.

Note that, if we restrict `merge1` polymorphic type `a` to `Int`, then:

```
merge1 xs ys == merge0 xs ys
```

That is, the generic function `merge1` can be specialized to be equivalent to `merge0`.

6.2.6 Hot spot #2: Variability in record format

The `merge1` function works with sequences of any type that have appropriate comparison operators defined. This allows the elements to be of some built-in type such as `Int` or `String` or some user-defined type that has been declared as an instance of the `Ord` class. Thus each individual data item is of a single type.

In general, however, applications in this family will need to work with data elements that have more complex structures. We refer to these more complex structures as *records* in the general sense, not just the Haskell data structure by that name.

The responsibility associated with hot spot #2 is to enable the elements of the sequences to be values with more complex structures, i.e., records. Each record is composed of one or more fields of which some subset defines the key. The value of the key provides the information for ordering the records within that sequence.

In this transformation, we still consider all three sequences as containing simple values of the same type. We abstract the `key` as a function on the record type that returns a value of some `Ord` type to enable the needed comparisons. We transform the `merge1` function by adding `key` as a higher-order parameter.

The function resulting from this generalization is `merge2`.

```
merge2 :: Ord b => (a -> b) -> [a] -> [a] -> [a]  -- key, xs, ys
merge2 key []  ys  = ys
merge2 key xs  []  = xs
merge2 key xs@(x:xs') ys@(y:ys')
  | key x <  key y = x : merge2 key xs' ys
  | key x == key y = x : merge2 key xs' ys'
  | key x >  key y = y : merge2 key xs  ys'
```

The higher-order parameter `key` represents hot spot #2 in the generalized function design.

Hot spot #1 is the implementation of Haskell class `Ord` for values of type `b`.

To satisfy the requirement represented by frozen spot #1, the sequence of keys corresponding to each input sequence, i.e., `map key xs` and `map key ys`, must be in ascending order.

Also note that

```
merge2 id xs ys == merge1 xs ys
```

where `id` is the identity function. Thus `merge1` is a specialization of `merge2`.

6.2.7 Hot spot #3: Independent variability of sequences

In `merge2`, the records are of the same type in all three sequences. The key extraction function is also the same for all sequences.

Some cosequential processing applications, however, require that the record structure vary among the sequences. For example, the sequential file update application usually involves a master file and a transaction file as the inputs and a new master file as the output. The master records and transaction records usually carry different information.

The responsibility associated with hot spot #3 is to enable the three sequences to be varied independently. That is, the records in one sequence may differ in structure from the records in the others.

This requires separate key extraction functions for the two input sequences. These must, however, still return key values from the same total ordering. Because the data types for the two input sequences may differ and both may differ from the output data type, we must introduce record transformation functions that convert the input data types to the output types.

The function resulting from the transformation is `merge3`.

```
merge3 :: Ord d => (a -> d) -> (b -> d)    -- kx, ky
                -> (a -> c) -> (b -> c) -- tx, ty
                -> [a] -> [b] -> [c]    -- xs, ys
merge3 kx ky tx ty xs ys   = mg xs ys
    where
        mg [] ys         = map ty ys
        mg xs []         = map tx xs
        mg xs@(x:xs') ys@(y:ys')
          | kx x <  ky y = tx x : mg xs' ys
          | kx x == ky y = tx x : mg xs' ys'
          | kx x >  ky y = ty y : mg xs  ys'
```

Higher-order parameters `kx` and `ky` are the *key* extraction functions for the first and second inputs, respectively. Similarly, `tx` and `ty` are the corresponding functions to *transform* those inputs to the output.

Hot spot #3 consists of these four functions. In some sense, this transformation subsumes hot spot #2.

To avoid repetition of the many unchanging arguments in the recursive calls, the definition of `merge3` uses an auxiliary function definition `mg`.

The nonrecursive legs use the higher-order library function `map`. To satisfy the requirement represented by frozen spot #1, the sequence of keys corresponding to each input sequence, i.e., `map kx xs` and `map ky ys`, must be in ascending order.

If `xs` and `ys` are of the same type, then it is true that:

```
merge3 key key id id xs ys ==  merge2 key xs ys
```

Thus `merge2` is a specialization of `merge3`.

6.2.8 Hot spot #4: Variability in sequence transformations

Function `merge3` enabled simple one-to-one, record-by-record transformations of the input sequences to create the output sequence. Such simple transformations are not sufficient for practical situations.

For example, in the sequential file update application, each key may be associated with no more than one record in the master file. However, there may be any number of update transactions that must be performed against a master record before the new master record can be output. Thus, there needs to be some local state maintained throughout the processing of all the transaction records associated with one master record.

Before we address the issue of this variation directly, let us generalize the merge function to make the state that currently exists (i.e., the evolving output list) explicit in the parameter list.

To do this, we replace the backward linear recursive `merge3` function by its *tail recursive* generalization. That is, we add an *accumulating parameter* `ss` that is used to collects the output during the recursive calls and then to generate the final output when the end of an input sequence is reached.

The initial value of this argument is normally a nil list, but it does enable some other initial value to be prepended to the output list. This transformation is shown as function `merge4a` below.

```
merge4a :: Ord d => (a -> d) -> (b -> d)        -- kx, ky
                 -> (a -> c) -> (b -> c)     -- tx, ty
                 -> [c] -> [a] -> [b] -> [c] -- ss, xs, ys
merge4a kx ky tx ty ss xs ys = mg ss xs ys
    where
        mg ss [] ys      = ss ++ map ty ys
        mg ss xs []      = ss ++ map tx xs
        mg ss xs@(x:xs') ys@(y:ys')
          | kx x <  ky y = mg (ss ++ [tx x]) xs' ys
          | kx x == ky y = mg (ss ++ [tx x]) xs' ys'
          | kx x >  ky y = mg (ss ++ [ty y]) xs  ys'
```

Note that the following holds:

```
merge4a kx ky tx ty ss xs ys == ss ++ merge3 kx ky tx ty xs ys
```

Thus function `merge3` is a specialization of `merge4a`.

Unfortunately, building up the state `ss` requires a relatively expensive appending to the end of a list (e.g., `ss ++ [tx x]` in the third leg).

Now consider hot spot #4 more explicitly. The responsibility associated with the hot spot is to enable the use of more general transformations on the input sequences to produce the output sequence.

To accomplish this, we introduce an explicit *state* to record the relevant aspects of the computation to some position in the two input sequences. Each call of the merge function can examine the current values from the input sequences and update the value of the state appropriately for the next call.

In some sense, the merge function "folds" together the values from the two input sequences to compute the state. At the end of both input sequences, the merge function then transforms the state into the output sequence.

To accomplish this, we can generalize `merge4a`. We generalize the accumulating parameter `ss` in `merge4a` to be a parameter `s` that represents the state. We also replace the two simple record-to-record transformation functions `tx` and `ty` by more flexible transformation functions `tl`, `te`, and `tg`, that update the state in the three guards of the recursive leg and functions `tty` and `ttx` that update the state when the first and second input sequences, respectively, become empty.

For the "equals" guard, the amount that the input sequences are advanced also becomes dependent upon the state of the computation. This is abstracted as functions `nex` on the first input sequence and `ney` on the second. To satisfy the requirement represented by frozen spot #3, the pair of functions `nex` and `ney` must make the following progress requirement true for each call of `mg`:

```
if (kx x == ky y) then
  (length (nex s xs) < length xs) ||
  (length (ney s ys) < length ys)
else True
```

That is, the client of the framework must ensure that at least one of the input sequences will be advanced by at least one element. We also introduce the new function `res` to take the final state of the computation and return the output sequence.

The above transformation results in function `merge4b`.

```
merge4b :: Ord d => (a -> d) -> (b -> d) -> -- kx, ky
                    (e -> a -> b -> e) ->   -- tl
                    (e -> a -> b -> e) ->   -- te
                    (e -> a -> b -> e) ->   -- tg
                    (e -> [a] -> [a])  ->   -- nex
                    (e -> [b] -> [b]) ->    -- ney
                    (e -> a -> e) ->        -- ttx
                    (e -> b -> e) ->        -- tty
                    (e -> [c]) -> e ->      -- res, s
```

```
                      [a] -> [b] -> [c]       -- xs, ys
    merge4b kx ky tl te tg nex ney ttx tty res s xs ys
          =  mg s xs ys
        where
            mg s [] ys       = res (foldl tty s ys)
            mg s xs []       = res (foldl ttx s xs)
            mg s xs@(x:xs') ys@(y:ys')
              | kx x <  ky y = mg (tl s x y) xs' ys
              | kx x == ky y = mg (te s x y) (nex s xs) (ney s ys)
              | kx x >  ky y = mg (tg s x y) xs ys
```

The function uses the Prelude function `foldl` in the first two legs. This function continues the computation beginning with the state computed by the recursive leg and processes the remainder of the nonempty input sequence by "folding" the remaining elements as defined in the functions `ttx` and `tty`.

As was the case for `merge3`, frozen spot #1 requires that `map kx xs` and `map ky ys` be in ascending order for calls to `merge4b`

Hot spot #4 consists of the eight functions `tl`, `te`, `tg`, `ttx`, `tty`, `nex` `ney`, and `res`.
The following property also holds:

```
    merge4b kx ky
       (\ss x y -> ss ++ [tx x])  -- tl
       (\ss x y -> ss ++ [tx x])  -- te
       (\ss x y -> ss ++ [ty y])  -- tg
       (\ss xs -> tail xs)        -- nex
       (\ss ys -> tail ys)        -- ney
       (\ss x -> ss ++ [x])       -- ttx
       (\ss y -> ss ++ [y])       -- tty
       id ss xs ys                -- res, ss, xs, ys
    == merge4a kx ky tx ty ss xs ys
    == ss ++ merge3 kx ky tx ty xs ys
```

That is, we can define the general transformation functions so that they have the same effect as the record-to-record transformations of `merge4a`. The statement of this property uses anonymous functions. (lambda expression) feature of Haskell.

Thus function `merge3` is a specialization of `merge4a`, which in turn is a specialization of function `merge4b`.

A problem with the above "implementation" of `merge3` is that the `merge4b` parameters `tl`, `te`, `tg`, `ttx`, and `tty` all involve an expensive operation to append to the end of the list `ss`.

An alternative would be to build the state sequence in reverse order and then reverse the result as shown below.

```
    merge4b kx ky
```

```
        (\ss x y -> reverse (tx x) ++ ss)  -- tl
        (\ss x y -> reverse (tx x) ++ ss)  -- tl
        (\ss x y -> reverse (ty y) ++ ss)  -- tg
        (\ss xs -> tail xs)                -- nex
        (\ss ys -> tail ys)                -- ney
        (\ss x -> x : ss)                  -- ttx
        (\ss y -> y : ss)                  -- tty
        reverse ss xs ys                   -- res, ss, xs, ys
    == merge4a kx ky tx ty ss xs ys
    == ss ++ merge3 kx ky tx ty xs ys
```

TODO: Possibly include a version with selective eager evaluation (similar to below) and rename coseq.

```
merge4c :: Ord d => (a -> d) -> (b -> d) -> -- kx, ky
                    (e -> a -> b -> e) ->   -- tl
                    (e -> a -> b -> e) ->   -- te
                    (e -> a -> b -> e) ->   -- tg
                    (e -> [a] -> [a])  ->   -- nex
                    (e -> [b] -> [b]) ->    -- ney
                    (e -> a -> e) ->        -- ttx
                    (e -> b -> e) ->        -- tty
                    (e -> [c]) -> e ->      -- res, s
                    [a] -> [b] -> [c]       -- xs, ys
merge4b kx ky tl te tg nex ney ttx tty res s xs ys
        =  mg s xs ys
    where
        mg s [] ys       = res (foldl tty s ys)
        mg s xs []       = res (foldl ttx s xs)
        mg s xs@(x:xs') ys@(y:ys')
          | kx x <  ky y = (mg $! (tl s x y)) xs' ys
          | kx x == ky y = (mg $! (te s x y)) ((nex $! s) xs)
                                               ((ney $! s) ys)
          | kx x >  ky y = (mg $! (tg s x y)) xs ys'
```

6.2.9 Hot spot #5 :Variability of sequence source/destination

Hot spot #5 concerns the ability to take the input sequences from many possible sources and to direct the output to many possible destinations.

In the Haskell merge functions, these sequences are represented as the pervasive polymorphic list data type. The redirection is simply a matter of writing appropriate functions to produce the input lists and to consume its output list. No changes are needed to the `merge4b` function itself.

Of course, for any expressions (e.g., function calls) `ex` and `ey` that generate the input sequence arguments `xs` and `ys` of `merge4b`, it must be the case that sequences `map kx ex` and `map ky ey` are ascending.

6.2.10 Bag and set operation implementations

TODO: Possibly reexpress some of the lambdas above with standard combinators.

Mathematically, a *bag* (also called a *multiset*) is an unordered collection of elements in which each element may occur one or more times. We can model a bag as a total function (called the *multiplicity function*) over the domain of elements to the natural numbers where the numbers 0 and above denote the number of occurrences of the element in the bag.

A *set* is thus a bag for which there is no more than one occurence of any element.

If we restrict the elements to a Haskell data type that is an instance of class `Ord`, we can represent a bag by an ascending list of values and a set by an increasing list of values. With this representation, we can implement the bag an set operations as special cases of cosequential processing.

The *intersection* of two bags consists of only the elements that occur in both bags such that the number of occurrences is the minimum number for the two input bags. We can express the bag intersection of two ascending lists in terms of `merge4b` as follows.

```
    bagIntersect xs ys =
        merge4b id id
                (\s x y -> s)
                (\s x y -> x:s)
                (\s x y -> s)
                (\s xs -> tail xs)
                (\s ys -> tail ys)
                (\s x -> s)
                (\s y -> s)
                reverse [] xs ys
```

This function only adds an element to the output when it occurs in both input lists.

If we require the two input lists to be increasing, the above also implements set intersection.

The *sum* of two bags consists of the elements that occur in either bag such that the number of occurrences is the total number for both bags. We can express the bag sum of two ascending lists in terms of `merge4b` as follows.

```
    bagSum xs ys =
       merge4b id id
               (\s x y -> x:s)
               (\s x y -> x:y:s)
```

```
(\s x y -> y:s)
(\s xs -> tail xs)
(\s ys -> tail ys)
(\s x -> x:s)
(\s y -> y:s)
reverse [] xs ys
```

The *union* of two bags consists of the elements that occur in either bag such that the number of occurrences is the minimum number in the two input lists. The prototype function `merge0` implements this operation on ascending lists.

The *subtraction* of bag `B` from bag `A`, denoted `A - B`, consists of only the elements that occur in both bags such that the number of occurrences is the number of occurrences in `A` minus those in `B`.

Questions:

- How can we represent set union in terms of `merge4b`?
- How can we represent a merge function that can be used in the merge sort of two lists (whose elements are from an instance of class `Ord`)?
- How can we implement a bag union function `bagUnion` in terms of `merge4b`?
- How can we implement a bag subtraction function `bagSub xs ys` in terms of `merge4b`?
- If the elements of the input lists are not instances of class `Ord`, how can we implement bag union? bag intersection?

6.2.11 Sequential file update algorithm (TODO)

```
-- Simple Master-Transaction Update
-- Master increasing list [(Account,Amount)]
--   with Account < maxAccount
-- Transaction ascending list [(Account,Amount)]
--   with Account < maxAccount
-- Result is new Master increasing list [(Account,Amount)]
--  with Account < maxAccount

type Account = Int
type Amount = Integer

seqUpdate :: [(Account,Amount)] -> [(Account,Amount)]
             -> [(Account,Amount)]
seqUpdate = merge4c fst        fst
                    masterlt   mastereq  mastergt
                    masternext transnext
                    notrans    nomaster
                    getResult  initState

initState = ([],[])

maxAccount = maxBound :: Account

masterlt (out,[])    m t = (m:out,[])
    -- no transactions for this master
masterlt (out,[cur]) m t = (cur:out,[])
    -- processed all transactions for this master

mastereq (out,[])        (ma,mb) (_,tc)
    = (out, [(ma,mb+tc)])   -- first transaction
mastereq (out,[(sa,sb)]) (_,_)   (_,tc)
    = (out, [(sa,sb+tc)])   -- subsequent transaction

mastergt (_,_) m t
    = error ("Transactions not ascending at " ++ show t)

masternext (_,_) ms = ms -- do not advance master on eq

transnext  (_,_) ts = let ys = tail ts
                      in if null ys then [(maxAccount,0)] else ys
                      -- advance transaction on eq
                      -- force master with final (maxAccount,0)

notrans (out,[])    m = (m:out,[])
notrans (out,[cur]) m = (m:cur:out,[])

nomaster ([],[])  t  -- only for empty master list
    = error ("Unmatched transaction " ++ show t)
nomaster (out,[]) (maxInt,_) -- transaction list ended
    = (out,[])
nomaster _        t
    = error ("Unmatched transaction " ++ show t)

getResult (nms,[]) = reverse nms
```

6.2.12 Recap

This case study illustrates the function generalization method. It begins with a simple Haskell program to merge two ascending lists of integers into third ascending list of integers. This program is generalized in a step by step fashion to produce a new program that is capable of carrying out any operation from the family of cosequential processing programs.

Although some members of the cosequential processing family can be rather complicated, the family has the characteristic that the primary driver for the algorithm can be concisely stated as a simple loop (i.e., recursive function).

6.3 Exercises

TODO

6.4 References

[**Cunningham-Tadepalli 2006**] H. Conrad Cunningham and Pallavi Tadepalli. "Using Function Generalization to Design a Cosequential Processing Framework," In *Proceedings of the 39th Hawaii International Conference on System Sciences*, January 2006.

6.5 Terms and Concepts

TODO