Feature (De)composition in 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 (de)composition of crosscutting concerns in a system with a hierarchical block structure. In two case studies we explore the problem of crosscutting concerns in functional programming and propose two solutions based on feature orientation.

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 the research area of programming languages, a wide variety of abstraction, modularization, and composition 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 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 (de)composition 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 for scattering and tangling is that typically only mechanisms are provided that support the (de)composition of concerns that align well with the given module structure, e.g., established by classes or functions. Overlapping or crosscutting concerns like transaction management are not supported. For example, Zhang and Jacobson have analyzed several large software

projects and found many crosscutting concerns that affect large parts of the code basis [6].

Since the 1990s, the problem of crosscutting concerns has been studied in depth and many approaches, most notably aspect-oriented programming, have been proposed [3, 7, 4, 8]. Many researchers have focused on enhancing object-oriented programming to support the separation, modularization, and composition of crosscutting concerns. Interestingly, in an early publication, 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 and not on code structure and modularity. So, it is not known what the shape and impact of crosscutting concerns in functional programs are. Also there are only few practical tools and languages that support crosscutting concerns (see Sec. 6).

In our work on software product lines [9,10,11], 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 or needs. As with other software artifacts, e.g., written in Java, we found that crosscutting concerns in functional programs lead to code scattering and tangling. This motivated us to explore the problem of crosscutting in functional programming and to develop proper development support. Our solution is based on feature orientation, a programming paradigm for large-scale software composition and software product lines [8, 12, 13].

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

2 Abstraction and Modularization in Functional Programming

For simplicity, we concentrate on a representative collection of mechanisms of functional programming languages: modules, algebraic data types, functions, and monads. We discuss each mechanism with respect to its capability to separate and modularize concerns, especially crosscutting concerns.

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 and persistence. For example, the support of Enterprise Java Beans in an IBM application server cuts across 35 top-level components [14]. In our case studies, we observed

this phenomenon 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 a hierarchical block structure on the program that does not align with many crosscutting concerns.

Algebraic Data Types. A programmer may define her / his own data type on the basis of previously defined 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 a list.

An algebraic data type encapsulates the data related to a concern. 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., synchronization in Oracle's Berkeley DB is scattered across 28 of 300 classes [15]. Our case studies revealed that scattering occurs in functional programming too, especially, since data types cannot be extended like classes in object-oriented languages.

Functions. The function is the primary means for decomposing and structuring the computation of a functional program. A function cooperates with other functions via its interface (its signature). The internal implementation of a function is not accessible from the outside. Instead, parameters and a return value are used for exchanging 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 implies that the number and types of the parameters and the return value of a function may be influenced by multiple concerns. Note that higher-order functions and lazy evaluation do not solve the problem of crosscutting. Both are means to modularize programs [16] but, since they take effect at function application and composition, the module structure they impose is hierarchical.

For example, in an expression evaluator, an evaluation function processes different shapes of terms. Depending on the presence of other concerns, e.g., support of variables or choice constructs, the implementation of the evaluation function varies. The implementations of the concerns 'basic evaluation', 'variables', 'choice' are tangled in the definition of a single function, even in individual equations. This situation occurs because the concerns overlap (i.e., crosscut) at the function level. Function-level crosscutting has been observed in object-oriented and imperative programs. For example, the disc quota concern in FreeBSD is scattered across 22 functions located in 11 files [17]. In our Haskell case studies, we have observed some function-level crosscutting, 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. 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 flow (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. Inside the implementation of a method or procedure, implementations of multiple concerns may be tangled, and a single concern may affect implementations of multiple methods or procedures as, e.g, in the case of the disc quota concern in FreeBSD [17]. Like methods or 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.

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 sufficient for separating and modularizing 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 dominates other models [18]. According to this view, there is a general problem with hierarchical and block decomposition such as provided by contemporary functional languages and their mechanisms: 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 types / functions / 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 that a proper separation and modularization of concerns can incur [2, 19, 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 we were interested in studying. The first program, called Arith, is an arithmetic expression evaluator written by the third author. The evaluator supports variables, choices, lambda abstraction, static and dynamic scoping, and different 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.

In a first step, we selected, for each of the two programs, a set of concerns. The selection was driven by two criteria: concerns that are (1) common and well-known in the target domain or (2) good candidates for crosscutting. For example, for Arith, we selected different evaluation strategies and arithmetic operations (13 concerns) and, for FGL, different flavors of graphs and graph algorithms (18 concerns). For Arith, selecting concerns was easy since the third author wrote the program; for FGL, we studied the documentation and examples.

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

For example, the code for evaluating expressions that contain variables is scattered in Arith 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 column 'overall elements', we show the number of elements (modules, functions, etc.) contained in Arith and FGL. For example, the upper left 2 means that, in Arith, there are two modules. In column 'elements affected', we show the number of elements influenced by multiple concerns. For example, the upper 2 means that, in Arith, both modules contain code of more than one concern. In column 'concerns crosscut', we show the number of concerns that are tangled inside an element. For example, the upper right 24 means that the two modules of Arith contain code of, in all, 24 concerns, which is, on average, 12 concerns per module. Numbers on all concerns are provided in Appendices A and B.

During our analysis we noted a difference between Arith and FGL. Arith contains all kinds of crosscutting concerns. 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. Furthermore, the crosscutting concerns of Arith are mainly at a coarse grain, i.e., at the module, type, and function level; we found only few instances of crosscutting at the monad or equation level. This observation will be relevant in the comparison of our two solutions (see Sec. 4.2). We did not find crosscutting for which higher-order functions or lazy evaluation are relevant.

[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.

Finally, an interesting question was whether the identified crosscutting concerns could be implemented more modularly using native Haskell mechanisms (not using feature orientation, as we propose in the next section). For Arith, we can answer this question definitely: most crosscutting concerns cannot be untangled without adverse effects on other parts of the program structure. The reason is that, in Arith, all kinds of crosscutting occur and that, especially at a fine grain, mechanisms like Haskell modules and functions are not capable of separating code belonging to different overlapping concerns. For example, a function's formal parameters, that belong to a different concern than the function itself, are difficult to separate from the function's definition.

For FGL, the answer is more difficult. On the one hand, we found only module-level crosscutting and, using a sophisticated module structure with proper imports, we could have untangled the different concern implementations. The difficulty with this approach is that we would hardwire the composition of modules, i.e., one could not easily remove or add new modules implementing additional concerns such as new algorithms – this is possible with feature orientation. On the other hand, the developers have accepted a structure with scattered and tangled code. The reason could be that they were not aware of the problem of crosscutting concerns (this unawareness is still wide-spread in real world programming) or that a different structure would not have match their intuition or would have lead to other structural problems, which we are not aware of.

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.

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, 12]; features are used to represent commonalities and variabilities of software systems [13]. *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 [12, 20]. Feature orientation has been used to structure and synthesize software systems of different sizes (up to 300 KLOC) written or represented in different languages, e.g., Java, C, C#, C++, XML, JavaCC, UML [13, 12, 9, 10, 20, 11, 21].

Here, we are interested mainly in the mechanisms that feature-oriented programming languages and tools offer in order to express, modularize, and compose crosscutting concerns. The subtle difference between the concept of a feature and of a concern is not relevant to our discussion (see Apel et al. [20]) – in the remainder of the paper, we use both terms synonymously. There are two principal approaches to the decomposition of a software system into features: physical decomposition and of virtual decomposition [22]. Common to both approaches is the support of variant generation of software systems. For example, most database systems have a transaction management but some do not need this feature, e.g., those for mobile and embedded systems. A physical or virtual decomposition allows programmers to configure a software system on the basis of a feature selection by removing or including feature-related code (crosscutting or not crosscutting).

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. There are numerous approaches for implementing feature modules, e.g., mixin layers [12], aspects [15], and hyperslices [4].

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 the #ifdef statement of the C preprocessor. In a more recent approach, the presentation layer of an editor is used to annotate code (e.g., by colors), instead of adding textual annotations to the program [9]. The advantage is that there can be different views on the source code to show only the code of a certain feature or feature combination [23]. For example, in order to analyze the multi-user support of a database system in isolation, a programmer can hide all code of other features, such as of the transaction management. Furthermore, it is possible to generate a variant that contains only the code of some selected features. For correctness, it is checked that only meaningful fragments of a program are assigned to features in order to avoid errors during and after composition [9, 24], but this detail is outside the scope of this paper.

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 the physical approach, mainly large structures such as packages, classes, and methods can be extended; extensions at the method level are difficult [9, 22, 15]. The advantage is that a programmer can achieve real information hiding by defining interfaces [18]. This is not

possible in the virtual approach, which intermixes (colored) code belonging to different features.

We explore the capabilities of a physical and virtual decomposition for separating crosscutting concerns in 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. Feature composition is performed by the superimposition of the software artifacts found inside the involved containment hierarchies. *Superimposition* merges two software artifacts by merging their corresponding substructures based on nominal and structural similarities [10, 11].

In Figure 1, we illustrate a physical decomposition of an expression evaluator consisting of a feature EXPR for the representation of simple expressions including operators for addition and subtraction, a feature EVAL for the evaluation of expressions, and a feature MULT for multiplication. When composed (denoted by '•') with feature EXPR, feature EVAL adds a new function (incl. three equations) to the module introduced by EXPR. Composing feature MULT with the result adds a new constructor to the data type for expressions and a new equation to the evaluation function.

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. More technical details on composition are reported elsewhere [11].

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 others such as Java, C, XML, and JavaCC documents.

In Figure 2, 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

[2] `http://www.fosd.de/fh`

[3] `http://www.fosd.de/cide`

[4] For readability, we have added comments that indicate which lines belong to which features.

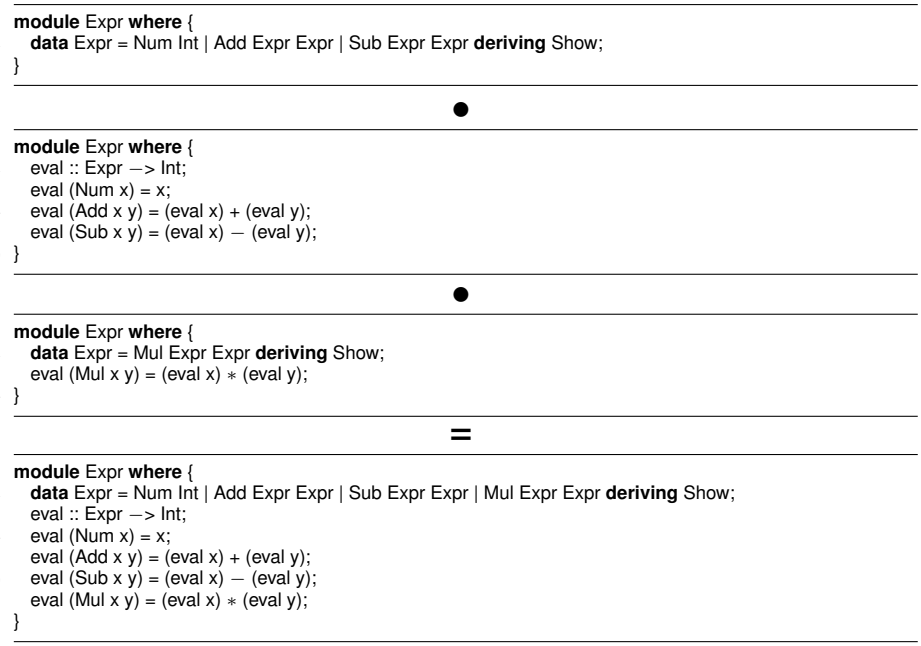

Fig. 1. Composing an expression evaluator from the features EXPR, EVAL, and MULT.

even removed in a generation step. For example, one could hide all code that belongs to features other than EVAL or set the focus on code belonging to MULT.

As mentioned previously, 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, 25]. For example, entire modules, functions, and data types, 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 isolated 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. More technical details on CIDE are reported elsewhere [9].

```
1  module Expr where {
2    data Expr = Num Int | Add Expr Expr
3              | Sub Expr Expr | Mul Expr Expr        // Feature MULT
4              deriving Show;
5    eval :: Expr -> Int;                             // Feature EVAL
6    eval (Num x) = x;                                // Feature EVAL
7    eval (Add x y) = (eval x) + (eval y);            // Feature EVAL
8    eval (Sub x y) = (eval x) - (eval y);            // Feature EVAL
9    eval (Mul x y) = (eval x) * (eval y);            // Feature MULT
10 }
```

Fig. 2. Colored version of the expression evaluator (EXPR, EVAL, MULT).

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.[5] Our goal has been to explore the capabilities of feature decomposition for separating crosscutting concerns in functional programs.

5.1 Physical Decomposition with FEATUREHOUSE

Arith. We have decomposed Arith into the 13 concerns described above. For reasons we explain shortly, we required multiple feature modules 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 and data type definitions to the Arith base program and extends existing functions by new equations and existing data types by new constructors. For example, 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 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, although this is more an exception than the rule. 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) = ...
```

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

When refining modules via superimposition, 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 between two equations is problematic. With superimposition and its implementation in FEATUREHOUSE there is no linguistic means to express this kind of extension properly. This 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 for statements in Java and other languages [9]. A workaround, called *sandwiching* [19], is to split the target module exactly at the position at which we want to refine it. We had to use this workaround twice in 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, function eval plays a central role in expression evaluation. Depending on the features selected, the definition of the function must vary. For example, if feature BINOP is selected, function eval must contain an equation that processes binary operations:

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

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

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

But, if we select 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 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 versions of function eval, add a new parameter, and pass it to the recursive invocations of eval:[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);
```

[6] 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 still be scattered across multiple monads.

Finally, we found that the number of implemented feature modules 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, 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 another for lambda expressions with strict evaluation order. This problem is also known as *feature interaction problem* [8] or *optional feature problem* [26], and our additional feature implementations are called *lifters* or *derivatives*.

FGL. We have decomposed FGL into the 18 concerns or 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 functions and data types, 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 sandwiching twice in order to extend a function by new equations, but we did not need lifters.

In Table 2, we provide numbers on the implemented feature modules in Arith and FGL. The first thing to observe is that the overall code size in terms of lines of code is higher in the physically decomposed variants than in the corresponding original variants (532 instead of 425 for Arith and 2736 instead of 2573 for FGL). The reasons are, on the one hand, that the use of feature modules imposes an overhead due to the additional syntax and boilerplate code and, on the other hand, that, for some features, there are multiple feature modules in order to cope with feature interactions. It seems that this is the price for decomposing features physically in order to be able to compose them again in different combinations.

In column 'overall' of Table 2, we list the number of elements of a particular element type contained in Arith and FGL. For example, the upper left 2 means that there are two modules in Arith. In column 'extended', we show the number of elements being extended by feature modules. For example, the middle left 2 means that each of the two modules of Arith has been extended by subsequently applied feature modules. In column 'extensions', we show the number of extensions applied to elements of a particular type. For example, the lower left 31 means that the two modules of Arith have been extended 31 times: each module, on average, 16 times.

5.2 Virtual Decomposition with CIDE

Arith. For the virtual decomposition of Arith, we began with a full version containing all functionality and proceeded by coloring code step by step, based on the analysis presented in Section 3. The coloring was straightforward and did not pose any challenges. Compared to the physical decomposition, (1) we were faster, which is due to the simpler process (no actual rewrites to the code) 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

	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.

crosscutting concerns at the monad and equation level, since CIDE is able to color individual parameters or subexpressions, and (4) feature interactions are straightforwardly represented and handled in CIDE with overlapping colors.[7]

Let us explain the virtual decomposition by an example. In the physical decomposition, we traded the possibility of separating feature VAR from the other features for some overhead in code size caused by code replication. This was necessary because superimposition does not support the addition of 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; for readability, we have underlined the code that belongs to feature VAR:

```
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);
```

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

FGL. Like 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 since, in FGL, we found only module-level crosscutting. That is, apart from a faster decomposition, in FGL, virtual decomposition did not outperform physical decomposition.

Overall, we were able to color exactly the concerns of Arith and FGL that we identified in our analysis; see Table 1 for more information. Using CIDE's views, we can analyze concern-related code in isolation and generate different variants of Arith and FGL, which is an unusual but useful form of separation of concerns.

5.3 Discussion

In summary, we made the following observations in our case studies:

[7] A program element that is annotated with multiple colors belongs to multiple features, and it is only present if all of the features in question are present. Hence, overlapping colorings in CIDE represent structural interactions between features [9].

1. There is indeed crosscutting at all levels (module, function, data type, monad).
2. Both a physical and virtual decomposition of Haskell programs into features achieve a proper separation of concerns at different levels of granularity. Compared to a native Haskell implementation, features can be combined more easily in different ways.
3. A physical separation leads to an increase in code size due to different kinds of boilerplate code. Also some features are implemented in more than one module (due to sandwiching and listers/derivatives), which does not contradict the idea of separation of concerns but which we felt is unintuitive and complicated.
4. A too fine-grained virtual decomposition is counter-productive since the colored code is difficult to understand – even using views on the source code.
5. There are concerns that cut across function signatures, equations, and expressions; these require workarounds in a physical decomposition 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.
6. 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, although the order of equations is in most situations irrelevant. A virtual decomposition or workarounds like sandwiching have to be used in these cases. The problem of the lexical order is not only a technical problem caused by the syntax and semantics of Haskell, but a general problem of physical decomposition [11].
7. Feature decomposition is largely orthogonal to data and function decomposition in functional programming. Only in some cases a feature is implemented by exactly one function, data type, or module, e.g., in the case of graph algorithms in FGL.
8. The feature optionality problem occurs also in functional programs and leads to an increased number of containment hierarchies in a physical decomposition (see third point above).

As a final remark, we are not able to judge whether a virtual or a physical separation is superior. Our analysis and our case studies were not intended to answer this question. In a different line of research, we have addressed this issue and identified complementary strengths and weaknesses of virtual and physical decomposition that suggest an integration of both [22].

6 Related Work

Kiczales et al. were among the first to conjecture that crosscutting concerns occur in functional programs. Their approach is 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 [20]: 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 which are simpler and

less expressive than aspect-oriented mechanisms [27]. Almost all previous work focuses on aspect-oriented mechanisms in the context of theory of functional programming languages. 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 its predecessors [28] 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 [29]. They do not aim at the analysis of crosscutting concerns in functional programs but at the relationship of aspect-oriented mechanisms and higher-order functions. Masuhara et al. have developed an aspect-oriented version of Caml, called Aspectual Caml [30]. 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 [31]. This work concentrates mainly on the principle of information hiding, not on the impact of crosscutting in functional programming.

Hofer and Ostermann have offered a simple example of crosscutting in a Haskell program [32]. 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 properly. We found that a virtual decomposition of monad-level crosscutting is feasible.

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, modularizing, and composing crosscutting concerns. 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 incidence 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.

Acknowledgments. This work is being supported in part by the German Research Foundation (DFG), project number AP 206/2-1.

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. Apel, S., Kästner, C., Lengauer, C.: FeatureHouse: Language-Independent, Automated Software Composition. In: Proc. Int. Conf. Software Engineering, IEEE CS (2009)
12. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Engineering **30** (2004) 355–371
13. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
14. Colyer, A., Clement, A.: Large-Scale AOSD for Middleware. In: Proc. Int. Conf. Aspect-Oriented Software Development, ACM Press (2004) 56–65
15. 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
16. Hughes, J.: Why Functional Programming Matters. Comput. J. **32** (1989) 98–107
17. 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
18. Mezini, M., Ostermann, K.: Untangling Crosscutting Models with CAESAR. In: Aspect-Oriented Software Development. Addison-Wesley (2005) 165–199
19. Parnas, D.: Designing Software for Ease of Extension and Contraction. IEEE Trans. Software Engineering **SE-5** (1979) 264–277
20. Apel, S., Leich, T., Saake, G.: Aspectual Feature Modules. IEEE Trans. Software Engineering **34** (2008) 162–180
21. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model Superimposition in Software Product Lines. In: Proc. Int. Conf. Model Transformation, Springer-Verlag (2009)
22. Kästner, C., Apel, S.: Integrating Compositional and Annotative Approaches for Product Line Engineering. In: Proc. GPCE Workshop Modularization, Composition, and Generative Techniques for Product Line Engineering. Number MIP-0804, Dept. of Informatics and Mathematics, University of Passau (2008) 35–40
23. Kästner, C., Apel, S., Trujillo, S.: Visualizing Software Product Line Variabilities in Source Code. In: Proc. SPLC Workshop Visualization in Software Product Line Engineering, Lero, International Science Centre, University of Limerick (2008) 303–313

24. Kästner, C., Apel, S., Trujillo, S., Kuhlemann, M., Batory, D.: Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In: Proc. TOOLS EUROPE, Springer-Verlag (2009)
25. Kästner, C., Apel, S.: Type-checking Software Product Lines – A Formal Approach. In: Proc. Int. Conf. Automated Software Engineering, IEEE CS (2008) 258–267
26. Liu, J., Batory, D., Lengauer, C.: Feature-Oriented Refactoring of Legacy Applications. In: Proc. Int. Conf. Software Engineering, ACM Press (2006) 112–121
27. 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)
28. 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
29. Tucker, D., Krishnamurthi, S.: Pointcuts and Advice in Higher-Order Languages. In: Proc. Int. Conf. Aspect-Oriented Software Development, ACM Press (2003) 158–167
30. Masuhara, H., Tatsuzawa, H., Yonezawa, A.: Aspectual Caml: An Aspect-Oriented Functional Language. In: Proc. Int. Conf. Functional Programming, ACM Press (2005) 320–330
31. Aldrich, J.: Open Modules: Modular Reasoning about Advice. In: Proc. Europ. Conf. Object-Oriented Programming, Springer-Verlag (2005) 144–168
32. Hofer, C., Ostermann, K.: On the Relation of Aspects and Monads. In: Proc. Workshop Foundations of Aspect-Oriented Languages, ACM Press (2007) 27–33

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
choice	2	1	2	0	2
lambda abstraction	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
min spanning tree	1	0	7	0	9
transitive closure	1	0	2	0	2