Framework Design Using Function Generalization: A Binary Tree Traversal Case Study

H. Conrad Cunningham
Computer & Information Science
University of Mississippi
University, MS 38677 USA
1-662-915-5358
cunningham@cs.olemiss.edu

Yi Liu
Electrical Engr. & Computer Science
South Dakota State University
Brookings, SD 57007 USA
1-605-688-5280
yi.liu@sdstate.edu

Pallavi Tadepalli
Computer & Information Science
University of Mississippi
University, MS 38677 USA
1-501-252-0868
pallavi@cs.olemiss.edu

ABSTRACT
A software framework is a technology that enables software reuse, potentially yielding rich dividends but requiring significant long-term investment. However, a framework is not a panacea because it is more difficult to design than a single application. Systematic approaches seek to make framework design more convenient and less error-prone. This paper illustrates the function generalization approach to framework design by developing a framework for binary tree traversals. A binary tree traversal is a well-known algorithm, which makes it an excellent example. The approach involves a systematic process for generalizing a fixed application expressed as Haskell functions to produce a set of functions that precisely describe the generalized application. This generalized application encompasses various common and variable aspects of a family of applications. By using design patterns as a guide, the resulting set of functions can be converted to a Java framework.

Categories and Subject Descriptors
D.2.13 [**Software Engineering**]: Reusable Software – *domain engineering, reusable libraries, reuse models.*

General Terms
Design.

Keywords
Software framework, function generalization, frozen spot, hot spot, tree traversal.

1. INTRODUCTION
David Parnas advocated in his classic paper [9] that a "software designer should be aware that he is not designing a single program but a family of programs". He defines a *program family* as "a set of programs with so many common properties that it is worthwhile to study the set as a group" [8]. His concept of program family has been incorporated into various software structures that support reuse. A software framework is one such structure. A *software framework* is "a generic application that allows different applications to be created from a family of applications" [12]. The properties common to all family members are called *frozen spots*; these embody the overall structure of the framework and are fixed and reusable for all the family members. The properties that vary among family members are called *hot spots*; the implementations of these can vary among family members to allow the framework to be customized to a specific member of the family [12].

ACM SE'06, March 10-12, 2006, Melbourne, Florida, USA

Because of the need for flexibility and variability in a framework, framework design is much more complicated than application design. Framework design requires a systematic approach. Typically, the development of a framework is a process of examining existing designs for family members, identifying the frozen spots and hot spots of the family, and generalizing the program structure to enable reuse of the code for frozen spots and use of different implementations for each hot spot. This generalization may be done in an informal, organic manner as codified by Roberts and Johnson in the Patterns for Evolving Frameworks [11] or it may be done using systematic techniques such as systematic generalization [12] and function generalization [2] as described in Section 2.

This paper illustrates the function generalization approach [2] to framework design using a binary tree traversal framework as an example [1]. Section 2 introduces Schmid's systematic generalization approach and the function generalization approach. Section 3 applies the function generalization approach to design a binary tree traversal framework and Section 4 looks at how to convert the framework to Java. Section 5 discusses this in relationship to other work and Section 6 concludes the paper.

2. GENERALIZATION APPROACHES
Schmid's *systematic generalization* methodology is one technique [12] that seeks to identify the hot spots a priori and construct a framework systematically. This methodology identifies the following steps for construction of a framework [12]:

- 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 for a specific application within the family. Once a complete model exists, the framework designer analyzes the model and the domain to discover and specify the hot spots. The hot spot's

features are accessed through the common interface of the abstract class. However, the design of the hot spot subsystem enables different concrete subclasses of the base class to be used to provide the variant behaviors.

Function generalization [2] is another systematic approach. Instead of generalizing the class structure for an application design as Schmid's methodology does, the function generalization approach generalizes the functional structure of an executable specification to produce a generic application. It 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 [10]. 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.

There are several advantages of using Haskell to represent the specifications. (1) Haskell functions are polymorphic. They can be defined to take parameters from a range of related data types. (2) 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. (3) Haskell programs tend to be concise. (4) Haskell programs can readily be manipulated mathematically using the language itself, which makes it easier to carry out the generalizing transformations and define the hot spot subsystems precisely.

Polymorphism and higher-order functions are important because they promote the reuse of software. 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 as needed [2].

In the next section, we use a classic problem, binary tree traversal, as a case study to specify how to apply the function generalization approach to the framework design.

3. GENERALIZING TREE TRAVERSALS

A binary tree is a hierarchical data type whose instances are either empty (having no nodes) or nonempty (having a root node, a left binary subtree, and a right binary subtree). Each nonempty node holds a single value of some type. In Haskell, a binary tree can be represented by a data type `BinTree` as follows:

```
data BinTree a = Nil |
                 Node(BinTree a) a BinTree a)
```

`Nil` represents the empty tree and `Node` represents a node of a nonempty tree that holds a value of polymorphic type `a` internally.

A traversal is a systematic technique for "visiting" all the nodes in a tree. One common traversal technique for a binary tree is the preorder traversal. This is a depth-first traversal, where it accesses a node's children before it accesses the node's siblings. This traversal can be expressed by a recursive procedure as follows:

```
procedure preorder(t)
{ if t null, then return;
   perform visit action for root node of tree t;
   preorder(left subtree of t);
   preorder(right subtree of t);
}
```

Figure 1 shows the `preorder` traversal function in Haskell. The first line of the definition gives the function's signature. It takes a single argument of type `BinTree` and returns a list of the values held by the nodes. The function definition has two legs. The first leg defines the function when the argument is `Nil`, in which case the function returns the empty list. The second leg defines the function when the argument is nonempty, that is, a `Node`. In this case, the function returns a list consisting of the node's value at the head of the list, followed by the result of appending the lists returned by applying the function recursively to the left and right subtrees, respectively.

```
preorder :: BinTree a -> [a]
preorder Nil  = []
preorder (Node l v r)
     = v : preorder l ++ preorder r
```

Figure 1. Preorder Traversal in Haskell

To build a software framework for binary tree traversals, we begin with the simple preorder operation given in Figure 1 and consider the domain of the family and identify the frozen and hot spots.

What is the scope of the family of binary tree traversals? The family should include at least the standard kinds of depth-first traversals (e.g., preorder, postorder, and in-order) and allow flexible visit actions on the nodes. In general, the visit action will be a function on the node's attributes and on the accumulated state of the traversal computed along the sequence of all the nodes accessed to that point in the computation. The framework should enable traversal orders other than the depth-first. The framework should also support binary search trees, but it is not necessary that it support multiway trees or general graphs.

What, then, are the commonalities—frozen spots—that all members of the family exhibit? Considering the scope and examining the prototype application, we choose the following frozen spots:

1. The structure of the tree (e.g., the `BinTree` data type) cannot be redefined by clients of the framework.
2. A traversal accesses every element of the tree once, unless the computation determines that it can stop before it completes the traversal.
3. A traversal performs one or more visit actions associated with an access to an element of the tree.

What are the variabilities—the hot spots—that exist among members of the family of binary tree traversals? Again, considering the scope and examining the prototype application, we identify the primary hot spots to be the following:

1. Variability in the visit operation's action. It should be a function of the current node's value and the accumulated result of the visits to the previous nodes in the traversal.
2. Variability in ordering of the visit action with respect to subtree traversals. That is, the client should be able to select preorder, postorder, inorder, etc.

3. Variability in the tree navigation technique. That is, the client should be able to select node access orders other than left-to-right, depth-first, total traversals.

Now, given these variabilities, we examine how they can be introduced into a framework by generalizing the Haskell program. We seek to define the expected behaviors of the hot spots very precisely using the Haskell functions.

3.1 Generalizing the Visit Action

Hot spot #1 requires making the visit action a feature that can be customized by the client of the framework to meet the specific application's needs. The visit action, in general, varies from one application to another. The fact that there are visit actions associated with the access to an element is a common behavior of the framework. The visit action itself is the variable behavior that is to be captured in a hot spot subsystem.

The generalized action preorder traversal `gaPre` requires four aspects of the preorder function to be generalized: (1) the addition of an explicit "state" parameter to represent the accumulated result of carrying out the traversal, (2) a state update function `us`, (3) an initial state `is`, represented by the nil list `[]` in the original function, and (4) a Nil-state function `un` to extract the final result from the accumulated state.

```
gaPre :: (a -> b -> b) -> (b -> b) -> b ->
         BinTree a -> b
gaPre us un is t = po t is
  where po Nil s          = un s
        po (Node l v r) s = po r (po l(us v s))
```

Figure 2. Generalized Visit Action

Figure 2 shows the resulting generalized program. The identity

```
gaPre (\x y -> y ++ [x]) id [] t = preorder t
```

holds, where `(\x y -> y ++ [x])` denotes an anonymous function that takes two arguments `x` and `y` and appends the element `x` at the end of list `y`.

3.2 Generalizing the Visit Order

Hot spot #2 requires making the "order" of the visit actions a feature that can be customized for a specific application. That is, it must be possible to vary the order of a node's visit action with respect to the traversals of its children. The framework should support preorder, postorder, and inorder traversals and perhaps combinations of those. A good generalization of the three standard traversals is one that potentially performs a visit action on the node at any of three different points—on first arrival (i.e., a "left" visit), between the subtree traversals (i.e., a "bottom" visit), and just before departure from the node (i.e., a "right" visit). This is sometimes called an Euler tour traversal [6].

To generalize the visit order of `gaPre`, we replace the update state function with three update state functions, one for each of the three visit points of the Euler traversal. The resulting generalized function is `gvTraverse` shown in Figure 3. The identity

```
gvTraverse us id id un is t = gaPre us un is t
```

holds where `id` represents the identity function (i.e., a single argument function that returns its argument unmodified).

```
gvTraverse :: (a -> b -> b) -> (a -> b -> b) ->
              (a -> b -> b) -> (b -> b) -> b ->
              BinTree a -> b
gvTraverse ul ub ur un is t = tr t is
   where tr Nil s = un s
         tr (Node l v r) s
           = ur v (tr r (ub v (tr l (ul v s))))
```

Figure 3. Generalized Visit Order

3.3 Generalizing the Tree Navigation

Hot spot #3 requires making the navigation of the tree structure a feature that can be customized to meet the needs of a specific application. In particular, it should enable variability in the order that nodes are accessed. For example, the framework should support breadth-first traversals as well as depth-first traversals.

The generalization step here is to separate the tree navigation from the sequential application of the visit actions. The resulting function `traverse` is shown in Figure 4. Its first parameter, `nav`, is a function similar to `gvTraverse`. Different traversal orders can be achieved by providing different implementations of `nav`. Instead of computing an updated state for each visit action, `nav` generates a list of functions that, when applied incrementally beginning with the initial state of the traversal, produces the desired result. The values of the nodes are bound into these functions at the appropriate positions in the list. The computation of the result is carried out by function `compose`, which uses a `fold` operator to compose the state update functions corresponding to visits to the nodes. The computation described by the list must preserve frozen spot #2, that is, include one access of every node of the tree.

```
traverse :: ((a -> b -> b) -> (a -> b -> b) ->
  (a -> b -> b) -> (b -> b) -> BinTree a ->
  [(b -> b)]) ->
  (a -> b -> b) -> (a -> b -> b)-> (a -> b -> b)
  -> (b -> b) -> b -> BinTree a -> b
traverse nav ua ub ud un is t
 = compose (nav ua ub ud un t) is
   where compose fs s = foldl (flip (.)) id fs s

euler:: (a -> b -> b) -> (a -> b -> b) ->
  (a -> b -> b) -> (b -> b) -> BinTree a ->
  [(b -> b)]
euler ua ub ud un t = doEuler t
  where doEuler Nil = [un]
        doEuler (Node l v r)
          = [(ua v)] ++ doEuler l ++ [(ub v)]
            ++ doEuler r ++ [(ud v)]
```

Figure 4. Generalized Tree Navigation

Function `euler` is an implementation of `nav` that does the navigation needed for an Euler tour traversal. Note that the following identity holds:

```
traverse euler ua ub ud un is t
 = gvTraverse ua ub ud un is t
```

The real-world framework applications are often implemented in object-oriented languages such as Java. The next section examines how to take the resulting generalized functions to define the binary tree traversal framework in Java.

4. TREE TRAVERSAL FRAMEWORK

Framework design and implementation often use design patterns [4] to help achieve the needed flexibility. Behaviors associated with

frozen spots are often captured in concrete *template methods* in a base class; behaviors associated with hot spots are typically assigned to abstract *hook* methods [3]. A difficulty in framework design is the specification of the behaviors of the hook methods. The Haskell functions above enable us to define those behaviors with precision and to test the behaviors.

In the design of the binary tree traversal framework in Java, we first apply the *Composite design pattern* to implement a structure similar to the Haskell data type `BinTree`. The Composite pattern "lets clients treat individual objects and compositions of objects uniformly" [4]. Figure 5 shows the structure of the resulting `BinTree` class hierarchy in UML. Class `BinTree` has the Component base-class role while subclass `Node` has the Composite role, and subclass `Nil` has the Leaf role. `Nil` is also implemented according to the *Singleton pattern* [4], which guarantees exactly one instance exists. Figure 6 sketches the Java code for the `BinTree` class hierarchy.

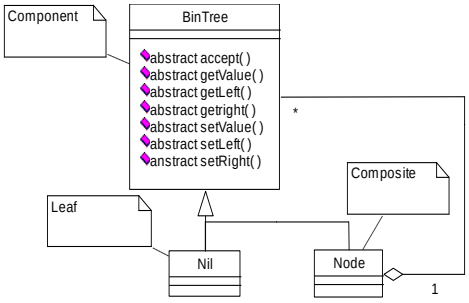

Figure 5. Binary Tree Using Composite Pattern

```
abstract public class BinTree
{ public void setValue(Object v) { } // mutators
  public void setLeft(BinTree l) { } // default
  public void setRight(BinTree r) { }
  abstract public void preorder(); // traversal
  public Object getValue() { return null; }
                                    // accessors
  public BinTree getLeft() { return null; }
                                     // default
  public BinTree getRight() { return null; }
}
public class Node extends BinTree
{ …
}
public class Nil extends BinTree
{ private Nil() { } // require use of getNil()
  public void preorder() { }; // traversal
  static public BinTree getNil()
  { return theNil; } // Singleton
  static public BinTree theNil = new Nil();
}
```

Figure 6. Binary Tree Class Hierarchy

The *Strategy pattern* is useful in implementing hot spots #1 and #2 in Java. It enables us to define a family of algorithms having a common interface, encapsulate each algorithm in a separate class, and make the algorithms interchangeable. In the Haskell programs, the algorithms we wish to encapsulate are the higher-order parameters to the traversal function. These parameters are functions that update the traversal state at the various visit points.

We introduce hot spots #1 and #2 into the Java traversal program by generalizing the classes in the `BinTree` hierarchy to have the Context role in the *Strategy pattern*. We generalize the `BinTree` method `preorder` to be a template method and define a Strategy interface `EulerStrategy` for objects that implement the hook methods. We generalize the behavior of the method `preorder` and replace it by a template method `traverse` that takes a state descriptor and a Strategy object. This method encodes the common features of all Euler tour traversals but delegates the visit actions to the hook methods defined on the Strategy object. Figure 7 sketches the code for a general Euler tour traversal with generalized visit actions.

```
abstract public class BinTree
{ ...
  abstract public Object traverse(Object ts,
EulerStrategy v);
  ...
}
public class Node extends BinTree
{ ...
  public Object traverse(Object ts,
        EulerStrategy v) // traversal
  { ts = v.visitLeft(ts,this);
                // upon arrival from above
    ts = left.traverse(ts,v);
    ts = v.visitBottom(ts,this);
                // upon return from left
    ts = right.traverse(ts,v);
    ts = v.visitRight(ts,this);
                // upon completion
    return ts;
  }
  ...
}
public class Nil extends BinTree
{ ...
  public Object traverse(Object ts,
        EulerStrategy v)
  { return v.visitNil(ts,this); }
  ...
}
public interface EulerStrategy
{  abstract public Object
       visitLeft(Object ts, BinTree t);
   abstract public Object
       visitBottom(Object ts, BinTree t);
   abstract public Object
       visitRight(Object ts, BinTree t);
   abstract public Object
       visitNil(Object ts, BinTree t);
}
```

Figure 7. Binary Tree with Euler Traversal

Hot spot #3 requires making the navigation of the tree structure a feature that can be customized to meet the needs of a specific application. The Haskell program achieves this by separating the navigation function from the incremental computation of the state changes along the traversal path. Thus, new navigation techniques can be incorporated into the Haskell program without modifying the basic structure of the binary tree or modifying the nature of the traversal function.

We can achieve a similar effect in the Java framework by applying the *Visitor pattern* [4]. The intent of the Visitor design pattern is to enable the functionality of an object structure to be extended without modifying the structure's code. It is quite compatible with object structures designed according to the Composite pattern.

In the binary tree traversal framework design, we assign the `BinTree` class hierarchy the role of the Element hierarchy in the Visitor pattern's description [4], and we introduce a `BinTreeVisitor` interface to take on the role of the Visitor class in the description. We also generalize the `traverse` method of the Java `BinTree` hierarchy and replace it by the `accept` method for the Visitor pattern. The `accept` method is similar to the `traverse` function in the Haskell code for hot spot #3. The `BinTreeVisitor` object plays a role similar to the navigation function `nav` in the Haskell code. The functionality of the Haskell `compose` function is implicit because of the mutable state of the Java classes.

The `accept` method of a `BinTree` element takes a `BinTreeVisitor` object and delegates the traversal back to an appropriate method of that Visitor object. This method applies the appropriate binary tree visit actions and navigates through the tree as needed for the application. The `BinTreeVisitor` interface has methods named `visit` with overloaded implementations for each subclass in the `BinTree` hierarchy. The constraint on the framework given by frozen spot #2 (i.e., to access each node once) becomes a requirement upon the designer of the visitor classes that implement `BinTreeVisitor`. Figure 8 illustrates the class structure of a traversal program based on the *Visitor design pattern.* Figure 9 shows the Java code for the traversal program.

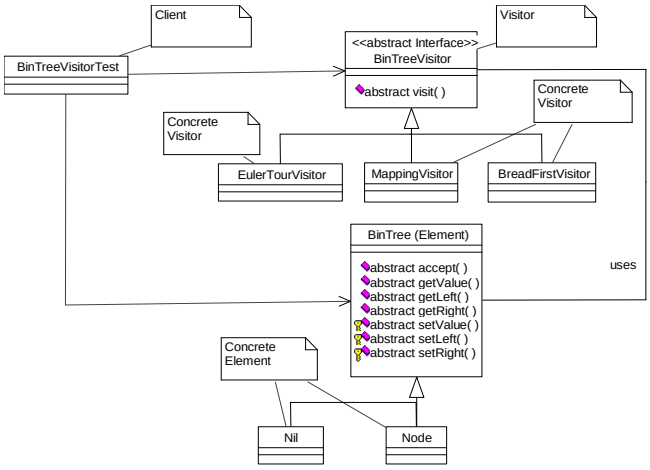

Figure 8. Binary Tree Visitor Framework

The Visitor framework has two levels. The upper level of the framework is characterized by the Visitor pattern as described above. However, the specific designs for the Visitor objects themselves may be small frameworks. Consider a program to carry out an Euler tour traversal. We can choose to design a concrete class `EulerTourVisitor` that implements the `BinTreeVisitor` interface. This class delegates the specific traversal visit actions to a Strategy object of type `EulerStrategy`. Figure 10 illustrates the class structure of this lower-level design. Figure 11 shows its implementation in Java.

The binary tree traversal framework is quite general. It supports a large set of binary tree algorithms. Although the initial prototype application is a depth-first traversal, the framework supports breadth-first traversals as well. This use of the framework requires the implementation of a `BinTreeVisitor` class that navigates through the tree appropriately. For example, this class could maintain a queue of nodes to visit next and enqueue the children of each node its visits. The result is that the traversal proceeds through the tree level by level. Other interesting applications of the framework might be to use it to implement programs for binary search trees.

The next section discusses the function generalization approach in the context of some of the related work.

```
abstract public class BinTree
{ public void setValue(Object v) { } // mutators
  public void setLeft(BinTree l) { } // default
  public void setRight(BinTree r) { }
  abstract public void accept(BinTreeVisitor v);
                                // accept Visitor
  public Object getValue() { return null; }
                                  // accessors
  public BinTree getLeft() { return null; }
                                // default
  public BinTree getRight() { return null; }
}
public class Node extends BinTree
{ public Node(Object v, BinTree l, BinTree r)
  { value = v; left = l; right = r; }
  public void setValue(Object v) { value = v; }
  public void setLeft(BinTree l) { left = l; }
  public void setRight(BinTree r) { right = r; }
                      // accept a Visitor object
  public void accept(BinTreeVisitor v)
  { v.visit(this); }
  public Object getValue() { return value; }
  public BinTree getLeft() { return left; }
  public BinTree getRight() { return right; }
  private Object value; // instance data
  private BinTree left, right;
}
public class Nil extends BinTree
{ private Nil() { }
          // private to require use of getNil()
  // accept a Visitor object
  public void accept(BinTreeVisitor v)
  { v.visit(this); }
  static public BinTree getNil()
  { return theNil; } // Singleton
  static public BinTree theNil = new Nil();
}
public interface BinTreeVisitor
{ abstract void visit(Node t);
  abstract void visit(Nil t);
}
```

Figure 9. Binary Tree Using Visitor Pattern

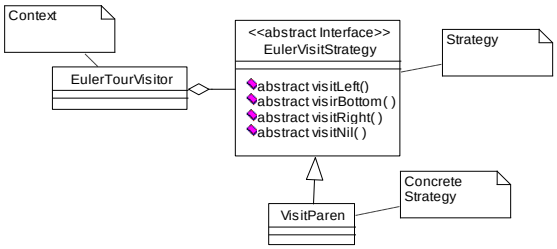

Figure 10. Euler Tour Traversal Visitor Framework

```
public class EulerTourVisitor
       implements BinTreeVisitor
{ public EulerTourVisitor(EulerStrategy es,
                Object ts)
  { this.es = es; this.ts = ts; }
  public void setVisitStrategy(EulerStrategy es)
  { this.es = es; }
  public void setResult(Object r) { ts = r; }
  public void visit(Node t)
             // Visitor hook implementations
  { ts = es.visitLeft(ts,t);
             // upon first arrival from above
    t.getLeft().accept(this);
    ts = es.visitBottom(ts,t);
              // upon return from left
    t.getRight().accept(this);
    ts = es.visitRight(ts,t);
              // upon completion of this node
  }
  public void visit(Nil t)
  { ts = es.visitNil(ts,t); }
  public Object getResult(){ return ts; }
  private EulerStrategy es;
           // encapsulates state changing ops
  private Object ts; // traversal state
}
public interface EulerStrategy
{ . . . . } //Listed in Figure 8
```

Figure 11. Euler Tour Traversal Visitor

5. DISCUSSION

Framework design is all about developing the right abstractions. Roberts and Johnson note that most programmers "develop abstractions by generalizing from concrete examples" [11]. 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 [11], they propose that framework designers first develop three example systems and then use those three examples to develop the needed general abstractions for the framework. They view frameworks as software systems that grow organically in response to changing requirements or the needs 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" [12]. 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.

The function generalization approach takes the same stance as Schmid, seeking to build the needed generality into the design from the beginning. 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. The ability to execute 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.

The authors began to explore this approach because of frustration with the difficulty of specifying the behaviors of a cosequential processing framework [2], in particular the behaviors of its hot spot abstractions. When informal framework design approaches are used, it is difficult to be precise and unambiguous in specifying the hot spot subsystems, particularly the methods that modify the state of an object. When more formal specifications such as method preconditions and postconditions and class invariants are used, a large intellectual leap is sometimes needed to go from the concrete examples motivating the design to the complete framework. It seems that the designer must sometimes be a magician who can pull a large rabbit out of his or her hat.

An incremental approach that seeks to construct a framework using a sequence of small, precise mathematical generalizations of the concrete example makes the process more intellectually manageable. Specifying each generalization as a set of new higher order arguments (i.e., hook methods) to a function (i.e., template method) keeps each "rabbit" smaller; the resulting hooks tend to be more precisely described. The use of a purely functional language like Haskell restricts the insidious effects of an implicit state that can be modified in a hot spot. Purely functional languages require that all state manipulated by the program to be explicit, hence, making the specifications more precise and unambiguous. The ability to test the specifications by executing the functional program is a bonus.

This case study uses the functional language Haskell, which is one of the primary languages used by the functional programming and type system research communities. Haskell researchers are generating many interesting ideas on generic programming that are relevant to framework design in object-oriented languages. For example, in a paper on *datatype-generic programming* (DGP) [5], Gibbons argues that a Haskell-based DGP notation "is sufficient to express essentially all the genericity found in" the popular C++ Standard Template Library (STL) [7]. He further notes "it could be argued that many of the" design patterns in [4] "are idioms for mimicking DGP in languages that do not properly support such a feature." Of interest for the work in this paper, Gibbons observes that the Visitor pattern [4] is essentially a datatype-generic `fold` operation in a functional program. As another example, consider the technique called *strategic programming.* In [13], Visser defines a set of "visitor combinator" classes that can be composed to create arbitrary traversals of tree structures. The Java combinators he presents are inspired by similar combinators defined in Haskell.

In future work on function generalization, the research on functional generic programming, in particular the efforts to capture design patterns as generic programs, should be examined more thoroughly for its usefulness as a guide to framework design. That research may suggest better functional programming idioms for realizing the desired hot spot generalizations. Similarly, this study may yield more systematic techniques for translating the Haskell programs to Java classes. Judicious use of other Haskell features

such as type classes and modules may also help clarify some of the design issues and make the specifications more readable.

A shortcoming of the work on function generalization reported in this paper and in [2] is that only small, well-known problems have been tackled with the approach. This initial focus is intentional because part of the motivation is to use "classic problems" as an aid in teaching framework design [1]. Future work should explore how well the approach scales up to larger, less known design problems.

6. CONCLUSION

This paper describes the process of framework design using function generalization and illustrates the process using binary tree traversal as an example. First, the approach analyzes the program family to determine the features that are common to all members (i.e., the frozen spots) and those that vary among the members (i.e., the hot spots). Next, the approach incrementally generalizes the functional model of the family to design a software framework that can be customized at the hot spots. When the generalization steps are complete, the functional model is then converted to Java classes. The approach structures the Java code using design patterns such as Strategy and Visitor. The function generalization approach potentially results in framework designs in which the behaviors of the hot spot abstractions are specified more precisely and less ambiguously than typically is the case in other common approaches to framework design.

7. ACKNOWLEDGEMENTS

The authors thank reviewer #1 for his or her comments that led us to expand the discussion of the related and future work.

8. REFERENCES

[1] H. C. Cunningham, Y. Liu, and C. Zhang. "Using Classic Problems to Teach Java Framework Design," *Science of Computer Programming*, Special Issue on Principles and Practice of Programming in Java (PPPJ 2004), Vol. 59, No. 1-2, pp. 147-169, January 2006.

[2] H. C. Cunningham and P. Tadepalli. "Using Function Generalization to Design a Cosequential Processing Framework," In *Proceedings of the 39th Hawaii International Conference on System Sciences*, 10 pages, IEEE, January 2006.

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

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

[5] J. Gibbons. "Patterns in Datatype-Generic Programming," In *Declarative Programming in the Context of Object-oriented Languages*, Uppsala, Sweden, August 2003.

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

[7] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Second edition, Addison-Wesley, 2001.

[8] D. L. Parnas. "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, Vol. SE-2, pp. 1-9, March 1976.

[9] D. L. Parnas. "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, Vol. SE-5, pp. 128-138, March 1979.

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

[11] D. Roberts and R. Johnson, "Patterns for Evolving Frameworks," In R. Martin, D. Riehle, F. Buschmann (Eds.), *Pattern Languages of Program Design 3*, pp. 471-486, Addison-Wesley, 1998.

[12] 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.

[13] J. Visser. "Visitor Combination and Traversal Control," In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '01), pp. 270-282, October 2001.