On Feature Orientation and Functional Programming

Sven Apel[†], Christian Kästner[‡], Armin Größlinger[†], and Christian Lengauer[†]

[†] Department of Informatics and Mathematics, University of Passau
{apel,groesslinger,lengauer}@uni-passau.de

[‡] School of Computer Science, University of Magdeburg
kaestner@iti.cs.uni-magdeburg.de

UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Technical Report, Number MIP-0806
Department of Informatics and Mathematics
University of Passau, Germany
November 2008

On Feature Orientation and Functional Programming

Sven Apel[†], Christian Kästner[‡], Armin Größlinger[†], and Christian Lengauer[†]

[†] Department of Informatics and Mathematics, University of Passau
{apel,groesslinger,lengauer}@uni-passau.de
[‡] School of Computer Science, University of Magdeburg
kaestner@iti.cs.uni-magdeburg.de

Abstract. The *separation of concerns* is a fundamental principle in software engineering. *Crosscutting concerns* are concerns that do not align with hierarchical and block decomposition supported by mainstream programming languages. In the past, crosscutting concerns have been studied mainly in the context of object orientation. *Feature orientation* is a novel programming paradigm that supports the implementation of crosscutting concerns in a system with a hierarchical block structure. We explore the problem of crosscutting concerns in functional programming and propose two solutions based on feature orientation. Two case studies support our claims.

1 Introduction

The principle of *separation of concerns* is fundamental in software engineering [1]. The idea is to break down software into manageable pieces in order to allow a programmer to concentrate on individual concerns in isolation. A *concern* is a semantically coherent issue of a problem domain that is of interest to a stakeholder, e.g., transaction management in a database system or multi-user support in an operating system. Concerns are the primary criteria for decomposing a software system into code units [2].

In programming languages, many abstraction and modularization mechanisms have been invented to achieve a high degree of separation of concerns, e.g., functions, classes, and packages. However, in the late 1990s, the point was made that traditional abstraction and modularization mechanisms are not sufficient for the implementation of a particular class of concerns, called *crosscutting concerns* [3].

Crosscutting is defined as a structural relationship between concern implementations that is alternative to hierarchical and block structure. A key observation is that programming languages that support only hierarchical and block structure, i.e., that offer only mechanisms like functions, classes, and packages, are not sufficient for the implementation of crosscutting concerns [3, 4, 5]. This limitation is called the *tyranny of the dominant decomposition* [4]: a program can be modularized in only one way at a time, and the concerns that do not align with this modularization end up scattered across many modules and tangled with one another. The reason is that typically only abstraction and modularization mechanisms are provided that support the decomposition of orthogonal concerns, e.g, classes or functions. Overlapping or crosscutting concerns like synchronization and logging are not supported. Zhang and Jacobson have analyzed several large software projects and found many crosscutting concerns that affect wide parts of the code basis [6].

Since the 1990s, the problem of crosscutting concerns has been studied in depth and many approaches have been proposed [3,7,4,8]. Many researchers have focused on enhancing object-oriented programming to support the separation and modularization of crosscutting concerns. Interestingly, in an early publication on this topic, it was conjectured that the problem of crosscutting concerns occurs also in functional programming [3]. Still, only few researchers explored the problem of crosscutting in functional programming – mainly with a focus on language theory. So, it is not known what the shape and impact of crosscutting concerns in functional programs are, and there are only few practical tools and languages that support crosscutting concerns (see Sec. 6).

In our work on software product lines [9, 10], we noted the existence of crosscutting concerns in functional programs when we wanted to decompose software systems into reusable code units that can be combined flexibly to produce different variants of a program tailored to specific scenarios/needs. As with other software artifacts, e.g, written in Java, we found that crosscutting concerns in functional programs lead to code scatting and tangling. This motivated us to explore the problem of crosscutting in functional programming and to provide proper development support. Our solution is based on feature orientation, a programming paradigm for large-scale software synthesis and software product lines [8, 11, 12].

We contribute an analysis and discussion of the problem of crosscutting concerns in functional programs and explain why traditional abstraction and modularization mechanisms of functional languages, such as functions, algebraic data types, and monads, are not sufficient to implement them. Based on this discussion, we propose two solutions that rely on feature orientation. Then, we discuss our experience with two case studies on crosscutting concerns in Haskell programs and how to deal with them using the two feature-oriented tools that we have built/extended for this purpose.

2 Abstraction and Modularization in Functional Programming

Since they are supported by most functional languages, we concentrate on the following mechanisms: modules, algebraic data types, functions, and monads. We discuss each mechanism with respect to its capability to separate and modularize concerns, especially crosscutting concerns.

2.1 Abstraction and Modularization Mechanisms

Modules. A module encapsulates a set of definitions, in particular, data type and function definitions. It exports an interface for communication with other modules and hides the details of the implementation of inner data type and function definitions. Modules are used to decompose a system into a hierarchy that is formed by "use" relationships. In this sense, modules are similar to packages and classes in object-oriented languages.

It has been observed that, in object-oriented languages, crosscutting concerns typically cut across the boundaries of packages and classes, e.g., concerns like synchronization, persistence, and authorization. For example, the support of Enterprise Java

Beans in an IBM application server cuts across 35 top-level components [13]. In our case studies, we observed that this phenomenon occurs in functional programs as well. Specifically, we found module-level crosscutting to be the most frequent form of crosscutting. The reason is that modules are coarse-grained building blocks and that they impose an hierarchical block structure on the program that does not align with crosscutting concerns.

Algebraic Data Types. A programmer may define her/his own complex data type on the basis of more basic data types. An algebraic data type definition provides a list of alternative constructors that are used by the programmer to construct different variants of the data type. For example, a list data type usually has a constructor for an empty list and a constructor for adding an element to an existing list.

An algebraic data type encapsulates the fraction of a concern that is related to its data representation. In this context, crosscutting means that a single data type definition contains constructors that belong to multiple concerns and the implementation of a single concern affects multiple data type definitions. For example, in a data type definition of a list, there may be constructors for transient and persistent lists. Of course, one could implement two data types, one for transient lists and one for persistent lists, but, in this case, the list concern would be scattered across two data type definitions.

Type-level crosscutting has been observed in object-oriented programming, e.g. the thread synchronization in Berkeley DB is scattered across 28 classes [14]. Our case studies revealed that this problem occurs in functional programming as well.

Functions. Functions are the primary means to decompose and structure the computation of a program. A function cooperates with other functions via its interface (its signature). The internal implementation of a function is not accessible from outside, instead parameters and a return value are used for passing data.

A function encapsulates a concern regarding the computation of a program. Crosscutting means here that the evaluation of a function involves the evaluation of terms that belong to multiple concerns and that a concern is implemented by the evaluation of terms in different functions. This also implies that the number and structure of the parameters and the return value of a function may be influenced by multiple concerns.

For example, in an expression evaluator, an evaluation function processes different shapes of terms. Depending on the presence of other concerns, e.g., support for if-then-else constructs or variables, the implementation of the evaluation function changes. That is, there must be equations, patterns, branches, and parameters for different kinds of terms. The implementation of the three concerns (basic evaluation, variables, if-then-else constructs) is tangled in the implementation of a single function, even in individual equations. This situation occurs because the concerns of evaluation and variables and if-then-else constructs overlap, i.e., crosscut, at the function level. Function-level crosscutting has been observed in object-oriented programs. For example, the disc quota concern in the FreeBSD operating system is scattered across 22 functions located in 11 files [15]. In our case studies, we have observed some function-level crosscutting in functional programs, although it does not occur as frequently as module-level crosscutting.

Monads. A monad is a kind of abstract data type used to represent computations (instead of data). With a monad, one can program with state while preserving referential transparency. Functional programs can make use of monads for structuring procedures that include sequenced operations (imperative programming style), or for defining arbitrary deterministic control flows (like handling concurrency, continuations, or exceptions).

Since monads can be used to emulate an imperative programming style, they may be subject to monad-level crosscutting. This is similar to method- or procedure-level crosscutting in object-oriented and imperative languages, where, inside the implementation of a method or procedure, implementations of multiple concerns are tangled, and a single concern may affect implementations of multiple methods or procedures, e.g, as in the case of the disc quota concern in FreeBSD [15]. Like methods and procedures, monads are used to decompose a program into blocks of stateful computations. For example, a monad can be used to untangle the different phases of a compilation process. In this case, the data exchanged between the phases are passed implicitly but, inside the monad, the different concerns (compilation phases) are still tangled. In our case studies, we found only few cases of monad-level crosscutting.

2.2 Discussion

Previous experience with object-oriented and imperative languages indicates that crosscutting is a real problem that is likely to occur also in functional programming. The reason for the latter is that, like object-oriented and imperative languages, functional languages provide mechanisms for hierarchical and block decomposition, but not for crosscutting decomposition. A conclusion from previous work is that a primary (dominant) decomposition is not able to separate and modularize all kinds of concerns [4]. Crosscutting is the relationship between different overlapping decompositions. Mezini and Ostermann even argue that crosscutting is a problem of knowledge representation in that it is caused by the fact that hierarchical decomposition requires a primary model of the data to be represented that subordinates other models [16]. According to this view, there is a general problem with hierarchical and block decomposition such as provided by contemporary functional languages and their mechanisms like modules, function and data type definitions, and monads. That is, in the presence of crosscutting concerns, (1) the implementation of multiple concerns is entangled inside a module / algebraic data type / function / monad and (2) the implementation of a crosscutting concern is scattered across multiple modules / algebraic data type definitions / function definitions / monads.

We assume the reader's agreement that a separation and modularization of concerns (incl. crosscutting concerns) is desirable. We do not repeat the previously postulated and observed positive effects a proper separation and modularization of concerns can incur [2, 17, 1, 3, 4], but concentrate on possible solutions for functional programming.

3 An Analysis of Crosscutting in Two Haskell Programs

We have analyzed the occurrences and properties of crosscutting concerns in two Haskell programs. We chose Haskell since it is a widely used functional language and offers all of the mechanisms that we have discussed. The first program, called Arith, is an arithmetic expression evaluator that has been written by the third author. The evaluator supports variables, if-then-else constructs, lambda expressions, static and dynamic scoping, and different kinds of evaluation strategies. The second program, called Functional Graph Library (FGL), is a library for graph data structures and algorithms developed by M. Erwig.[1] The library contains implementations for directed and undirected graphs, with labeled and unlabeled edges and nodes, dynamic and static allocation, and all kinds of algorithms.

First, we selected, for each of the two programs, a set of concerns that are common in the particular domain. For example, for Arith we selected different evaluation strategies and arithmetic operations, and for FGL different properties of graphs and graph algorithms. Overall, we selected 13 concerns of Arith and 18 concerns of FGL. For Arith, selecting concerns was easy since the third author wrote the program; for FGL, we studied the documentation and example use cases for identifying concerns of interest.

Then, we browsed the source code of both programs, looking for the code that implements the selected concerns, and judged their locality and separation from other concerns in the code basis. When we found indications of crosscutting concerns, i.e., code scattering and tangling, we classified it into module-level, type-level, function-level, monad-level, and equation-level crosscutting. We have added equation-level crosscutting in order to distinguish between the case that a function contains two or more equations that belong to different concerns (function level) and the case that an equation itself contains code belonging to different concerns (equation level).

For example, the support for evaluating expressions containing variables in Arith is scattered across the entire program at all levels. Specifically, it affects the parser and main modules, the expression, type, and error data types, the evaluation and lookup functions, as well as internals of several equations of the evaluation function.

In Table 1, we list the numbers of occurrences of crosscutting in Arith and FGL. In particular, we show the number of elements (modules, functions, etc.) affected by multiple concerns and the number of concerns that are tangled inside a type of element. (More details about the concerns are provided in the Sections A and B.) We noted that there is a difference between Arith and FGL. In Arith, all kinds of crosscutting concerns occur. In FGL, we found only module-level crosscutting. A reason may be that FGL is a library and many functions and data types are largely independent, e.g., individual graph algorithms do not interfere at all. In Arith, all kinds of crosscutting concerns occur, but mainly at a coarse grain, i.e., at module, type, and function level; we found only few instances of crosscutting at the monad or equation level. This observation will be relevant for the comparison of our two solutions (see Sec. 4.2).

4 Feature-Oriented Decomposition of Functional Programs

In order to achieve a proper separation of concerns in functional programs, we propose to use the paradigm of feature orientation.

[1] `http://web.engr.oregonstate.edu/~erwig/fgl/haskell`

	Arith (425 LOC, 13 concerns)			**FGL** (2573 LOC, 18 concerns)		
	overall elements	elements affected	concerns crosscut	overall elements	elements affected	concerns crosscut
module level	2	2	24	43	8	21
type level	7	5	21	11	0	0
function level	25	3	21	289	0	0
monad level	11	2	6	52	0	0
equation level	69	1	5	582	0	0

overall elements: overall number of elements; **elements affected**: number of elements affected by multiple concerns ; **concerns crosscut**: number of concerns tangled inside a type of element

Table 1. An overview of crosscutting concerns in Arith and FGL.

4.1 Feature Orientation

The basic idea of feature orientation is to decompose software systems in terms of features. A *feature* is the realization of a requirement on a software system relevant to some stakeholder [8, 11]; features are used in software product lines to represent commonalities and variabilities of software products [12]. *Decomposition* means both the mental process of structuring a complex problem into manageable pieces and the structuring that results from this process. Technically, the implementation of a feature is an increment in program functionality and involves the addition of new program structures and the extension of existing program structures [11, 18]. Feature orientation has been used to structure and synthesize software systems of different sizes (up to 300 KLOC) written / represented in different languages, e.g., Java, C, C#, C++, XML, JavaCC, UML [12, 11, 9, 10, 18].

Here, we are interested mainly in the mechanisms that feature-oriented programming languages and tools offer in order to express, separate, and modularize crosscutting concerns. The subtle difference between the concept of a feature and of a concern is not relevant for our discussion (see Apel et al. [18]) – in the remainder of the paper, we use both terms synonymously. Likewise, the questions of what a feature really is or what the relationship to software product lines is are out of scope. There are two principal approaches for the decomposition of a software system into features, namely the approaches of physical decomposition and virtual decomposition.

In a *physical decomposition*, code belonging to a feature is encapsulated in a designated *feature module*. In order to generate a final program, the desired feature modules are composed. The mechanisms for expressing and composing feature modules must cope with crosscutting concerns because, usually, features cannot be implemented by single components, classes, or functions [18]. There are numerous approaches to implement feature modules, e.g., mixin layers [11], aspects [14], hyperslices [4], and aspectual feature modules [18].

In a *virtual decomposition*, code belonging to a feature is not isolated in the form of a physical code unit, but only annotated. That is, the code belonging to different concerns is left intermixed inside a module implementation, and it is specified which code belongs to which feature. A standard approach is to use #ifdef statements of the C preprocessor. In a more recent approach, colors in the presentation layer of an editor are used to annotate code, instead of adding textual annotations to the program text [9]. The advantage is that the programmer can easily distinguish the program text and information regarding features. A virtual separation of concerns is achieved with views on the source code to show only the code of a certain feature or feature combination. Furthermore, in the color approach, it is checked that only meaningful fragments of a program are assigned to features in order to avoid errors during and after composition [9]. For example, it is not possible to color only the name of a class or an opening bracket.

4.2 Our Approach

Presently, it is not clear whether a physical or virtual decomposition is superior. The advantage of the virtual approach is that every optional syntax element of a program can be annotated with a feature, including parameters and (sub)expressions. In a physical separation, mainly large structures such as packages, classes, and methods can be extended; extensions at the method level are difficult [9]. In turn, the advantage of the physical approach is that a programmer can achieve real information hiding by defining interfaces [16]. This is not possible in the virtual approach, which intermixes (colored) code belonging to different features.

We do not intend here to judge the advantages and disadvantages of the two approaches, but we want to stress that the two approaches are different. This motivates us to explore their capabilities in the decomposition of functional programs, in particular, of Haskell programs. In order to support the physical decomposition of Haskell programs, we have extended an existing tool for feature composition, called FEATUREHOUSE,[2] and in order to support the virtual decomposition of Haskell programs, we have extended an existing tool for virtual feature decomposition, called CIDE.[3]

Composing Haskell Files with FEATUREHOUSE. FEATUREHOUSE is a tool for feature composition. A feature is represented by a containment hierarchy. A containment hierarchy is a directory that contains possibly further subdirectories and a set of software artifacts such as Java, Haskell, and HTML files. In Figure 1, we show the containment hierarchies of a simple expression evaluator consisting of a feature EXPR for the representation of simple expressions including operations for addition and subtraction, a feature EVAL for the evaluation of expressions, and a feature MULT for multiplication. All features contain source and non-source code artifacts. Feature composition, denoted by '•', basically merges all involved files. It is implemented by the recursive superimposition of the directory and file structure [10]. For example, the resulting artifact Expr.hs is composed of its counterparts in the features MULT, EVAL, and EXPR.

The process of superimposition does not terminate at the level of files [10]. It proceeds with the internal structure of the artifacts contained in a feature's containment hierarchy – in our case, with the internal structure of Haskell files (modules, data types, functions, etc.). In Figure 2, we depict an excerpt of the file Expr.hs of the feature EXPR,

[2] www.fosd.de/fh
[3] http://www.fosd.de/cide

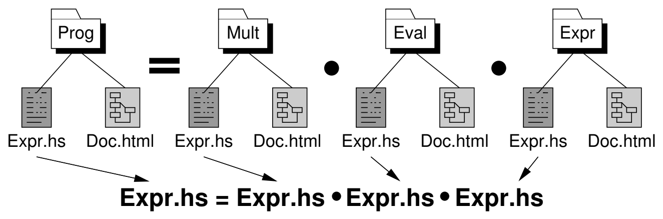

Fig. 1. Composing the containment hierarchies of MULT, EVAL, and EXPR.

of its counterparts in EVAL and MULT, and of their superimposition. The feature EVAL adds a new function (incl. three equations) to the module introduced by EXPR. The feature MULT adds a new constructor to the data type for expressions and adds a new equation to the evaluation function.

```
module Expr where {
  data Expr = Num Int | Add Expr Expr | Sub Expr Expr deriving Show;
}
```

●

```
module Expr where {
  eval :: Expr −> Int;
  eval (Num x) = x;
  eval (Add x y) = (eval x) + (eval y);
  eval (Sub x y) = (eval x) − (eval y);
}
```

●

```
module Expr where {
  data Expr = Mul Expr Expr deriving Show;
  eval (Mul x y) = (eval x) ∗ (eval y);
}
```

=

```
module Expr where {
  data Expr = Num Int | Add Expr Expr | Sub Expr Expr | Mul Expr Expr deriving Show;
  eval :: Expr −> Int;
  eval (Num x) = x;
  eval (Add x y) = (eval x) + (eval y);
  eval (Sub x y) = (eval x) − (eval y);
  eval (Mul x y) = (eval x) ∗ (eval y);
}
```

Fig. 2. Composing an expression evaluator from the features EXPR, EVAL, and MULT.

With FEATUREHOUSE, composing a feature with a Haskell program can result in the following additions to the program:

– definitions to a module (e.g., functions, data types, type classes, instances)
– imports and exports to a module
– type constructors and derived instances to a data type definition

– equations to a function
– signatures to a type class

The FEATUREHOUSE tool along with the case studies presented in Section 5 can be downloaded from FEATUREHOUSE's website.

Coloring Haskell Files with CIDE. CIDE is a tool for virtual feature decomposition. As explained in Section 4.1, a programmer assigns colors to code fragments. Each color stands for a separate feature. We have extended the CIDE tool in order to be able to color Haskell programs, beside Java, C#, C, XML, HTML, JavaCC, and ANTLR documents.

In Figure 3, we depict an excerpt of the expression evaluator in which code belonging to the features EVAL and MULT has been colored.[4] Using views on the source code, code belonging to individual features can be selected and hidden in the editor or even removed in a generation step. For example, one could hide all code that belongs to other features than EVAL or set the focus on code belonging to MULT.

```
module Expr where {
  data Expr = Num Int | Add Expr Expr
                 | Sub Expr Expr | Mul Expr Expr          // Feature MULT
                 deriving Show;
  eval :: Expr -> Int;                                    // Feature EVAL
  eval (Num x) = x;                                       // Feature EVAL
  eval (Add x y) = (eval x) + (eval y);                   // Feature EVAL
  eval (Sub x y) = (eval x) - (eval y);                   // Feature EVAL
  eval (Mul x y) = (eval x) * (eval y);                   // Feature MULT
}
```

Fig. 3. Colored version of the expression evaluator (EXPR, EVAL, MULT).

As mentioned, CIDE enforces a principle of safe coloring. Typically, it is not meaningful to allow a programmer to color code fragments arbitrarily. The reason is that colored code fragments can be hidden or removed in CIDE, and the remaining code (without the code of the removed feature) should be still syntactically correct Haskell code. To this end, CIDE uses information of the language's syntax to ensure syntactical correctness [9,19]. For example, entire modules, function and data type definitions, as well as individual type constructors, function equations, module imports and exports, and even single parameters or (sub)expressions can be colored. Examples of non-optional elements that must not be colored individually are a module's, function's, or data type's name, opening or closing brackets, or single keywords like where or case.

CIDE including support for Haskell and the case studies presented in Section 5 can be downloaded from CIDE's website.

[4] For grayscale versions of the paper, we have added comments that indicate which lines belong to which features.

5 Case Studies

In order to separate the crosscutting concerns identified in our analysis, we have decomposed Arith and FGL with our tools FEATUREHOUSE and CIDE. Our goal is to explore the capabilities of feature decomposition for separating crosscutting concerns in functional programs.

5.1 Physical Decomposition with FEATUREHOUSE

We have decomposed Arith into the 13 concerns/features described above. For reasons we explain shortly, we required multiple feature modules (containment hierarchies) for some concerns so that we implemented overall 13 concerns with 27 feature modules.

The main task of the decomposition was to divide the two Haskell modules of Arith into multiple fragments that contain the definitions that belong to the different features. Typically, a feature adds new function definitions and data type definitions to the base Arith program and extends existing functions by new equations and existing data types by new constructors. For example, the feature UNOP adds a new data type UnOp to Arith and extends the existing data type Exp by a constructor for unary operations.

When adding new equations to a function, we stumbled over a problem, e.g., when adding the equation 'eval env (Bin op exp1 exp2)...' to the function eval in order to support the evaluation of binary operations. The problem is that the order in which the equations of a function appear in a module may matter. That is, when swapping two equations of a function, the program behavior may change, e.g, the program fragments below on the left and right side are not equivalent because their patterns overlap:

```
eval env (Bin op exp1 exp2) = ...          eval _ _ = ...
eval _ _ = ...                             eval env (Bin op exp1 exp2) = ...
```

The problem of refining modules via superimposition is that we can always add something at the end or in front of an existing element. This is no problem when adding new functions, type classes, and data types, since their lexical order within the enclosing module does not matter. But, adding a new equation right before another equation or in the middle of two other equations is problematic. With superimposition and its implementation in FEATUREHOUSE there are no linguistic means to express this kind of extensions properly. This also implies that implementing the different equations of eval using a case expression would not solve the problem, either. However, this problem is not specific to Haskell but occurs also in Java and other languages [9]. A workaround is to split the target module exactly at the position at which we want to refine it, called *sandwiching* [17]. We had to use this workaround twice in the decomposition of Arith.

A further problem was to separate code of crosscutting concerns at the monad and function level. Let us illustrate this by an example. In Arith, the function eval plays a central role for expression evaluation. Depending on the features selected, the definition of the function must change. For example, if the feature BINOP is selected, the function eval must contain an equation that processes binary operations:

```
eval (Bin op exp1 exp2) = zRes (tvBinOp op) (eval exp1) (eval exp2);
```

Likewise, if the feature UNOP is selected, the function eval must contain an equation that processes unary operations:

```
eval (Un op exp) = mRes (tvUnOp op) (eval exp);
```

But, if we select the feature VAR for processing expressions containing variables, we cannot simply add a further equation. We have to change *every* equation of eval in order to pass an environment parameter through the expression evaluation. That is, we have to extend the signature of the function eval by a new parameter that represents the environment that maps variable names to values. Accordingly, the definition of the function has to be changed from

```
eval :: Exp TVal −> Res TVal;    to    eval :: Env TVal −> Exp TVal −> Res TVal;.
```

But extending a given function with a new parameter and changing the function's equations is not possible in FEATUREHOUSE. This problem is also known in object-oriented and feature-oriented languages [9]. Hence, we had to copy the existing eval functions, add a new parameter, and pass it to the recursive invocations of eval:

```
eval env (Bin op exp1 exp2) = zRes (tvBinOp op) (eval env exp1) (eval env exp2);
eval env (Un op exp) = mRes (tvUnOp op) (eval env exp);
```

A different solution would be to write eval as monadic function whose type is parameterized with the monad in which the evaluation takes place. Different evaluation strategies would be obtained by running eval in different monads.In this case, the code for evaluation would be still scattered across multiple monads.

Finally, we found that the number of implemented features (containment hierarchies) is higher than the number of concerns that we identified upfront. The reason is that for some concerns we had to implement more than one feature module. For example, the evaluating lambda expressions is very different for a lazy and a strict evaluation order. So we had to implement a feature for lambda expressions with lazy evaluation order and for lambda expressions with strict evaluation order. This problem is also known in object-oriented programming as feature optionality problem [8], and our additional feature implementations are called lifters in this context.

Furthermore, we have decomposed FGL into the 18 concerns/features based on the analysis of Section 3 using 20 feature modules. Most features separate code concerning different kinds of graphs or different graph algorithms. In contrast to Arith, the spectrum of extensions features make to the base program is broader. Beside adding new function and data type definitions, some features add new type classes and instance declarations; and beside extending existing functions with new equations and existing data type definitions with new constructors, some features extend modules with new import and export declarations. Like in Arith, we had to use the sandwiching workaround twice in order to extend a function by new equations; unlike in Arith, we did not experience the feature optionality problem.

In Table 2, we provide, beside the numbers of lines of code (LOC) and features, the numbers on how many modules, data types, functions, and monads exist, how many of them have been extended, and how frequently they have been extended.

5.2 Virtual Decomposition with CIDE

For the virtual decomposition of Arith, we built first a full version containing all functionality and proceeded by coloring code step by step based on the analysis of Section 3.

	Arith (532 LOC, 27 feature modules)				**FGL** (2730 LOC, 20 feature modules)			
	modules	data types	functions	monads	modules	data types	functions	monads
overall	2	7	25	11	43	11	289	52
extended	2	5	4	2	8	0	0	0
extensions	31	18	32	4	28	0	0	0

overall: overall number of occurrences; **extended**: number of elements being extended; **extensions**: number of extensions applied to the type of element

Table 2. An overview of the extensions made by features in physical decomposition.

The coloring was straightforward and did not pose any challenges.[5] Compared to the physical decomposition, (1) we were faster, which is due to the simpler process and the knowledge we gained from the physical decomposition, (2) we did not have the problem of changing equation orders, since the order is already predefined in the colored code, (3) we could easily decompose crosscutting concerns at the monad and equation level, since CIDE is able to color individual parameters or subexpressions, and (4) we did not experience the feature optionality problem, since there is less concern overlap than in Arith.

Let us explain the virtual decomposition by an example. In the physical decomposition, we traded the possibility of separating the feature VAR from the other features for some overhead in code size caused by code replication. This was necessary because superimposition does not support adding new parameters to a function (see Sec. 5.1). Exactly this kind of situation can be solved elegantly with a virtual decomposition. In the colored variant of Arith, we have only one variant for each equation and the additional parameters and parameter passing are colored:[6]

```
eval env (Bin op exp1 exp2) = zRes (tvBinOp op) (eval env exp1) (eval env exp2);
eval env (Un op exp) = mRes (tvUnOp op) (eval env exp);
```

Similarly, we handled monad-level crosscutting; instead of replicating the monad implementation, we colored the parts that belong to different features. Nevertheless, we feel that when coloring definitions of functions and monads with too many colors, the code became difficult to understand.

Like in Arith, we have decomposed FGL using CIDE into a similar set of features as in the physical decomposition using FEATUREHOUSE. This process was very simple and straightforward forward since in FGL we found only module-level crosscutting. That is, in FGL, virtual decomposition did not outperform physical decomposition.

Overall, we were able to color exactly the concerns of Arith and FGL that we have identified in our analysis (Sec. 3). Consequently, the reader can infer information about the coloring of the code bases from the properties of crosscutting shown in Table 1.

[5] We thank Malte Rosenthal and Fabian Wielgorz, two students of us, for helping us coloring Arith and FGL.

[6] For grayscale versions of the paper, we have underlined the code that belongs to feature VAR.

5.3 Discussion

To summarize, we made the following observations in our case studies:

- There is indeed crosscutting at all levels (module, function, data type, monad).
- Both a physical and virtual decomposition of Haskell programs into features achieve a proper separation of concerns at different levels of granularity.
- There are concerns that cut across function signatures, equations, and expressions that require workarounds or a virtual decomposition à la CIDE. However, most crosscutting occurs at the level of modules and data type definitions, at which a physical decomposition is appropriate, too.
- Functional programming in Haskell aligns mostly well with feature decomposition. However, in the physical approach, the significance of the lexical order of function equations causes problems. A virtual decomposition or workarounds like sandwiching have to be used in these cases.
- Feature decomposition is largely orthogonal to data and function decomposition in functional programming. Only in rare cases a feature is implemented by exactly one function, data type, or module, e.g., in the case of graph algorithms.
- The feature optionality problem occurs also in functional programs and leads to an increased number of containment hierarchies in a physical decomposition.
- A too fine-grained virtual decomposition is counter-productive since the colored code is difficult to understand – even using views on the source code.

6 Related Work

Kiczales et al. were among the first to conjecture that crosscutting concerns occur in functional programs. They propose to use aspect-oriented programming to separate and modularize crosscutting concerns [3].

Aspect orientation is related to feature orientation – the two paradigms differ mainly in the language mechanisms that are commonly used [18]: typically, aspect-oriented languages use metaprogramming constructs like pointcuts to quantify over the events in the program's execution a crosscutting concern affects, and implicit invocation mechanisms like advice to execute code transparently. Feature-oriented tools and languages for a physical decomposition support mainly mixin and collaboration-based programming techniques. The techniques used are simpler and less expressive than aspect-oriented mechanisms [20]. Almost all related work focuses on aspect-oriented mechanisms in the context of functional language theory. Our study extends the state of the art with an analysis of crosscutting concerns in functional programs and the proposal of feature orientation as a possible solution.

AspectML and it predecessors [21] are functional programming languages with support for aspects. These languages are not intended for programming but for studying type systems. Consequently, there is no experience on whether crosscutting occurs in functional programs or what the properties of the crosscutting concerns are.

Tucker and Krishnamurthi have developed a variant of Scheme with aspect-oriented mechanisms [22]. They model pointcuts and advice as first-class objects. They do not aim at the analysis of crosscutting concerns in existing functional programs but at the relationship of aspect-oriented mechanisms and mechanisms like higher-order functions.

Masuhara et al. have developed an aspect-oriented version of Caml, called Aspectual Caml [23]. Like Tucker and Krishnamurthi, they focus on language theory and not on the properties and impact of crosscutting in functional programs.

Aldrich has used a simple functional language, called TinyAspect, to explore crosscutting at the module level [24]. This work concentrates mainly on issue on the principle of information hiding, not on the impact of crosscutting in functional programming.

Hofer and Ostermann offered a simple example of crosscutting in a Haskell program [25]. They noted that there is a relationship between aspects and monads. They argue that some significant properties of aspects, such as quantification, are not supported by monads and, consequently, monads are not capable of separating crosscutting concerns. We found that a virtual decomposition of monad-level crosscutting is feasible.

Findler and Flatt have formalized the concept of a mixin [26]. Although mixins have been proposed as an extension of classes in object-oriented languages, they draw a connection to functional programming: a mixin is a function that receives a class and that returns an extended class. The mixin mechanism is closely related to module refinement and superimposition. However, Findler and Flatt did not aim at crosscutting.

7 Conclusions

We have explored the problem of crosscutting concerns in functional programming. We have analyzed and discussed the capabilities of mechanisms of functional languages for separating and modularizing crosscutting concerns. Based on the limitations we identified – that mainly are rooted in the hierarchical and block decomposition scheme – we have proposed an approach based on physical and virtual feature decomposition and have extended two corresponding tools. In two case studies, we have explored the existence and characteristics of crosscutting as well as the performance of our tools in separating crosscutting concerns. We found that crosscutting occurs in functional programs and that physical and virtual decompositions are able to alleviate the problem of code scattering and tangling, however, with different mutual strengths and weaknesses.

References

1. Dijkstra, E.: On the Role of Scientific Thought. In: Selected Writings on Computing: A Personal Perspective. Springer-Verlag (1982) 60–66
2. Parnas, D.: On the Criteria to be Used in Decomposing Systems into Modules. Comm. ACM **15** (1972) 1053–1058
3. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proc. Europ. Conf. Object-Oriented Programming, Springer-Verlag (1997) 220–242
4. Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. Int. Conf. Software Engineering, IEEE CS (1999) 107–119
5. Kiczales, G., Mezini, M.: Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In: Proc. Europ. Conf. Object-Oriented Programming, Springer-Verlag (2005) 195–213

6. Zhang, C., Jacobsen, H.A.: Efficiently Mining Crosscutting Concerns Through Random Walks. In: Proc. Int. Conf. Aspect-Oriented Software Development, ACM Press (2007) 226–238
7. Skotiniotis, T., Palm, J., Lieberherr, K.: Demeter Interfaces: Adaptive Programming without Surprises. In: Proc. Europ. Conf. Object-Oriented Programming, Springer-Verlag (2006) 477–500
8. Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In: Proc. Europ. Conf. Object-Oriented Programming, Springer-Verlag (1997) 419–443
9. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. In: Proc. Int. Conf. Software Engineering, ACM Press (2008) 311–320
10. Apel, S., Lengauer, C.: Superimposition: A Language-Independent Approach to Software Composition. In: Proc. Int. Symp. Software Composition, Springer-Verlag (2008) 20–35
11. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Engineering **30** (2004) 355–371
12. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
13. Colyer, A., Clement, A.: Large-Scale AOSD for Middleware. In: Proc. Int. Conf. Aspect-Oriented Software Development, ACM Press (2004) 56–65
14. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features using AspectJ. In: Proc. Int. Software Product Line Conf., IEEE CS (2007) 222–232
15. Coady, Y., Kiczales, G.: Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In: Proc. Int. Conf. Aspect-Oriented Software Development, ACM Press (2003) 50–59
16. Mezini, M., Ostermann, K.: Untangling Crosscutting Models with CAESAR. In: Aspect-Oriented Software Development. Addison-Wesley (2005) 165–199
17. Parnas, D.: Designing Software for Ease of Extension and Contraction. IEEE Trans. Software Engineering **SE-5** (1979) 264–277
18. Apel, S., Leich, T., Saake, G.: Aspectual Feature Modules. IEEE Trans. Software Engineering **34** (2008) 162–180
19. Kästner, C., Apel, S.: Type-checking Software Product Lines – A Formal Approach. In: Proc. Int. Conf. Automated Software Engineering, IEEE CS (2008)
20. Apel, S., Batory, D.: How AspectJ is Used: An Analysis of Eleven AspectJ Programs. Technical Report MIP-0801, Dept. of Informatics and Mathematics, University of Passau (2008)
21. Dantas, D., Walker, D., Washburn, G., Weirich, S.: AspectML: A Polymorphic Aspect-Oriented Functional Programming Language. ACM Trans. Programming Languages and Systems **30** (2008) 1–60
22. Tucker, D., Krishnamurthi, S.: Pointcuts and Advice in Higher-Order Languages. In: Proc. Int. Conf. Aspect-Oriented Software Development, ACM Press (2003) 158–167
23. Masuhara, H., Tatsuzawa, H., Yonezawa, A.: Aspectual Caml: An Aspect-Oriented Functional Language. In: Proc. Int. Conf. Functional Programming, ACM Press (2005) 320–330
24. Aldrich, J.: Open Modules: Modular Reasoning about Advice. In: Proc. Europ. Conf. Object-Oriented Programming, Springer-Verlag (2005) 144–168
25. Hofer, C., Ostermann, K.: On the Relation of Aspects and Monads. In: Proc. Workshop Foundations of Aspect-Oriented Languages, ACM Press (2007) 27–33
26. Findler, R., Flatt, M.: Modular Object-Oriented Programming with Units and Mixins. In: Proc. Int. Conf. Functional Programming, ACM Press (1998) 94–104

A Overview of the Concerns of Arith

The below table lists for each concern of Arith the number of elements being involved in the concern's implementation:

concern	module	data type	function	monad	equation
binary operations	2	2	3	0	12
unary operations	1	2	2	0	6
boolean	2	3	3	0	4
variables	2	4	8	0	17
if-then-else	2	1	2	0	2
lambdas	2	3	4	0	10
lazy evaluation	1	1	1	0	8
strict evaluation	1	1	1	0	7
dynamic scoping	1	1	1	0	4
static scoping	1	1	1	0	4
no variables	1	1	1	0	4
windows console	1	0	0	1	0
linux console	1	0	0	1	0

B Overview of the Concerns of FGL

The below table lists for each concern of FGL the number of elements being involved in the concern's implementation:

concern	module	data type	function	monad	equation
static graph	2	0	21	0	21
dynamic graph	3	0	20	0	20
graphviz interface	2	1	9	0	6
monadic graph	3	2	51	1	52
unlabeled nodes	2	0	2	0	2
unlabeled edges	2	0	19	0	19
articulation points	2	1	9	0	15
breadth first search	1	0	18	0	19
depth first search	2	1	31	0	38
connected components	1	0	5	0	8
independent components	1	0	2	0	3
shortest path	2	0	11	0	12
dominators	1	0	10	0	11
voronoi diagrams	1	0	7	0	7
max flow 1	1	0	8	0	11
max flow 2	1	1	21	0	32
minimum span tree	1	0	7	0	9
transitive closure	1	0	2	0	2