Using Function Generalization to Design a Cosequential Processing Framework

H. Conrad Cunningham and Pallavi Tadepalli
Department of Computer and Information Science, University of Mississippi,
201 Weir Hall, University, MS 38677 USA
Email: {cunningham,pallavi}@cs.olemiss.edu

***Abstract*— *Framework design is a multifaceted endeavor undertaken to promote reuse of software within a family of related applications. Traditional approaches involve either the evolution or the systematic design of the needed generic structure. This paper explores a systematic design approach called function generalization. In this approach, framework design begins with an executable specification expressed as a set of functions in a functional programming language. This set is analyzed to identify the common and variable aspects of the family of related applications. The set of functions is then transformed in a series of steps to produce a generalized application corresponding to the family. Each step generalizes one variable aspect of the family by introducing higher-order (function) parameters or polymorphic parameters into the functions in the set. The resulting set of generalized functions can be converted to Java code using design patterns to guide the framework construction.*

I. Introduction

A *software framework* is a generic application that provides the skeleton upon which various customized applications can be constructed. It is an object-oriented technique for reuse of software [7] within a family of related applications [15]. Each framework consists of some common aspects that are reusable, called *frozen spots*, and certain variable aspects, called *hot spots*, whose implementations vary among the different applications of the family [19]. The frozen spots are embodied in the overall collaborative structure of the classes in the framework and concrete implementations of various methods and classes. The hot spots are represented as abstract base classes (or interfaces) in the framework. A particular custom application can be built by providing appropriate implementations of the relevant hot spot abstractions. Frameworks are generally built using design patterns [10] that fit with the concept of hot spots and frozen spots [19].

In most nontrivial frameworks, it is not possible to come up with the right hot spot abstractions just by thinking about the problem informally. Using ad hoc methods, three implementation cycles are often needed to develop a sufficient understanding of the domain to construct good abstractions [18]. Various explicit design techniques seek to systematize the process by identifying the frozen and hot spots a priori. In Schmid's *systematic generalization* methodology [19], framework designers take the object-oriented design for a specific application within the family and convert it into a framework design by a sequence of generalizing transformations. Each transformation corresponds to the introduction of a hot spot abstraction into the structure. The methodology proposes techniques for analyzing the hot spot and constructing an appropriate design for the hot spot subsystem.

This paper takes a similar systematic approach. However, instead of generalizing the class structure of an application design, it generalizes the functional structure of an executable specification to produce a generic application. This *function generalization* approach introduces the hot spot abstractions into the design by replacing concrete operations by more general abstract operations. These abstract operations become parameters of the generalized functions. That is, the generalized functions are higher order, having parameters that are themselves functions. We explore these ideas here by expressing the specification in the concrete syntax of the purely functional programming language Haskell [16], [23]. After generalizing the various hot spots of the family, we can use the resulting generalized functions to define a framework in an object-oriented language such as Java.

This paper uses cosequential processing, which is described in Section II, as the example family. Section III presents the generalization of the key Haskell function for cosequential processing, generalizing the hot spots one by one. Section IV describes the translation of the resulting higher-order Haskell function into a Java framework, and Section V uses that framework to construct a sequential file update application. Then, Section VI discusses this function generalization approach in relation to other framework development techniques and Section VII summarizes the paper.

II. Cosequential Processing

Cosequential processing concerns the coordinated processing of two ordered sequences to produce some result, often a third ordered sequence [8]. One key aspect of cosequential processing is that both input sequences must be ordered according to the same total ordering. Another is that the processing should, in general, be incremental. That is, only a few elements of each sequence (perhaps just one) are examined at a time. This important family includes set operations [11] and sequential file update applications [5], [6].

As the baseline example, consider taking two ascending sequences of integers and merging them together to form a third ascending sequence. In Haskell, we can define a merge function as shown in Fig. 1. The first line of the definition gives the type signature for `merge0` as a function that takes two

```
merge0 :: [Int]->[Int]->[Int]
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'
```

Fig. 1. Baseline merge

(Curried) arguments that are lists of integers and returns a list of integers. The second and third lines are equations that define the result when the first and second input lists, respectively, are empty. That is, the other input list is returned. The four remaining lines give an equation that defines the function for the case when both input lists are nonempty. That is, they match a pattern like `xs@(x:xs')` where `x` takes on the value of the head element, `xs'` takes on the value of the tail (i.e., the list remaining after the head is removed), and `xs` refers to the entire list argument. This equation includes three guarded definitions for the cases where the head of the first list is less than, equal to, and greater than the head of the second list. For example, the less-than case results in an output list that consists of the head of the first input list prepended onto the list formed by recursively applying `merge0` to the remainder of the first list and the second list.

The `merge0` function must satisfy a number of properties. A precondition is that the two input lists must be in ascending order. A postcondition is that 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. To guarantee termination of the recursive processing, the sum of the lengths of the two input sequences must decrease by at least one for each call of the recursive function.

Building a framework for cosequential processing involves the general principles of framework design. We discuss those in the next section.

III. Framework Design

Schmid's systematic generalization methodology identifies the following steps for construction of a framework [19]:

- creation of a fixed application model
- hot spot analysis and specification
- hot spot high level design
- generalization transformation

In Schmid's approach, the fixed application model is an object-oriented design. A generalizing transformation replaces a fixed, specialized class at a hot spot by an abstract base class. The hot spot's features are accessed through the common interface of the abstract class. However, different concrete implementations of the base class provide the variant behaviors required for the hot spot.

The function generalization approach taken in this paper begins with an executable specification for an application, that is, a specific set of functions expressed as a Haskell program. A generalizing transformation may replace a data type at a hot spot by one that is more general, typically making it a polymorphic parameter of functions in the program. Alternatively, a generalizing transformation may replace a fixed, specialized operation at a hot spot by an abstract operation. The abstract operation typically becomes a higher-order (function) parameter of functions in the program.

Haskell is a good choice because it has several characteristics that make it useful in design of generic programs, which is what frameworks are. First, Haskell functions are polymorphic. They can be defined to take parameters from a range of related data types. Second, Haskell offers powerful features to encapsulate abstractions. In particular, Haskell functions are higher-order. That is, they can take functions as parameters and return functions as results. Third, Haskell programs tend to be concise. Fourth, Haskell programs can readily be manipulated mathematically using the language itself. Fifth, as with OOP languages, Haskell programs can be developed using design patterns [25], [26] and refactored with tool support [14], [24].

Polymorphism and higher-order functions are important because they promote the reuse of software [22]. These features also form a bridge between the functional programming and object-oriented programming (OOP) approaches, enabling similar structures to be created within each paradigm [20]. Higher-order functions in Haskell and inheritance and delegation in an OOP language both provide the means for definition of generic code that can be readily customized to a specific application. The polymorphic parameters of Haskell functions make the functions reusable across a number of types. Similarly, the subclass polymorphism that exists in OOP languages enables abstract interfaces to be reused among a number of types.

Haskell's conciseness and its amenability to mathematical reasoning using the language itself make it attractive as a means for carrying out generalizing program transformations. OOP languages are, relatively speaking, more verbose and more difficult to manipulate mathematically. Therefore, the function generalization approach in this paper uses Haskell for the generalizing transformations. After each transformation, it expects that the result is a valid Haskell program that can be compiled and executed. The program can potentially be restructured using refactoring techniques and tools [14].

For the cosequential processing example, function `merge0` from Fig. 1 is the initial executable specification. The first step of the systematic process is to define the scope of the family, as we do in Section II. The second step is to identify the frozen spots and hot spots. Care should be taken to avoid enumerating hot spots that are unlikely to be needed in an application of the framework [2]. The third step is to analyze each hot spot, design a hot spot subsystem, and carry out the appropriate transformations to generalize the Haskell program. The final step is to transform the Haskell program into an appropriate Java program.

Considering the scope and examining the initial specification `merge0`, we can identify the following frozen spots:

1) The inputs consist 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.

Again considering the scope and examining the initial specification, 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 sequences 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.

The merge function represents the frozen spots of the framework. 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 function, type, and class definitions that add the desired variability into the merge function.

The subsections that follow examine the above hot spots and carry out appropriate generalization steps. To keep the presentation, we deemphasize error detection, reporting, and recovery. However, a production version of the framework would likely have additional frozen spots and hot spots to support these tasks in a customizable manner.

A. Variability in Total Ordering

In the function `merge0`, the input and output sequences are restricted to elements of type integer, that is, to `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 using the polymorphic feature of Haskell programs. Using a type variable `a`, we can redefine the type signature to be `[a] -> [a] -> [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 the relational operators ==, /=, <, <=, >, and >= have been defined.

Fig. 2 shows `merge1`, the function resulting from this generalization step. This function represents the frozen spots

```
merge1 :: Ord a => [a] -> [a] -> [a]
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'
```

Fig. 2. Generalized comparisons

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`'s type variable `a` to be `Int` values, then:

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

B. 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 are records.

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.

Fig. 3 shows `merge2`, the function resulting from this generalization. Its first parameter (i.e., `key`) is of type `(a -> b)`, which denotes a unary function type in Haskell. This higher-order parameter represents hot spot #2 in the framework 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. (Function `map` from the Haskell library applies its function argument to each element of a list and returns the resulting list.) Also note that

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

where `id` is the identity (combinator) function.

C. 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,

```
merge2 :: Ord b =>
           (a -> b) -> [a] -> [a] -> [a]
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'
```

Fig. 3. Generalized record format

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.

Fig. 4 shows `merge3`, the function resulting from the transformation. Parameters `kx` and `ky` are the key extraction functions for the first and second inputs, respectively; `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 a local 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. Note that, 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
```

D. 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.

```
merge3 :: Ord d => (a -> d) -> (b -> d)
                -> (a -> c) -> (b -> c)
                -> [a] -> [b] -> [c]
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'
```

Fig. 4. Independent sequences

```
merge4a :: Ord d => (a -> d) -> (b -> d)
             -> (a -> c) -> (b -> c)
             -> [c] -> [a] -> [b] -> [c]
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'
```

Fig. 5. Tail recursion

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 output) explicit in the parameter list. To do this, we replace the backward linear recursive `merge3` function by its *tail recursive* generalization [12]. That is, we add an *accumulating parameter* `ss` that is used to collect 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` in Fig. 5. Note that the following holds:

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

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

```
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
```

Fig. 6. Variable sequence transformations

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.

Fig. 6 shows the function `merge4b` resulting from the above transformation. The function uses the Haskell library function `foldl` in the first two legs. This function continues

```
merge4b kx ky
   (\ss x y -> ss ++ [tx x])
   (\ss x y -> ss ++ [tx x])
   (\ss x y -> ss ++ [ty y])
   (\ss xs -> tail xs)
   (\ss ys -> tail ys)
   (\ss x -> ss ++ [x])
   (\ss y -> ss ++ [y]) ss xs ys
== merge4a kx ky tx ty ss xs ys
```

Fig. 7. Equivalence of `merge4a` and `merge4b`

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`. Note that the property stated in Fig. 7 holds. 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 the anonymous function (lambda expression) feature of Haskell. For example, `(\ss xs -> tail xs)` denotes a function that takes two arguments (the second one being a list) and returns the result of applying the built-in Haskell function `tail` to the second argument.

E. 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 cpsalls) `es` and `ey` that generate the input sequence arguments `xs` and `ys` of `merge4b`, it must be the case that sequences `map kx es` and `map ky ey` are ascending.

IV. Java Framework Construction

In the previous section we transform a simple merge function expressed in Haskell into a more general function with several higher-order parameters. The generalization steps correspond to points of variability (i.e., hot spots) that can be identified in the scope of the desired software family. This section looks at how we can generate a Java framework from the Haskell program in the previous section. To keep the presentation simple, we do not use Java's generic classes or interfaces.

In an object-oriented framework, we implement a common behavior (frozen spot) by a concrete *template method* in a base class, and we represent a variable aspect of the system (hot spot) by a group of abstract *hook methods*. As we have

constructed it, the merge function embodies the common behaviors of the cosequential processing framework (i.e., the template method). The variable behaviors are abstracted into the parameters of the merge function (`merge4b`) or into the Haskell class `Ord`. The type signature of the `merge4b` function defines the relationships among these "hook methods."

There are two principles for framework construction—unification and separation [9]. The *unification principle* uses inheritance to implement the hot spot subsystem. Both the template methods and hook methods are defined in the same abstract base class. The hook methods are implemented in subclasses of the base class. This effect can be achieved by applying the *Template Method design pattern* to structure the framework [10]. The *separation principle* uses delegation to implement the hot spot subsystem. The template methods are implemented in a concrete client class; the hook methods are defined in a separate abstract class and implemented in its subclasses. The template methods thus delegate work to an instance of the subclass that implements the hook methods. This effect can be achieved by applying the *Strategy design pattern* to structure the framework [10]. We can develop a cosequential processing framework using either principle.

If we choose to use the Template Method pattern to give the program structure, we generate a class similar to the `Coseq` class shown in Fig. 8. We make the merge function the template method `merge()` as shown in Fig. 9. We transform the tail recursive leg of the function definition into a `while` loop that processes the records when neither sequence is empty. We also transform each of the nonrecursive legs (which involve the application of the `foldl` library function) into a simple loop that processes the remaining records in a sequence once the other input sequence has been completely processed. To simplify the presentation, the Java code uses concise names similar to the names used in the Haskell function definitions.

Hot spot #1, the variability of the total ordering used (i.e., the comparison operators), is implemented by class `Ord` in Haskell. The interface `Comparable` from the Java API is a convenient choice as the abstraction for items that can be compared according to a total ordering. Interface `Comparable` has the accessor operation `compareTo(Object)` that returns -1, 0, or 1 depending upon whether the implicit argument of operation is less-than, equal-to, or greater-than the explicit argument. User classes can implement `Comparable` and provide the needed operation. Built-in Java API classes such as `String` and `Integer` already implement the interface.

Hot spot #2, the ability to have complex records in the sequences is implemented by the key extraction function `key` in Haskell function `merge2` and functions `kx` and `ky` in `merge3` and later. Note that these functions return a value of an `Ord` type. In Java, we achieve this by defining an interface `Keyed`, which must be implemented by the elements of the sequences. This interface defines one operation `getKey()` that returns an object of type `Comparable`.

Hot spot #3, which enables the independent variability of the input sequences, is implemented in Haskell by allowing each input sequence to be of a different type, providing different

```
import java.util.*;
abstract public class Coseq
{   public Coseq(Iterator _xs,
                 Iterator _ys, OutSeq _zs)
    {   xs = _xs;  ys = _ys;  zs = _zs;
        xsNotEmpty
          = (xs != null) && xs.hasNext();
        ysNotEmpty
          = (ys != null)& ys.hasNext();
        if (zs == null)
        {   throw new RuntimeException(
            "Output not initialized.");}
    }
    public final void merge()  { ... }
    abstract protected void transLt();//tl
    abstract protected void transEq();//te
    abstract protected void transGt();//tg
    protected void transXsEmpty() //tty
    {   transGt();  }
    protected void transYsEmpty() //ttx
    {   transLt(); }
    protected void advEqXs()      //nex
    {   advXs(); }
    protected void advEqYs()      //ney
    {   advYs(); }
    protected void finish() {}    //res
    protected final Keyed getX() //helpers
    {   return xRec; }
    protected final Keyed getY()
    {   return yRec; }
    protected final void  put(Object r)
    {   zs.put(r); }
    protected final void  advXs()
    {   xRec = null; xKey = null;
        xsNotEmpty = xs.hasNext();
        if (xsNotEmpty)
        {   xRec = (Keyed) xs.next();
            xKey = xRec.getKey();
        }
    }
    protected final void  advYs()
    {   yRec = null;  yKey = null;
        ysNotEmpty = ys.hasNext();
        if (ysNotEmpty)
        {   yRec = (Keyed) ys.next();
            yKey = yRec.getKey();
        }
    }
    private Iterator xs, ys;
    private boolean xsNotEmpty,ysNotEmpty;
    private Keyed xRec, yRec;
    private Comparable xKey, yKey;
    private OutSeq zs;
}
```

Fig. 8. Cosequential framework base class

```
public final void merge() // template
{   advXs(); advYs();
    while(xsNotEmpty && ysNotEmpty)
    {   int cmpxy = xKey.compareTo(yKey);
        if (cmpxy < 0)              // Ord
        {   transLt(); advXs(); }  // tl
        else if (cmpxy == 0)
        {   transEq();              // te
            advEqXs();              // nex
            advEqYs();              // ney
        }
        else
        {   transGt(); advYs(); }  // tg
    }
    while (xsNotEmpty)
    {   transYsEmpty(); advXs(); } // ttx
    while (ysNotEmpty)
    {   transXsEmpty(); advYs(); } // tty
    finish();                       // res
}
```

Fig. 9. Cosequential framework template method

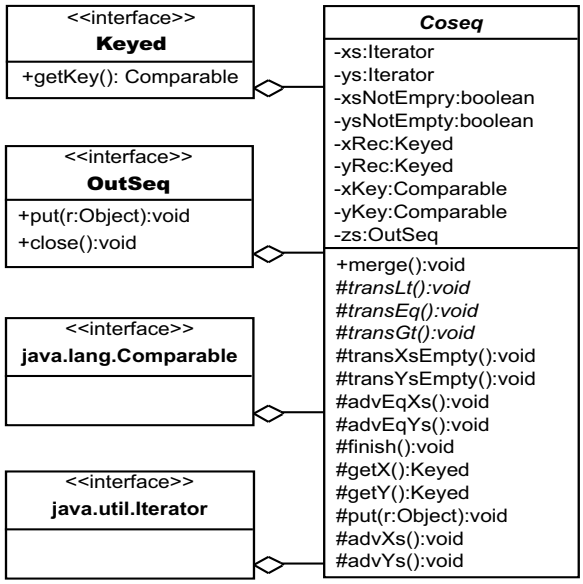

Fig. 10. Cosequential framework classes

key extraction functions for each (i.e., `kx` and `ky` in `merge3` and later), and providing the record-to-record transformation functions `tx` and `ty` to generate the required output record. The `Keyed` abstraction in the Java framework also supports this capability. There can be several classes that implement the interface and support different record structures.

Hot spot #4, which enables the needed variability in the actual transformations carried out on the input sequences to generate the output sequence, is implemented in Haskell by the higher-order function parameters `tl`, `te`, `tg`, `ttx`, `tty`, and `res`. Higher-order parameters `nex` and `ney` also provide the variable ability to step through the input sequences in the case of the keys from both sequences being equal. In Java, these functions become hook methods of the Template Method class for the framework. The overall structure of the Template Method class for the framework is shown in Fig. 8.

Hot spot #5, which enables variable sources and destinations for the values in the sequences, is implemented using lists in Haskell. It is convenient to use Java's built-in interface `java.util.Iterator` to provide the needed abstraction for the input sequences. This interface provides the methods `hasNext()` and `next()` to check for the existence of an additional element and to return that element, respectively. Objects that implement the `Iterator` interface are returned by a number of the built-in collection classes in the Java API. Programmers can also develop their own classes that implement this interface, e.g., classes that provide iterators that read their values from files. The `Coseq` class in Fig. 8 encapsulates the use of the `Iterator` methods inside helper methods `advXs()` and `advYs()` for advancing the "current record" through the input sequences and `getX()` and `getX()` for accessing the current record on each sequence.

Similarly, we introduce a Java interface `OutSeq` that abstracts the output sequence. This interface provides the void methods `put(Object)` and `close()` to enable the appending of new elements and to terminate the sequence when finished. The classes that implement the `OutSeq` interface encapsulate the destination of the elements of the sequence. For example, an `OutSeq`-implementing class might use a file as the destination for the sequence of values output. The `Coseq` class in Fig. 8 encapsulates the use of the `OutSeq` `put()` method inside the helper method `put()`; method `close()` is only used by the client of the framework.

Fig. 10 shows a class diagram giving the relationships among the framework class and interfaces described above. A specific application of the framework must subclass `Coseq` to provide appropriate definitions of the hook methods. It must also provide appropriate implementations of the the `Keyed` and `OutSeq` interfaces. Depending upon the nature of the keys, a new implementation of the `Comparable` interface may also be needed. Similarly, a new implementation of the `Iterator` interface may be needed to provide the input sequences to the `merge()` procedure.

As described here, the framework itself does not include extensive error detection, reporting, or recovery features. These, of course, should be implemented within an application of the framework. A production version of the framework would likely need to expand the functionality of the `Coseq` class to include some error checking features that best fit in the template or helper methods. For example, checking that the input files are properly ordered can be handled in the `advXs` and `advYs` helper methods. This would likely involve the creation of additional hook methods to enable the specific behaviors to vary among applications of the framework.

A cosequential processing framework developed according to the separation principle and Strategy pattern is similar in

many ways to the framework described above. There is a concrete context class (e.g., `CoseqContext`) that includes the template method `merge()` and the concrete helper methods that encapsulate the input and output sequences. The hook methods are collected in an abstract strategy class (e.g., `CoseqStrategy`). The client of an application creates an instance of the context class and supplies it with an instance of the strategy class for the particular application. The instance of the context class delegates the hook method calls to the instance of the strategy class. The methods of the strategy class may call back to the helper methods in the context class as needed.

In the next section we examine a sequential file update application as a specific application of the framework.

V. Sequential File Update Application

The Java framework constructed in the previous section should allow any application in the cosequential processing family to be developed. An interesting example, the sequential file update application, should be easily developed by appropriately extending the framework code. This application involves two files, a master file and a transaction file, as inputs to generate a new master file whose values depend on the initial values of the input files. In our example, the master file records contain the status of ledger account numbers [6]. The record is assumed to contain two fields, the ledger account number as the key and the balance amount in the account. Ledger accounts have different events associated with them which include [6]:

- create an account
- debit or credit an account
- delete an account

For the purpose of illustration, we consider only debits and credits to an account as the allowed transactions. The records in the master file are in ascending order according to keys. The records in the transaction file are ordered according to the keys in the master file. If there are multiple transactions for a single master record, the sequence is the order of their external occurrence. The merging should apply transactions to the master record in order of occurrence. The resultant master file will have master records where [6]:

- the status of some remain unchanged
- some are changed by a single transaction
- the status of others are changed more than once in response to multiple transactions

The application created from the framework consists of a subclass of `Coseq` along with appropriate implementations of the `Iterator` and `OutSeq` interfaces provided by the classes `MasterTrans`, `FileInput`, and `FileOutput`, respectively. It also has an implementation of the `Keyed` interface in the class `MTRecord`. The subclass `MasterTrans`, as shown in Fig. 11, provides functionality to the hook methods `transLt()`, `transEq()`, and `transGt()` declared in class `Coseq`. `MasterTrans()` also overrides the method `advEqXs()`. In the example, the master file record is stored on one line of text and has the format:

```
Key: Balance_amount
```

Similarly, the transaction record is a line with the format:

```
Key: Amount: Transaction_type
```

The class `MTRecord` provides an implementation of the `getKey()` method of the `Keyed` interface, which extracts the account number as the key from each file. It also extracts the amount from the master and transaction records and the transaction type from the transaction record. `FileInput` provides a wrapper to open and read files stored on the disk. Its implementation for the `hasNext()` method checks to see if a record exists in the file while the `next()` method returns the record from the input file. `FileOutput` creates the output file that is the new master file. Its `put()` method is used (via the like-named method in `Coseq`) to append the new master record to the output file and the `close()` method is used by the client to close the output file.

In the `transEq()` method, a transaction is applied to the master record to create a new master record that is stored internally. Further transactions are applied to the internal master record that changes with each transaction. This continues until all transactions corresponding to the master record are completed. In `transLt()`, when the key of the transaction record is greater than the master record key, the internal master record is sent to the output if not null. If it is null, then the master record is output. Method `transGt()` generates an error message if the key of the master record is greater than the corresponding key of the transaction record, that is, there is some transaction not corresponding to an existing account.

In this example, most of the error handling, such as checks on the validity of records, is relegated to the methods of class `MTRecord` and other classes supplied by the application programmers. At the cost of some extra complexity, the framework could be generalized to add other template and hook methods to organize the approach to error handling and recovery. For example, there potentially could be additional hook methods that are called by `advXs()` or `advYs()` to check the validity of the input records.

VI. Related Work

Framework design, and program family design in general, is all about developing the right abstractions. Roberts and Johnson note that most programmers "develop abstractions by generalizing from concrete examples." [18] They warn that an "attempt to determine the correct abstractions on paper without actually developing a running system is doomed to failure." In their Patterns for Evolving Frameworks [18], they propose that framework designers first develop three example systems (at least prototypes) and then use those three examples to develop the needed general abstractions for the framework. They argue that fewer applications probably do not provide a sufficient view of the requirements for the family and more applications may make framework development too time consuming and costly. They view frameworks as software systems that grow organically in response to changing requirements or the needs

```
import java.util.*;
public class MasterTrans extends Coseq
{   public MasterTrans (Iterator xs,
             Iterator ys, OutSeq zs)
    {   super(xs,ys,zs);
        curRec = false; curAcc = null;
        curBal = 0.0;
    }
    protected void transLt() // mst < tr
    {   if (curRec)
        {   put(new MTRecord
                     (curAcc,curBal));
            curRec = false;
        }
        else
        {   put(getX()); }
    }
    protected void transEq() // mst = tr
    {   MTRecord xRc = (MTRecord)getX();
        MTRecord yRc = (MTRecord)getY();
        if (!curRec)
        {   curAcc =
                (Integer)(xRc.getKey());
            curBal = xRc.getAmount();
            curRec = true;
        }
        double amt = yRc.getAmount();
        int    trT = yRc.getTransType();
        if (trT == MTRecord.DEBIT)
        {   curBal = curBal - amt; }
        else if (trT == MTRecord.CREDIT)
        {   curBal = curBal + amt; }
        else
        {   putErr(
            "Invalid trans type skipped: "
            + yRc);
        }
    }
    protected void transGt() // mst > tr
    {   putErr(
        "Invalid acct # on transaction: "
         + curAcc);
    }
    protected void advEqXs() { }

    private void putErr(String e)
    {   System.out.println(e); }
    private boolean curRec;
    private Integer curAcc;
    private double  curBal;
}
```

Fig. 11. Master-Transaction File Update Application

of different customers. As knowledge of the domain's requirements grows, the design is refactored and evolved over time into a framework.

Schmid takes a more proactive stance in his Systematic Generalization approach to framework design. This approach emphasizes that designers should "preplan evolution and generalize as much as possible a priori." [19] The approach starts by building an application model for a typical fixed application in the family. The approach then seeks explicitly to generalize this design into a framework. It conducts a systematic analysis of the application, applying domain knowledge to determine the likely points of variation (i.e., the hot spots) among the family members. Using this analysis, the approach seeks to generalize the design around these hot spots by applying systematic transformations of the design that are driven by the analysis of the hot spot. Schmid's approach has much in common with the commonality/variability analysis techniques for developing software product lines, another form of program family [1].

Carey and Carlson's framework process patterns also seek to design the needed generality into a framework from the beginning [2]. They do, however, offer some cautions. They warn designers to "beware of extreme requirements that unnecessarily increase framework complexity" (in their discussion of the requirements gathering pattern Tor's Second Cousin). They suggest that "in some cases a contingency is mentioned, examined, and not handled" within a framework's design (in their discussion of the Pass the Buck design pattern).

The work described in this paper takes the same stance as Schmid, seeking to build the needed generality into the design from the beginning, but also taking note of Carey and Carlson's cautions. While Schmid generalizes the concrete class structure of an object-oriented program, the work in this paper generalizes the structure of the functions in an executable specification written in the functional language Haskell. Like Schmid's approach, this work seeks to separate the various dimensions of variability and deal with each one separately, introducing the needed new abstractions into the program systematically. Unlike Schmid's work, each function presented in this paper is executable when appropriate arguments are supplied to the higher-order functions. Each of the merge functions can be executed using the Hugs interpreter [13] for the Haskell language. The ability to compile and run the Haskell programs at each step enables the designers to debug their specifications, explore the functionality of the framework, and make appropriate changes to the specifications.

In addition to the ability to execute the framework models, the authors chose to explore the use of Haskell for framework design because of its conciseness and of its amenability to mathematical manipulation [12], [23] and refactoring [14], [24]. Haskell is quite expressive, capable of defining relatively powerful programs in relatively few symbols and lines of code. Because Haskell is relatively close to normal mathematical notation in its syntax and semantics, it is convenient to state and possible to prove mathematical properties such at the equivalence properties stated for each transformation. How-

ever, functional programs have a few disadvantages. Haskell is unfamiliar to most programmers. Also the architectural design of Haskell programs must be understood using a text-based language. In Schmid's approach, the framework's architecture can be readily shown more visually using class diagrams in the Unified Modeling Language (UML).

Of course, other languages or even mathematical functions could be used for deriving frameworks by generalization of functions. For example, Tadepalli and Cunningham [21] and Cunningham, Liu, and Zhang [4] use similar techniques to generalize Java functions into frameworks.

The work described in this paper is also close in spirit to the approach of Cortes, Fontoura, and Lucena in [3]. Their approach evolves an object-oriented framework through a sequence of refactoring and unification transformations. A refactoring changes the structure of a framework, while preserving its behavior. It is often carried out to increase the flexibility of the framework. A unification seeks to add new functionality into the design, and, hence, does not preserve behavior. These involve identification of template and hook methods and their realization as small frameworks or framelets [17]. Their work is aimed toward systematically "restructuring the framework hot-spots during evolution" of the framework. The systematic generalization approach of Schmid and the function generalization approach in this paper are aimed more at a priori identification of hot spots and transformation of a prototype design into a framework.

VII. Conclusion

This paper illustrates the technique of framework design by function generalization. It begins with a simple Haskell program to merge two ascending lists of integers into third ascending list of integers containing all integers in the first two sequences. 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 [8]. Finally, the program is transformed into a Java framework with that capability. To illustrate the framework, the paper constructs a simple master/transaction file update program that extends the framework code.

Although some members of the cosequential processing family can be rather complicated, it has the characteristic that the primary driver for the algorithm can be concisely stated as a simple loop (i.e., recursive function). The function generalization technique worked reasonably well for this family. More examples should be studied to determine how broadly the technique can be applied effectively. Also the rules for generalization should be formalized more thoroughly than they are presented in this paper.

VIII. Acknowledgments

This work was supported, in part, by a grant from Acxiom Corporation titled "The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)." The authors thank Yi Liu for her useful suggestions concerning this work and for drawing Fig. 10.

References

[1] M. Ardis, N. Daley, D. Hoffman, and D. Weiss, "Software product lines: A case study," *Software—Experience and Practice*, vol. 30, pp. 825–847, 2000.

[2] J. Carey and B. Carlson, *Framework Process Patterns: Lessons Learned Developing Application Frameworks*. Addison-Wesley, 2002.

[3] M. Cortes, M. Fontoura, and C. Lucena, "Using refactoring and unification rules to assist framework evolution," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 6, November 2003.

[4] H. C. Cunningham, Y. Liu, and C. Zhang, "Using classic problems to teach Java framework design," *Science of Computer Programming*, in press.

[5] E. W. Dijkstra, "Updating a sequential file," in *A Discipline of Programming*. Prentice Hall, 1976, ch. 15, pp. 117–122.

[6] B. Dwyer, "One more time—How to update a master file," *Communications of the ACM*, vol. 81, no. 1, pp. 3–8, January 1981.

[7] M. E. Fayad, D. C. Schmidt, and R. E. Johnson, "Application frameworks," in *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, M. E. Fayad, D. C. Schmidt, and R. E. Johnson, Eds. Wiley, 1999.

[8] M. J. Folk, B. Zoellick, and G. Riccardi, *File Structures: An Object-Oriented Approach with C++*, 3rd ed. Addison-Wesley, 1998.

[9] M. Fontoura, W. Pree, and B. Rumpe, *The UML Profile for Framework Architectures*. Addison-Wesley, 2002.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, 3rd ed. Wiley, 2004.

[12] R. Hoogerwoord, "The design of functional programs: A calculational approach," Ph.D. dissertation, Technical University of Eindhoven, The Netherlands, 1989.

[13] Hugs Project, "Hugs online," `http://www.haskell.org/hugs`, 2004.

[14] H. Li, C. Reinke, and S. Thompson, "Tool support for refactoring functional programs," in *Proceedings of the ACM SIGPLAN Haskell Workshop*, Uppsala, Sweden, August 2003.

[15] D. L. Parnas, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, pp. 1–9, March 1976.

[16] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[17] W. Pree and K. Koskimies, "Framelets—Small is beautiful," in *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, M. E. Fayad, D. C. Schmidt, and R. E. Johnson, Eds. Wiley, 1999, pp. 411–414.

[18] D. Roberts and R. Johnson, "Patterns for evolving frameworks," in *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, and F. Buschmann, Eds. Addison-Wesley, 1998, pp. 471–486.

[19] H. A. Schmid, "Framework design by systematic generalization," in *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, M. E. Fayad, D. C. Schmidt, and R. E. Johnson, Eds. Wiley, 1999, pp. 353–378.

[20] Y. Smaragdakis and B. McNamara, "Bridging functional and object-oriented programming," College of Computing, Georgia Institute of Technology, Tech. Rep. 00-27, 2000.

[21] P. Tadepalli and H. C. Cunningham, "Using function generalization with Java to design a cosequential framework," in *Proceedings of the Conference on Applied Research in Information Technology*. Acxiom Laboratory for Applied Research, February 2005, pp. 95–101.

[22] S. Thompson, "Higher-order + polymorphic = reusable," Computing Laboratory, University of Kent, Tech. Rep., May 1997.

[23] ——, *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999.

[24] S. Thompson and C. Reinke, "A case study in refactoring functional programs," in *Proceedings of the 7th Brazilian Symposium on Programming Languages*, May 2003.

[25] E. Wallingford, "Roundabout: A pattern language for recursive programming," in *Proceedings of the Fourth Pattern Languages of Programs Conference*, Allerton, Illinois, 1997.

[26] ——, "Envoy: A pattern language for maintaining state," in *Proceedings of the Sixth Pattern Languages of Programs Conference*, Allerton, Illinois, 1999.