Detecting Features Through Concept Analysis

Curran Nachbar

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-03-14
August 2003

Detecting Features Through Concept Analysis

Curran Nachbar

Department of Computer Science

Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University

1 Introduction

Concept analysis is a useful technique for identifying feature implementations in an object-oriented program. The task of reading and understanding someone else's code can be arduous and time-consuming, particularly when documentation is scarce and the author adhered to unfamiliar coding conventions. It becomes worse when a program has undergone revisions by multiple authors whose conventions are not identical. Even in the best case, the job can be difficult simply because the relationships between objects in the program are complex, and differences in execution may arise from nuances of the run-time environment. To combat these difficulties, programmers seek ways of both writing "more readable" code and equipping themselves with more powerful tools for reading programs. The readability approach tries to achieve a universal programming idiom, but largely fails, because programming languages are flexible by design. There is nearly always more than one way to tell the computer to do something. If the compiler or interpreter doesn't constrain the programmer to use certain coding conventions, then the conventions are probably not enforced throughout the entire software production process. In addition, once a program is "finished", there is little incentive to expend time and labor bringing it up to *future* coding standards, even if the program is still in use. Thus, the software engineering community continues to seek better tools and techniques for understanding programs at the source level and above.

One of the directions this search has taken is to seek better ways of tying together the design, implementation, testing, and documentation of a program. The rationale behind this approach is that during the design phase of a project, the team is engaged in determining exactly what the program must do, and how it *should* do it; the "correctness" of the following implementation is measured by how well the program actually performs these tasks. During each of these phases, the focus is on units of functionality, or *features*. To borrow a definition from Turner et al. [TFLW99], a feature is "a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective." A feature implementation is the set of program fragments that, taken together, comprise the functionality of the feature. Better software development management results from keeping artifacts from the design, implementation, and testing phases of development linked together. Hence, the software industry needs feature-oriented techniques, and tools that are cognizant of the unifying role features have in the software development cycle.

The difficulty in identifying features in an existing body of source code is that they are very rarely confined to just one function, data structure, or file. Features cross-cut programs, and even in simple systems, they may depend on other features. The problem becomes more complicated when the entire lifespan of a program is taken into account; from a bird's-eye view, the process of design-implementation-testing-deployment is cyclical. Once the software has been put into use, it continues to evolve under two conflicting pressures. To satisfy more requirements, it expands, and to become more reusable, it must be refactored and thus consolidated. In fact, Foote [Foo93] has argued that the complexity is actually fractal: expansion and consolidation are repeated at every level of the code, from the individual classes and modules all the way up to the level of related applications.

The complexity of software evolution and refactoring necessitates the development of methodical, general-purpose techniques for analyzing and understanding existing software systems. Several lines of inquiry have begun to converge on feature-oriented tools and on analyses which examine the behavior of fine-grained system components. In particular, program slicing techniques emerged as an aid to understanding system behavior [Tip95] and as a complement to traditional system design [Pre97].

Other investigators have used various clustering techniques to attempt to identify cross-cutting characteristics of software systems (e.g. bugs that manifest in multiple places). Mehta and Heineman [MH02] adopted a dynamic program slicing technique for feature detection by profiling the source code as it performed under a set of unit tests. Presumably, a program's unit tests will test different features by design, so that when one test fails, one or at most a few feature implementations become suspect. However, the disadvantages of this approach stem from its reliance on the unit tests. If the tests don't exercise the entire program, or if they are not clearly separated into independent feature tests, the data produced by the analysis will be confounded; a more accurate technique must be able to identify the smaller building blocks that make up interdependent features. Worse, if unit tests aren't available at all, then the analysis is unusable. Dynamic program slicing techniques should be augmented by static techniques, to correct for this dependency on artifacts that aren't part of the code itself. Thus, I introduce a new application of an existing static technique — concept analysis — which previously has been applied to the related problem of refactoring. By applying a concept analysis algorithm directly to source code, it is possible to derive a set of candidate feature implementations from the organization of the resulting concept lattice.

2 Methods

2.1 Formal Concept Analysis

Formal concept analysis is based on mathematical order theory. The classical representation was introduced by Garrett Birkhoff [Bir40], who proved that a binary relation (a boolean table) can be transformed into a lattice (and back) without losing any of the properties of either. The lattice view presents the data more compactly than the relation, and for large datasets, this representation often conveys more information to the human viewer than a boolean table would. Ganter and Wille [GW99] adopted the original transformation and formalized the notion of a *concept*, and began to use them for conceptual data analysis.

A binary relation R on two sets O and A is a subset of $O \times A$. In concept analysis, the sets O and A are sets of objects and attributes, respectively. The *formal context* of the analysis is defined as the triple (O, A, R). Table 1 contains the boolean table for an example relation. In this example context, $O = Cities =${"Boston", "Providence", "New York", "Philadelphia", "Seattle"}, and $A = Trivia =$ {"has population greater than one million", "has an NHL team", "has a subway", "established in a British colony", "is in New England", "is in the Mid-Atlantic region", "is in the Pacific Northwest"}.

	Pop1M	NHL	Subway	BritCol	NewEngland	MidAtlantic	PacificNW
Boston		×	×	×	×		
Providence				×	×		
New York	×	×	×			×	
Philadelphia		×	×	×		×	
Seattle							×

Table 1: The binary relation T.

Concept analysis provides a way to organize this information by paying particular attention to the grouping of objects[1] with common attributes. For a set $J \subseteq O$ of objects, let $A_J \equiv \{a \in A \mid \forall\, o \in J,\ (o,a) \in R\}$. Likewise, for a set $K \subseteq A$ of attributes, define $O_K \equiv \{o \in O \mid \forall\, a \in K,\ (o,a) \in R\}$. A_J is the maximal set of all attributes common to the objects in J, and O_K is the maximal set of all objects which have each of the attributes in K. A pair of sets, mutually related in this way, is called a *concept*:

A formal concept of the context (O, A, R) is a pair (J, K) with $J \subseteq O,\ K \subseteq A,\ A_J = K$ and $O_K = J$.

$\mathcal{C}(O, A, R)$ denotes the set of all concepts of the context (O, A, R). Table 2 lists the concepts of our example context.

By defining an order relation on the set of concepts, Ganter and Wille were able to apply lattice theory to concept analysis. If (O_1, A_1) and (O_2, A_2) are concepts of a context, and $O_1 \subseteq O_2$ (which

[1] Throughout this document, I have used the word "object" to refer to the term as used in formal concept analysis, and explicitly avoided using it to refer to Java "objects", except where written as `Object`.

C_0	$O_0 = \{\}$ $A_0 = \{$NHL, BritCol, MidAtlantic, NewEngland, PacificNW, Pop1M, Subway$\}$
C_1	$O_1 = \{$Seattle$\}$ $A_1 = \{$PacificNW$\}$
C_2	$O_2 = \{$Philadelphia$\}$ $A_2 = \{$NHL, BritCol, MidAtlantic, Subway$\}$
C_3	$O_3 = \{$New York$\}$ $A_2 = \{$NHL, MidAtlantic, Pop1M, Subway$\}$
C_4	$O_4 = \{$NewYork, Philadelphia$\}$ $A_4 = \{$NHL, MidAtlantic, Subway$\}$
C_5	$O_5 = \{$Boston$\}$ $A_5 = \{$NHL, BritCol, NewEngland, Subway$\}$
C_6	$O_6 = \{$Boston, Providence$\}$ $A_6 = \{$BritCol, NewEngland$\}$
C_7	$O_7 = \{$Boston, Philadelphia$\}$ $A_7 = \{$NHL, BritCol, Subway$\}$
C_8	$O_8 = \{$Boston, Philadelphia, Providence$\}$ $A_8 = \{$BritCol$\}$
C_9	$O_9 = \{$Boston, NewYork, Philadelphia$\}$ $A_9 = \{$NHL, Subway$\}$
C_{10}	$O_{10} = \{$Boston, NewYork, Philadelphia, Providence, Seattle$\}$ $A_{10} = \{\}$

Table 2: The set of concepts $\mathcal{C}(Cities, Trivia, T)$

is equivalent to the condition that $A_2 \subseteq A_1$), then (O_1, A_1) is called a subconcept of (O_2, A_2) and (O_2, A_2) is called a superconcept of (O_1, A_1). When $\mathcal{C}(O, A, R)$ is partially ordered under the $\leq$ relation, it is called the concept lattice of the context. The line diagram, or Hasse diagram, of the partially ordered set of concepts for $(Cities, Trivia, T)$ is shown in Figure 1.

There are two special nodes in any lattice, called *top* ($\top$) and *bottom* ($\bot$). $\top$ has O as its set of objects (its *extent*), and its attributes are only those enjoyed by all other concepts, $\{a : \forall\ o \in O,\ (o, a) \in R\}$. Because our A does not contain any characteristics common to all the cities in O, $\top$'s set of attributes (its *intent*) is empty. Likewise, $\bot$ has A as its intent, and its extent contains only those objects which occur in all other concepts: $\{o : \forall\ a \in A,\ (o, a) \in R\}$. A subset of an ordered set $(M, \leq)$ in which any two elements are comparable is called a chain; an antichain is a subset in which any two elements are incomparable. In our example, there are exactly eight maximal chains, running all the way from $\top$ to $\bot$. One of the antichains is the subset $\{C_4, C_5, C_7\}$. The subset $\{\top, C_8, C_9, C_6, C_7, C_4, C_5, C_2, C_3, \bot\}$ is neither a chain nor an antichain. However, it may be spoken of as a *sublattice* of the entire concept lattice, as may the chain $\{\top, C_1, \bot\}$. Because concept C_1 has no objects or attributes in common with the other concepts in the lattice, it appears in a different sublattice.

The directional arrows in the diagram are actually only half the picture, so to speak. To traverse the lattice from $\top$ to $\bot$, in the direction of the arrows, is to collect new attributes and shed objects, while to travel in the opposite direction is to collect objects and shed attributes. This duality is not accidental. The inverse relation $\geq$ of an order relation $\leq$ is also an order relation, called the dual order of $\leq$.

2.2 Predecessors' Analyses

Because concept analysis is a purely mathematical way of organizing information, the usefulness of the technique depends on how carefully the sets of objects and attributes are chosen. For source code analysis, the objects are usually some subset of program elements, such as functions, methods, fields, and so forth. The definition of attributes depends on how the set of objects is defined. I considered two different applications of concept analysis and selected one of them as more appropriate for the task of identifying features in object-oriented source code.

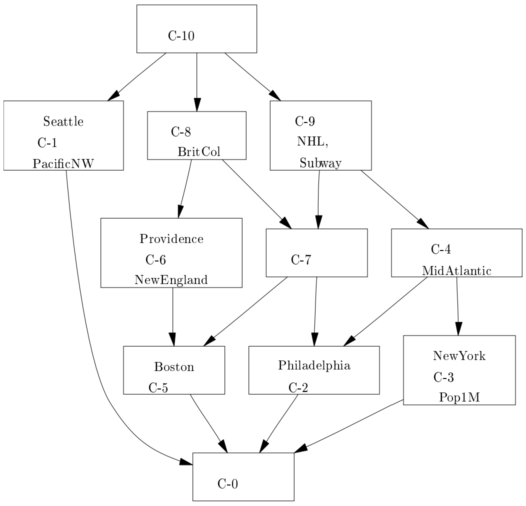


Figure 1: For legibility, only the newly introduced objects and attributes are shown for each concept. $\top = C_{10}$ and $\perp = C_0$.

The analysis suggested by Siff and Reps [SR97] was originally intended for identifying modules in legacy C code. The set of objects was defined as the set of functions in the program, and their attributes were any sort of properties of those functions (the authors suggested attributes such as "has return type τ_1", "has argument type τ_2", "accesses fields of struct σ_1"). This analysis is less useful for feature detection, because it tends to group together functions that have the same argument or return types by coincidence rather than by design. Results could be improved by ignoring `void`, `int`, and other primitive types in the data collection phase, but this would produce varying levels of improvement for different programs, and would have to be tuned by hand for each application.

The analysis proposed by Snelting and Tip [ST00] produces a concept lattice which "reflect[s] the access and subtype relationships between variables, objects, and class members". They suggest that the lattice should be interpreted as a refactored class hierarchy. My intuition was that a feature's presence in a program is spread across the organizational components of the program (methods, classes, modules), and that the objects and attributes should be fine-grained enough to avoid misleading agglomeration. I decided to use the Snelting-Tip analysis and interpret clustering in a lattice representation of a program as an indication of interacting "cliques" of program fragments. Because their method was developed for use with C++ code, I made some minor adjustments for compiling Java programs.

In the Snelting-Tip analysis and my implementation, P denotes a program (or corpus containing several related programs) that contains a class hierarchy.[2] The set of objects contains program variables, which may be variables of a class type defined in P (a "PType"), or instances of class types defined in P. Attributes are declarations and definitions of class members (fields and methods). An association is made between an object and an attribute according to a particular algorithm described below.

[2]Every Java program contains a class hierarchy, because all Java classes are subclasses of the class `Object`.

2.3 The Concept Compiler

The concept compiler traverses the abstract syntax tree (AST) of the input program several times, collecting information about relevant program fragments. A summary of collected fragment sets appears in Figure 2.

PTypes	All class types defined in P.
Instances	All class instances created in P (through the `new` instantiation construct).
DataMembers	All data members (fields) of classes defined in P.
ClassMembers	All members (fields and methods) of classes defined in P.
LocalVariables	All lexically scoped variables in P. This specifically includes method arguments and excludes data members.
ThisPointers	All implicitly or explicitly declared `this` references to the class instance through which a method is invoked. The type of a `this`-pointer is the type of the class in which the method was defined. There is one element of the set *ThisPointers* for each non-`static` method and each constructor defined in P.

Figure 2: The program fragments from which the concept compiler creates the boolean table.

First, the compiler discovers all types defined in P (the set *PTypes*) and their subtype relationships (including those Java-supplied classes of which they may be subtypes). In Java, the subtype relationship between classes has two implications: first, that the subclass may inherit methods and fields from its immediate parent and less immediate superclasses; second, that an instance of the subclass may be substituted wherever an instance of one of its parent types is required. After decorating the tree with certain information, the compiler collects the identity of all variables and methods declared or defined in the program.

Next, the compiler records the relationships between program fragments. In the Snelting-Tip analysis, the set *ClassVars* is made up of all C++ "reference" (not pointer) variables in P. Because Java does not distinguish between pointers and "references" in the sense that C++ does, I approximated the set *ClassVars* with the set of all instances of classes defined in P.

$$\frac{i \in \mathit{Instances} \quad i : \tau \quad \tau \in \mathit{PTypes}}{i \in \mathit{ClassVars}}$$

The set *ClassPtrVars* is comprised of all local variables, class members, and `this`-pointers[3], whose types are class types defined in P.

$$\frac{v \in \mathit{LocalVariables} \quad v : \tau \quad \tau \in \mathit{PTypes}}{v \in \mathit{ClassPtrVars}}$$

$$\frac{m \in \mathit{DataMembers} \quad v : \tau \quad \tau \in \mathit{PTypes}}{m \in \mathit{ClassPtrVars}}$$

$$\frac{p \in \mathit{ThisPointers} \quad v : \tau \quad \tau \in \mathit{PTypes}}{p \in \mathit{ClassPtrVars}}$$

The set $\mathit{ClassVars} \cup \mathit{ClassPtrVars}$ is the set of objects for the concept analysis algorithm.

Before computing the set of attributes, the compiler creates a *PointsTo* map, which maps a member of *ClassPtrVars* to the members of *ClassVars* to which it may refer. In order for variable x to refer to instance y, y's type must be equal to or a subtype of x's type. Thus, this computation relies solely on the type hierarchy and not on any kind of pointer flow analysis.

$$\frac{p \in \mathit{ClassPtrVars} \quad v \in \mathit{ClassVars} \quad p : \tau_1 \quad v : \tau_2 \quad \tau_2 <: \tau_1}{\langle p, v \rangle \in \mathit{PointsTo}}$$

[3]In Java, constructors do have `this`-pointers and may contain "virtual" method calls. This practice is not advised in C++, the language for which Snelting and Tip originally developed their analysis.

The *PointsTo* rule operates under one additional condition: if p is a `this`-pointer of a virtual (non-`static`) method, no overriding definitions of the method are visible in τ_2.

Next, the compiler traverses the AST examining method accesses and variable uses, and recording information about which members are accessed through which variables (and types). This includes accesses made through the implicit `this`, and accesses which cross multiple levels of the class hierarchy. The *MemberAccess* map stores this information.

$$\frac{\texttt{p.m}\text{ occurs in } P \quad \texttt{m}\text{ is a class member} \quad \texttt{p} \in ClassPtrVars}{\langle p, m\rangle \in MemberAccess}$$

$$\frac{\texttt{p.m()}\text{ occurs in } P \quad \texttt{m}\text{ is a non-}\texttt{static}\text{ method} \quad \langle \texttt{p}, y\rangle \in PointsTo}{\langle y, m\rangle \in MemberAccess}$$

During the final data collection step, the compiler gathers a list of all variable assignments. Members of the *Assignments* set have the later effect of connecting program variables that refer to the same object.

$$\frac{\begin{array}{c}\texttt{x = y}\text{ appears in } P \quad x \in ClassPtrVars \\ y \in ClassPtrVars\ \textbf{or}\ y \in ClassVars\end{array}}{(x, y) \in Assignments.}$$

There are certain kinds of information the concept compiler does *not* collect. For instance, consider the following program fragment:

```
if (false) {
   Vendor pete = new Vendor();
   Shipment truck1 = new Shipment("trombones", 76);
   pete.log(truck1);
}
```

Although this block of code would never execute, the local variables `pete` and `truck1` would enter $ClassPtrVars$, an instance of `Shipment` would be added to $ClassVars$, and the access of the method `log(Shipment)` through the `Vendor` variable `pete` would still be recorded.

After all data collection steps, the compiler creates the boolean table. An element of the table acquires a truth for the object and attribute pair (x, y) under the following rules.

The entries derived from member accesses may introduce both `decl` and `def` attributes. Informally, an entry $(y, decl(X.m)) \in T$ means that the class hierarchy must allow the reference y to have knowledge of the declaration, if not the definition, of member m through type X. An entry $(y, def(X.m)) \in T$ means that the reference y must also have knowledge of the definition of m through X, such as when m is a static method or a data member. This approach preserves the access relationships in P.

To accurately model the virtual lookup of non-`static` methods in Java, I retained Snelting and Tip's use of a "static-lookup" algorithm. When a method m is accessed through a particular `Object` y, the run-time type of y determines which definition of m actually executes. At compile time, however, the compiler must make the best inference it can, based on the static type of y. The static-lookup algorithm takes a class C and a member m and returns the base class B containing m; B is either C or a transitive base class of C. With this static-lookup algorithm, the compiler adds four kinds of member-access entries to the table:

$$\frac{\begin{array}{c}\langle y, m\rangle \in MemAccess \quad m \text{ is a data member} \\ X \equiv \text{static-lookup}(TypeOf(P, y), m)\end{array}}{(y, decl(X.m)) \in T}$$

$$\frac{\begin{array}{c}\langle y, m\rangle \in MemAccess \quad m \text{ is a } \texttt{static} \text{ method} \\ X \equiv \text{static-lookup}(TypeOf(P, y), m)\end{array}}{(y, def(X.m)) \in T}$$

$$\frac{\langle y, m\rangle \in MemAccess \quad m \text{ is a non-}\texttt{static}\text{ method} \quad y \in ClassPtrVars \quad X \equiv \text{static-lookup}(TypeOf(P, y), m)}{(y, decl(X.m)) \in T}$$

$$\frac{\langle y, m\rangle \in MemAccess \quad m \text{ is a non-}\texttt{static}\text{ method} \quad y \in ClassVars \quad X \equiv \text{static-lookup}(TypeOf(P, y), m)}{(y, def(X.m)) \in T}$$

The boolean table also acquires entries due to the requirement that the `this`-pointer for a method must be of the same type as the class that defines the method. These `this`-pointer entries follow one rule:

$$\frac{(y_1, def(X.m)) \in T \quad y_2 \equiv \texttt{this}\text{-ptr}(X.m)}{(y_2, def(X.m)) \in T}$$

The member-access and `this`-pointer table entries are the input for two kinds of implications, which add entries to preserve typing relationships from P. While my analysis does not have, as an ultimate goal, the production of a refactored class hierarchy, the dominance/hiding and assignment implications from the Snelting-Tip analysis do record interactions between program fragments; thus, I retained them in the implementation. The implications for assignments in P copy elements from one row of the table to another. The notation $x \rightarrow y$ reads "if $(x, a) \in T$, then $(y, a) \in T$."

$$\frac{(o_1, o_2) \in Assignments}{o_1 \rightarrow o_2}$$

The implications for preserving dominance and hiding in P copy elements from one column of the table to another. Because in P's class hierarchy, a class type may contain more than one member with the same name, there are occasions when member accesses in P rely on the order of the member lookup defined in the Java Language Specification [GJS00]. Snelting and Tip used implications between attributes to preserve the dominance and hiding relationships in a C++ program, and my implementation follows theirs closely. They gave four implication rules for attributes:

$$\frac{(x, decl(A.m)) \in T \quad (x, decl(B.m)) \in T \quad A \text{ is a transitive base class of } B}{decl(B.m) \rightarrow decl(A.m)}$$

$$\frac{(x, decl(A.m)) \in T \quad (x, def(B.m)) \in T \quad A = B \text{ or} A \text{ is a transitive base class of } B}{def(B.m) \rightarrow decl(A.m)}$$

$$\frac{(x, def(A.m)) \in T \quad (x, def(B.m)) \in T \quad A \text{ is a transitive base class of } B}{def(B.m) \rightarrow def(A.m)}$$

$$\frac{(x, def(A.m)) \in T \quad (x, decl(B.m)) \in T \quad A \text{ is a transitive base class of } B}{decl(B.m) \rightarrow def(A.m)}$$

Because an implication across objects may introduce new implications across attributes, and vice versa, the compiler must apply implications to the boolean table iteratively until it reaches a fix-point. When all implications have propagated to the point of exhaustion, the compiler is finished. It produces the table as a text file, which becomes input for the lattice generation step. The Colibri program [Lin00] takes the table and produces a concept lattice (in `dot` format). To layout, view, and manipulate the lattices, I used the `dot` family of digraph tools [Lab].

2.4 Strengths, Weaknesses, Compromises

The reason Snelting and Tip's algorithm is appropriate for feature detection, and the reason I selected it for use, is that it records the member access and subtyping relationships in a program. In order to preserve these relationships in a refactored class hierarchy, it stores exactly the relationships that are actually present in P, ignoring relationships that were initially considered but removed from the design, and incorporating relationships that emerged as the program evolved. Type information is relevant, but the member accesses are foremost in the analysis, in contrast to the [SR97] approach. This makes the analysis more sensitive to code differences at the level of individual statements and expressions, below the level of the class hierarchy, and more useful for feature detection.

One shortcoming of this analysis is that it does not collect quite *all* of the static information it could collect. It is possible to make certain inferences about class accesses based on the static type of objects actually returned from methods (in addition to the information gathered about the declared return type of the method). Also, I did not consider an extension of Snelting and Tip's analysis that would take access modifiers (`public`/`private`/`protected`) of class members into account, although that information is fully available at compile time. Because the Java compiler enforces the correct use of public, private, and protected members, there seems to be no extra information we could glean from the *declarations* of these attributes, when what we really care about is how they are *used*.

In the interest of time, the compiler's current form retains one compromise. In a preprocessing step, I removed all arrays from the input programs and replaced them with objects of type `Array`, which has accessor and modifier methods, and was treated by the concept compiler as a built-in class type. In theory, this could have caused the compiler to miss a few member accesses, but in practice the input programs only used arrays to store built-in types (e.g. `String[]`), and thus no member accesses were lost to this device.

2.5 Interpreting the Output

As described in [ST00], the concept lattice is a refactored class hierarchy that reflects and preserves the access and subtype relationships between program elements. The truth table expresses individual relationships between the types of variables, fields, and methods, such as "the type of x must be a base class of the type of y", and "member m must occur in a base class of the type of variable x", while the concept lattice aggregates variables or members based on the information they have in common. The lattice nodes (i.e., concepts) may be viewed as classes of a restructured class hierarchy that reflects the usage of the original class hierarchy by the client programs. Thus, if one were to analyze a library of related classes, first in conjunction with a client program A, and then in conjunction with a client program B, the resulting lattices might reflect quite different "refactored hierarchies" if the patterns of usage in programs A and B were significantly different.

In a strictly object-oriented framework, temporarily leaving aside language-specific constraints, one may interpret the lattice as a class hierarchy by calling each concept an individual class. According to [ST00], the partial ordering of concepts translates into a hierarchy of inheritance relationships among classes. A variable v would have type C if v appears at concept C in the lattice, and would inherit the types of the concepts above C. A class member m would occur in class C if m appears in concept C.

In a Java world, treating the lattice as a proposed class hierarchy is problematic. The algorithm may produce a lattice in which certain concepts have an in-degree greater than one, indicating multiple inheritance. However, Java does not allow C++-style multiple inheritance. A class may inherit from multiple interfaces, and a class type may be a subtype of many different and unrelated types, but a class may not inherit the implementation of methods from more than one class (even if the method signatures do not collide) [GJS00]. Thus, I have avoided interpreting lattices as class hierarchies altogether, and instead used the lattices to guide the task of feature extraction. If the lattice organizes the program elements into cliques, based on the access and subtype relationships, then one reasonably might use the clusters as a first approximation of program slices for individual features.

2.6 A Concept Lattice Is Not a Dependency Graph

While it might be tempting to interpret a concept lattice produced by one of these analyses as a UML-style dependency graph, or an interaction diagram showing the flow of requests between objects, this is actually misleading.

The connecting lines in the lattice do not represent "is-a" or "has-a" relationships. If a particular concept in a diagram is labeled with the method `foo()`, and there is an edge leading out to another concept labeled with the method `bar()`, it *does not* necessarily mean that `foo()` calls `bar()` in every situation. The connection between the concepts might, in fact, be due to an entirely different relationship between the types of the classes that "own" `foo()` and `bar()`, respectively. It is unwise to look at only certain objects and attributes in a particular concept and ignore the others; the entire lattice could be different if some objects or attributes were removed from only some of the concepts.

2.7 Predictions

A lattice is a reorganization of the information contained in a boolean table. A broad, short lattice (many sublattices of short length) is a representation of a table with many discrete "clumps" of true-values. The arrangement of concepts into many independent sublattices reflects the partition of the rows and columns of the table into many nonoverlapping maximal rectangles. A concept lattice in which there are only a few chains, and the coupling of the chains is high, represents a truth table in which the maximal rectangles are not easily partitioned, but can be organized into super- and sub-rectangles.

Before performing the analysis, we suspected that the lattice for a program would be more or less easily separable into independent program slices, and that connections between sublattices or chains would indicate the composition of smaller feature-groups (perhaps modules or individual classes) into a larger programmatic whole.

3 Experimental Programs

3.1 Characteristics of the Input Programs

I tested the concept compiler with several different Java programs ranging in size from 20 LOC / 1 class (for testing specific stages of the compiler, observing line-by-line execution) to 15.3 kLOC / 154 classes. When the compiler was complete, I compiled a set of student programs to lattices and examined them with an eye towards feature detection. The programs were from two assignments, a text-based Scheme interpreter (`ttyscheme`) and the same interpreter extended with a GUI interface (`guischeme`). Three students' programs were analyzed. Because the `guischeme` assignment was a direct outgrowth of the `ttyscheme` assignment, it was easy to identify ahead of time the set of `ttyscheme` features that should be present in both assignments and the set of `guischeme` features that should only be present in the second assignment.

Of course, the number of lines in each program is a poor indicator of how much functionality each program had. For instance, programs A, B, and C each had roughly the same implementation of a Scheme interpreter, yet program A is significantly larger than the others. Programs A', B', and C' differ more in functionality because students were encouraged to expand on the basic specification, and add features they thought would be useful to a Scheme programmer. So, though each program implements certain basic debugging tools, they differ significantly in that program B has a more elaborate editor and allows the user to evaluate code in multiple windows at once, while program C

Program A:	1546 LOC, 22 classes, 190 concepts
Program B:	832 LOC, 25 classes, 157 concepts
Program C:	953 LOC, 21 classes, 171 concepts
Program A':	1983 LOC, 28 classes, 222 concepts
Program B':	1457 LOC, 33 classes, 244 concepts
Program C':	1723 LOC, 35 classes, 280 concepts

Table 3: Characteristics of the interpreter programs.

adds support for graphical primitives and line drawing to the language. Thus, when comparing the results of the concept analysis across programs, I will focus on two kinds of comparison: that of the three base programs (A, B, and C) to each other, and comparisons of each program to its extended version (X to X').

3.2 Identification of Program Fragments By Category

The appendix contains a broad overview of the `ttyscheme` and `guischeme` features. For detailed information on the Scheme programming language, see [KCR98]. The high-level feature categories in Table 4 serve to differentiate between programs A, B, and C. As a preliminary pass at associating the candidate feature implementations in a lattice with actual user-level features, I assigned each feature category a color, and colored the nodes of the lattice according to the role their objects and/or attributes play in the program. For instance, the concept containing $def\langle \mathit{Scheme.readInput}()\rangle$ was colored red, because this particular method contains the read-eval-print loop (REPL) and catches exceptions arising from parsing and interpretation errors, and because the concept did not contain any objects or attributes that belonged to a different feature category. Overall, there were very few concepts (about 5 in each program's lattice) that contained objects and attributes from different feature categories, and those were colored black to indicate a conflict.

A	A'	B	B'	C	C'	High-level features	
×	×	×	×	×	×	(A)	Scheme language
×	×	×	×	×	×	(B1)	Infrastructure: I/O and parsing
×	×	×	×	×	×	(B2)	Infrastructure: Stack frames
×	×	×	×	×	×	(C1)	Debugging: Stack backtrace
×	×	×	×	×	×	(C2)	Debugging: `break`
×	×	×	×	×	×	(C3)	Debugging: `trace-all` and `untrace-all`
	×		×		×	(D)	Inspector
	×				×	(E1)	Debugging: Stack frame browser
	×		×			(E2)	Debugging: Environment browser
	×		×			(E3)	Debugging: Secondary REPL
			×			(F)	Buffer editor
					×	(G)	Graphical primitives

Table 4: The high-level feature categories present in each input program. Parenthesized labels refer to the notes in Appendix A.

4 Results

4.1 Clustering in the Lattices

The concept lattices turned out to be very broad and flat, with depth no greater than 3 or 4 nodes. There were many chains of length 3 (including $\top$ and $\bot$), which is due to the fact that nearly all the methods in the programs are virtual,[4] and thus the *decl* entries remain separated from the *def* entries in the boolean table and in the concept lattice. In addition, in each lattice there were many independent *clusters* of nodes, made up of small disjoint sets of concepts. The color-by-feature step made two kinds of observations very straightforward. First, it indicated which clusters were involved with multiple feature implementations — perhaps indicating dependent features — by the presence of more than one color in a cluster, or by the presence of black (conflicting) nodes in a particular cluster. Second, it indicated which sublattices made up a complete (high-level) feature implementation when taken together, by the presence of multiple clusters all of the same color.

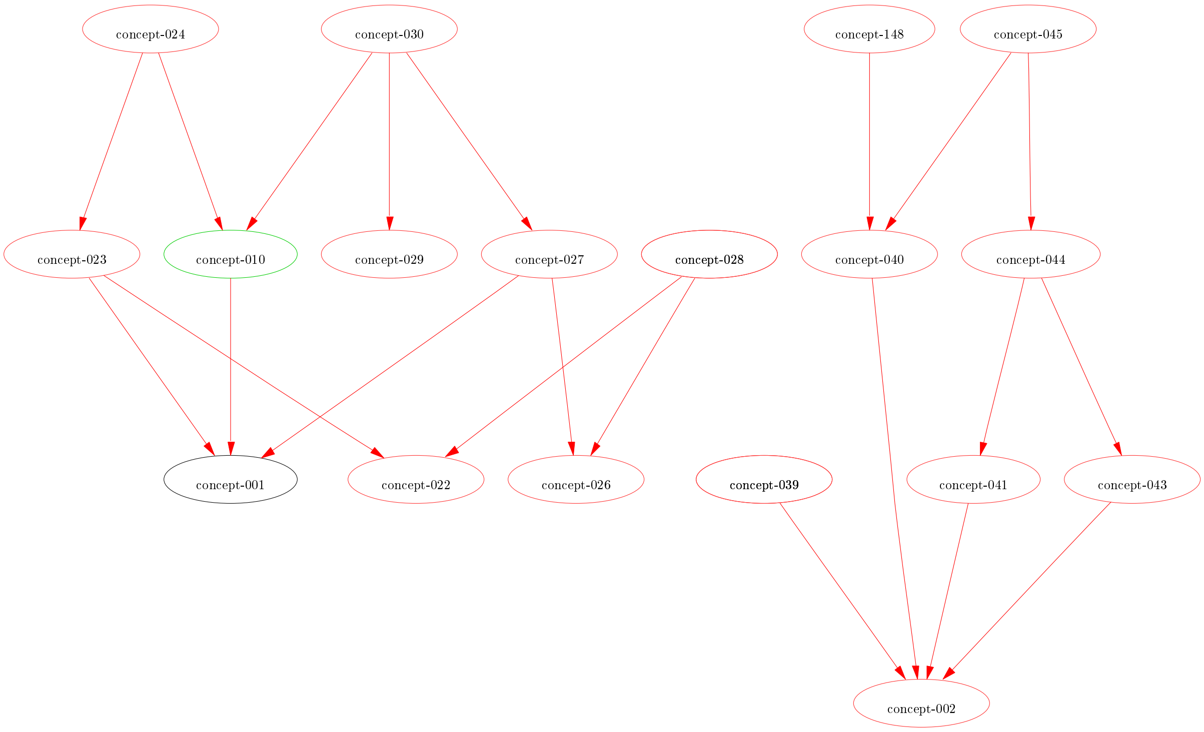


Figure 3: The clusters associated with infrastructure and core program functionality in program B. For space considerations, I have left out $\top$ and $\perp$ in the figures. Any node with an indegree of zero is implicitly an immediate subconcept of $\top$, and any node with an outdegree of zero is implicitly an immediate superconcept of $\perp$.

For instance, the two clusters in Figure 3 are both associated with the data members, methods, and instances of the `Scheme` and `SchemeReader` classes that make up the infrastructure of the program. In the left-hand cluster, these include the methods `Scheme.changeReadLocation()` and `Scheme.readInput()`, the *decl*s and *def*s for these methods, and an instance of the `Scheme` class that was created in the program's `main()` method. In the right-hand cluster, there are the methods `SchemeReader.configureTokenizer()`, `SchemeReader.read()`, `SchemeReader.parseSExp()`, and `SchemeReader.parseSExpListTail()`; the methods' *def*s and *decl*s; and two `SchemeReader` instances created in `Scheme.changeReadLocation()`. The `ttyscheme` program uses delegation to add I/O functionality to the main REPL of the program. While the structure of the concept lattice does not explicitly suggest the use of the delegation pattern, it does collect the relevant program fragments together into clusters.

[4]Java methods are "virtual" by default and must be explicitly declared as `static` to avoid a runtime lookup in the vtable.

As the green and black nodes in the left-hand cluster indicate, the algorithm has also associated some "rogue" program fragments with the fragments that implement I/O and the REPL. The black node (concept-001) contains two `Scheme` instances as objects: one created in `main()`, and one created, as it turns out, in the `SLoadProcedure.apply()` method and assigned to the `sr` variable. The green node (concept-010) contains the object `SLoadProcedure.apply().if.block.reader`, which turns out to be a `SchemeReader` variable in the implementation of the Scheme `load` procedure. This procedure takes a file name as an argument and returns an undefined value; as a side effect, it opens the file by that name and evaluates the contents of the file as Scheme expressions. To do this, it must use a `Scheme` instance and access the `changeReadLocation()` and `readInput()` methods through a `Scheme` variable. Thus, the `load` feature depends on core program functionality, and the presence of the green and black concepts in a primarily red cluster is a clue to the programmer that those two features interact in some way and should be investigated more closely.

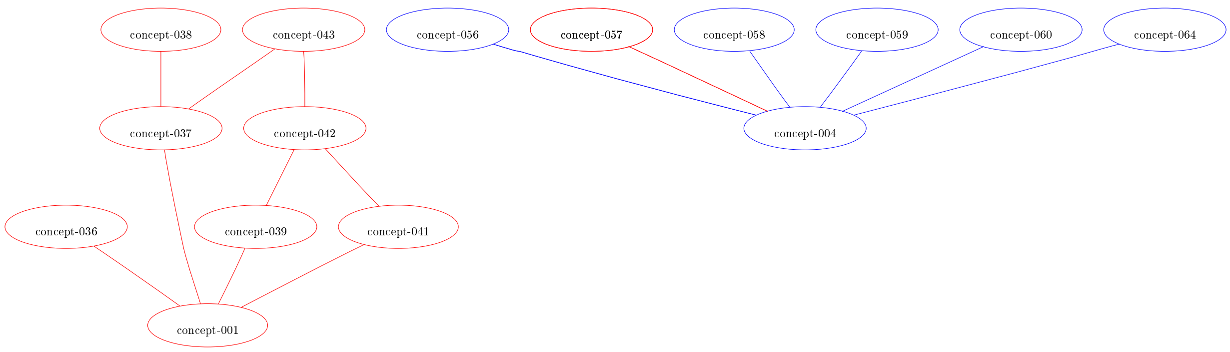


Figure 4: Clusters associated with infrastructure and core program functionality in program C.

In program C, the student did not implement the REPL in a separate method of the `Scheme` class, but instead implemented it directly in `main`. Thus, the cluster on the left side of Figure 3 does not appear in the concept lattice for program C (Figure 4), as the `main()` method simply instantiates a `SchemeReader` and invokes its methods. However, because programs A and B share the same `SchemeReader` class design, the right-hand cluster also appears in program C's lattice.

In program C (Figure 4), the `load` procedure turns up in another cluster as well, indicating a feature interaction. The red node, containing the method `SSpecialProcedure.loadProc()` and its *def*, appears in the midst of several blue-colored concepts, which we associated with the "Scheme language" feature category. Closer inspection reveals that these concepts each contain another method and *def* from `SSpecialProcedure`, which implement the Scheme procedures `quote`, `lambda`, `define`, and other special forms. The concept at the bottom of this fan-shaped cluster contains an `SSpecialProcedure` instance and two *def*s: $def\langle \mathtt{S}SpecialProcedure, \mathtt{S}Procedure.apply()\rangle$ and $def\langle \mathtt{S}SpecialProcedure, \mathtt{S}Exp.eval()\rangle$. As it turns out, the student implemented application of special forms by multiplexing within the `apply()` method (see Figure 5), and calling the `loadProc()` "helper" method when the `load` procedure is invoked. In fact, this fan-shaped cluster is repeated elsewhere in program C's lattice (see Figure 6), and can be interpreted as an indicator of the multiplexing call pattern.

There are two other significant things a programmer should notice because of the fan-shaped clusters. First, the presence of the red or purple concepts in the midst of concepts comprised of Scheme-language program fragments should tell us that the `load` and `trace-all/untrace-all` features are presented to the Scheme programmer *through the Scheme language*, not through a button or pull-down menu, and not through a key command that is captured by the interpreter. In this

```
public class SSpecialProcedure extends SProcedure{
    private String name;

        ...

    public SExp apply(SFrame f) {
        /* Call the appropriate handler */
        if (name.equals("load")) {
            return loadProc(f);
        } else if (name.equals("quote")) {
            return quoteProc(f);
        } else if (name.equals("if")) {
            return ifProc(f);
        } else if (name.equals("define")) {
            return defineProc(f);
        } else if (name.equals("lambda")) {
            return lambdaProc(f);
        } else {
            throw new SchemeInternalError("Unimplemented Special: "
                                          + name);
        }
    }

    /* Loads in the provided file (filename is in a symbol) */
    private Sexp loadProc(SFrame f) { ... }

}
```

Figure 5: Program C's implementation of special forms in the Scheme language.

case, this happens by design and not by accident. However, the second thing the programmer should notice is that the `load` concept appears in the wrong fan-cluster! The `load` procedure is not, in fact, a special form, but an ordinary procedure.[5] In order to find this bug without the concept compiler, a programmer would have to either notice it while reading the code — and the comments left by the student make the placement of `load`'s implementation seem quite deliberate — or test the `load` procedure with an expression that required evaluation in order to be meaningful as a filepath.

4.2 Evidence of Implementation Differences In the Lattices

There were some subtle and some obvious differences among the concept compiler's output for the student programs. Programs A and B produced fairly similar lattices, with certain clusters showing up in both programs. One cluster, in particular, exhibited a nice symmetrical shape and was present in both programs A and B (Figure 7). This cluster is part of the feature implementation for the `car` and `cdr` procedures in the Scheme language. In Scheme, nonempty lists have two parts: the first item in the list (the `car`) and the rest of the list (the `cdr`). If the list contains only one item, the "rest" of the list is an empty list. In the students' implementations, lists can be either empty

[5] In Scheme, the difference between procedures and special forms is that the arguments of a procedure are evaluated before the procedure is applied to them; special forms are "special" because their arguments are *not* evaulated before the form is applied. To appreciate the necessity of this convention, consider the evaluation of the expression `(if #t 'a (quit))`.

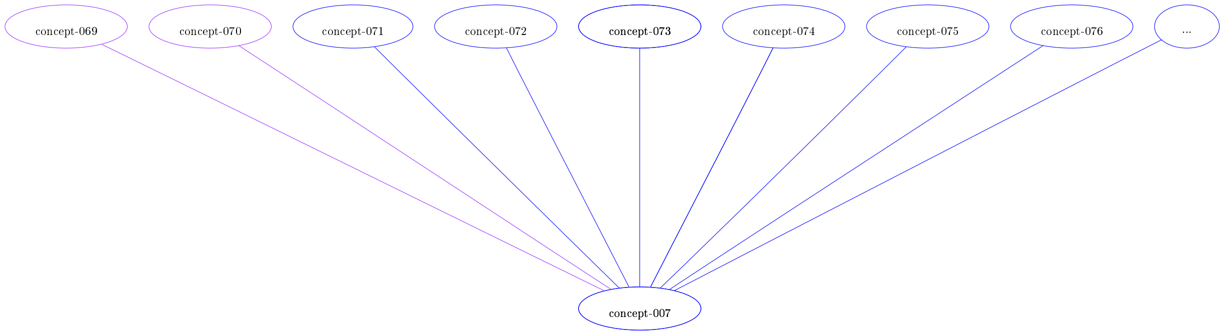


Figure 6: Another multiplexing cluster in program C. In this cluster, the purple concepts are part of the `trace-all`/`untrace-all` implementation.

(`SNil`) or nonempty (`SCons`, so named for the list constructor `cons`). (See Figure 8.) The `SCons` class has two data members: a Scheme expression, `SExp car`, and the rest of the list, `SList cdr`. The symmetry in the cluster reflects the first-rest symmetry of lists in Scheme, and I expected to see an identical symmetrical structure in each of the students' programs.

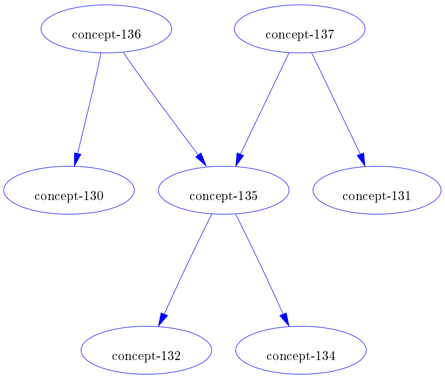


Figure 7: The `car`/`cdr` clusters from programs A, B, and C.

Programs A and B did in fact contain identical clusters for the `car`/`cdr` data members, accessor methods, and mutator methods. However, at first glance it appeared to be missing from program C. A closer examination revealed the symmetrical sublattice did, in fact, exist in program C's concept lattice. The simple structure was obscured because it was embedded *within* a larger cluster, comprised of additional accessor methods for the same `car` and `cdr` fields. At first I thought this was an error, but when I pulled up the implementation of the `SCons` class, I found the following comment:

```
// A bit redundant, but hey.
public SExp car() {
    return getCar();
}
```

The concept analysis technique delivered structurally *similar*, yet still noticeably different, clusters for two similar but significantly different feature implementations. This suggests that concept analysis is a very promising technique for detecting changes in the implementation of specific features as a corpus of code evolves over time, and that it should be used, like regression testing, as a

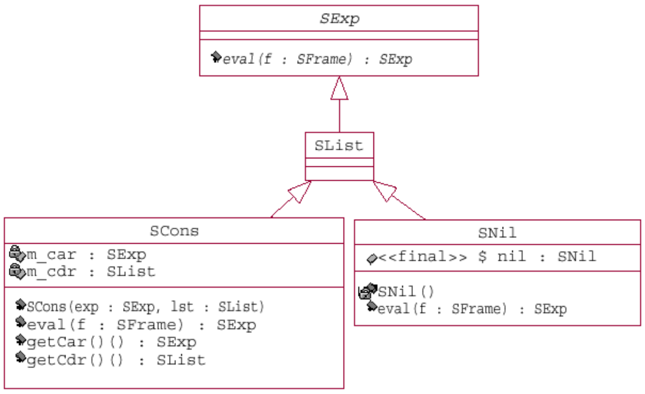


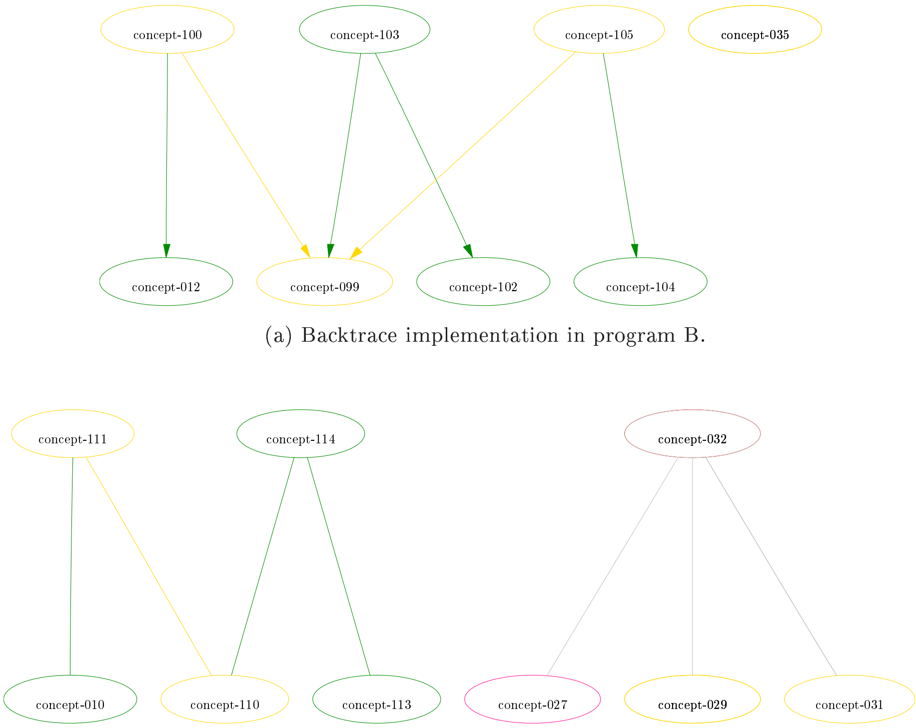
Figure 8: The basic implementation of Scheme lists in **ttyscheme** and **guischeme**.

means of tracking the evolution of a program from a high-level and *feature-oriented* perspective.

The idea of using the concept compiler as a way to follow the evolution of a program is further supported by one of the changes it revealed between programs B and B'. In Figure 9, the differences between the clusters for stack backtrace printing (orange) are striking. As it turns out, the student changed his implementation of the stack backtrace feature between the two versions of the interpreter in non-superficial ways. He modified the recursive method `SFrame.printBacktrace()`, renaming it `backtraceString()` and removing a call to the method `SFrame.printCurrent()` from the `SFrame` constructor. This had the effect of collapsing the yellow $\bigvee$-shaped connections into a single line. Even more noticeable is the change which aggregates several unattached yellow and grey nodes into one cluster. The grey concepts contain program fragments of the InterpretationException, which is thrown whenever an interpretation error occurs; the yellow concepts contain `InterpretationException.getBacktrace()` and `InterpretationException.printBacktrace()` methods and *def*s. In program B', the student converted the `printBacktrace()` method into a `getBacktrace()` method, but left the original husk in the code (a mistake); this is why there are two yellow nodes instead of one. He also added the method `InterpretationException.showEnvBrowser()`, which is a `guischeme` debugging feature. Finally, each of these additional methods accesses the `SFrame` variable `sif`, which is a member of concept-032. Where in program B, only one method accessed `sif`, three methods access it in program B', and the lattice for program B' reflects the increased dependency on this particular program fragment.

5 Comparison to Dynamic Techniques

Since the concept compiler shows promising use as a feature-oriented technique, but at the moment depends heavily on the user's own knowledge of the software she wants to analyze, it would be handy if the feature identification step could be automated. Augmenting the static technique with information derived from another program-slicing analysis might provide the additional boost necessary to color the lattice automatically. To this end, I followed up the concept analysis of the students' Scheme interpreters with a slicing technique styled after [MH02]. Mehta and Heineman performed dynamic slicing, based on feature-separated regression tests, to identify where features are implemented in the program. This approach was convenient partly because we already had a test suite



(a) Backtrace implementation in program B.

(b) Backtrace implementation in Program B'.

Figure 9: The implementation changes between programs B and B', for the backtrace feature.

for the interpreters, and partly because their definition of a "candidate feature implementation" is very similar to ours. By profiling the code that is executed by each test in the test suite, one can collect the set of all statements which fire when a particular feature or group of features is exercised. In [MH02], this is a "feature implementation"; for our purposes, this data may determine to which feature implementation a particular concept belongs.

I performed a unit test analysis (after [MH02]), using an in-house profiler and the JUnit/JFCUnit libraries as the regression test harness [BG98, CWA]. The profiling data was summarized in a table of the percentage of lines exercised in each method by each test case. A portion of program A's table is shown in Figure 5.

Mehta and Heineman used the standard deviation, applied across each row of the table, to group the methods into candidate feature implementations. I suggest using the standard deviation as a first step towards deciding how to color a particular concept. Assuming that the user has separated the tests into feature groups, and assigned a color to each group, my algorithm for coloring the lattice would iterate over the concepts in the lattice and determine the color of the node based on certain properties of the methods it contains.[6] The critical information for each method is contained in a single row of the table. If the standard deviation across the method's row is 0, this indicates that the method is exercised by every test case, and the algorithm assigns it a special "core color" to show that it is probably part of the program infrastructure (such as `main()`, or a common constructor like `SExp()`). If the standard deviation is greater than 0, then the algorithm attempts to determine

[6] The profiler we used did not track the use of data members; any concepts that did not contain at least one method, *def*, or *decl* would be colored grey to indicate that they could not be assigned the color of a feature group.

	Car	Cdr	CarErr3	False	SymbolIf	IfFalse	SymbolUndef	STDEV
SExp.<init>()	1	1	1	1	1	1	1	0
SchemeReader.read()	1	1	1	1	1	1	1	0
SchemeReader.parseSExp()	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0
SPrimitiveProcedure.cdrProc()	0	0.05	0	0	0	0	0	0.02
SPrimitiveProcedure.carProc()	0.05	0	0.05	0	0	0	0	0.03
SSpecialProcedure.ifProc()	0	0	0	0	0	0.09	0	0.03
InterpretationException.<init>()	0	0	0	0	0	0	1	0.38
InterpretationException.<init>()	0	0	1	0	0	0	0	0.38
SFrame.getEnvironment()	1	1	1	0	0	1	0	0.53
SList.<init>()	1	1	1	0	0	1	0	0.53

Table 5: Methods and fraction coverage per test case.

which group of tests exercised the function by further examining the data in that method's row. If exactly three tests exercised, say, 67% of the function, and the user had identified these three tests as belonging to the same feature group, then the method would be assigned the color for that feature group.

The problem gets "interesting" when a concept contains methods exercised by different feature groups. The node should be assigned a meaningful color, yet the algorithm has no way of knowing which feature group is most important to the user (if any is). A simple solution would be to have a conflicting node cycle through the colors of each of the competing feature groups. This would certainly draw the user's attention to that area of the lattice, which is the point. Experience suggests that feature groups are not likely to be completely independent of each other, and in fact, some unit tests will necessarily test basic program functionality each time they test dependent features. The unit tests for the `ttyscheme` and `guischeme` interpreters exhibited this behavior especially strongly, because in order to evaluate the Scheme expression `(+ 3 7)` — a simple test of the addition procedure — the interpreter also had to evaluate two numbers, a symbol, and a list. Therefore, the addition test exercised a superset of the functions the number-evaluation test, symbol-evaulation test, and list-evaluation test exercised. To deal with highly dependent feature groups, an advanced concept analysis tool might offer several coloring algorithms, each tuned for different levels of feature interdependence, and would allow the user to examine the objects and attributes of a concept and override the automatic coloring as he or she saw fit.

In practice, there was a high degree of correlation between clusters I would identify as implementing a particular feature, and the functions the profiling algorithm identified as exercised by a particular feature. This is not surprising, because the Snelting-Tip analysis adds entries to the boolean table when class members are (potentially) accessed through data members or variables, and the Mehta-Heineman analysis records which methods are accessed when the program is actually run. Some divergence is to be expected, because the concept analysis doesn't discriminate between code that might be run and code that definitely will or will not be run, and because the regression test analysis only captures complete information when features are *thoroughly* exercised, and all possible inputs or interactions are handled.

6 Improving the Algorithm

To make the concept analysis algorithm better at clustering feature-related program fragments, I would add more information to the binary relation (boolean table) before computing the lattice. One must be careful to avoid associating program fragments simply because they are defined in the same class, because as a program evolves, several unrelated or minimally related features may end up in the same class definition. A different idea is to make use of static information about methods' return values. The analysis currently interprets the assignment "`foo = bar;`" as an indication that members accessed through the `foo` variable and members accessed through the `bar` variable are, at some point in the program, actually accessed through the same object. Likewise, the statement "`foo = baz();`", taken in conjunction with the method definition

```
public Something baz() {
    Something s = new Something();
    ...
    return s;
}
```

indicates that any accesses made through `foo` are also made through `s`. Taking this logic one step farther, an association should be made between the `this`-pointer of a constructor and all instances of that type.

$$\frac{\begin{array}{c}\texttt{x = new Y(...)}\ \text{appears in}\ P\\ x \in ClassPtrVars \quad Y \in PTypes\\ y \equiv \texttt{this}\text{-ptr}(Y\text{constructor})\end{array}}{(x, y) \in Assignments}$$

This would have the effect of unifying several outlying clusters, particularly the isolated *def*s of "virtual" methods, with the larger clusters containing the *decl*s of the same methods.

One of the drawbacks of the static analysis is that it does not take into account the relative order of program statements. Without actually running the program, it is usually impossible to tell whether certain statements are ever executed, and therefore the concept analysis might be collecting information about assignments and method returns that never actually take place. Therefore, while it is tempting to look for more and more possible inferences about program execution, especially those an experienced programmer will make automatically and (usually) correctly while debugging, it is best to proceed with caution.

7 Using Concept Analysis in Software Development

The concept compiler produces a useful, but incomplete, representation of the input program. To fill out the picture, a concept analysis tool would also tell the user which program fragments did *not* end up in the lattice (unused variables or fields, for instance). If the compiler is combined with a profiling analysis, the tool should show two other significant sets of program fragments: those that show up in the concept lattice but were not exercised by the unit tests, and those which were exercised by the unit tests but did not appear in the lattice. Having this information at the user's fingertips would direct him towards "problem" areas of the code more quickly, saving him the step of deducing this information from the lattice and profiling statistics each time.

A typical scenario of how someone would use the concept compiler is as follows. First, generate the concept lattice and run the unit tests to collect coverage data. Then, the user would have to group the unit tests and, if desired, suggest colors for each group. The tool would then compute which concepts are related to which unit tests, and color them automatically. Finally, the user would employ a "lattice browser" to navigate between the augmented lattice, the source code, and perhaps the profiling data. For instance, the user might begin with a bird's-eye view of the lattice, noticing clusters of similar colors and the shapes and connectedness of those clusters. His attention would be drawn to interdependent feature implementations by the visual clash of different colors or changing colors. To view a particular region in more detail, he could zoom in on a region of the lattice or select a particular node to investigate. An advanced tool would allow the user to select one or many concepts, objects, or attributes, and display the actual source code with those program fragments highlighted. It would be able to hide selected concepts or edges, on demand or according to certain criteria, and be able to search the concept lattice and highlight nodes or program fragments according to the user's wishes (for instance, highlighting all program fragments defined in a particular class). Although the concept lattice organizes and presents the boolean relation in a compact and useful manner, it is still easy to lose track of details when faced with the entire lattice. Augmenting the concept compiler with a full-featured navigation tool would do much to promote its use in the software development community.

This kind of tool, built around the concept compiler, will have several overlapping uses. First, it can speed up the process of familiarizing a programmer with novel code. A user can navigate a software system by examining the program slices of related code, instead of being forced to pick a starting file and wade through the code more or less linearly. The concept lattice provides a scaffolding for the programmer to build an understanding of how the program fragments interact. Second, because the concept analysis algorithm is sensitive to program changes at the level of individual statements and expressions, it provides an excellent way of capturing a "snapshot" of a software system at a given point in time. Successive compilations of the system, taken at different points in its development, provide a development team with a new, feature-oriented way of tracking changes in the system.

8 Related Work

Siff and Reps [SR97] described a general technique for applying concept analysis to source code. Snelting and Tip [ST00] presented a means of refactoring C++ class hierarchies, using concept analysis. The resulting class hierarchy is closely tied to the current state of the implementation, however, and successive refactorings may actually discourage programmers from using the analysis, because of the complexity of large programs' lattices. Additionally, the algorithm is not quite general enough to be applied to other object-oriented languages without careful consideration.

While techniques exist for identifying program slices at runtime, they interpose a layer of indirection between the actual code and the set of candidate feature implementations. Mehta and Heineman [MH02] developed a feature-oriented method for reengineering software systems, which uses unit tests and a profiling analysis to identify feature implementations. Dynamic techniques such as this should be used in conjunction with static analyses such as mine to reduce this dependency on the runtime environment.

9 Conclusion And Future Work

The concept analysis algorithm of [ST00] is an appropriate and useful program slicing technique for identifying feature implementations in object-oriented source code. The algorithm is flexible enough to apply to different kinds of software systems, and it should be refined with additional, carefully chosen, rules for adding objects and attributes. Although a concept lattice is not an isomorph of the source code — you can't reverse-engineer a program from its lattice representation — its structure and shape convey important information, concisely and consistently. As a static technique, it complements the dynamic program slicing approach of [MH02]; it offers new feature-oriented methods for understanding and managing the development of source code.

10 Acknowledgements

Dr. Shriram Krishnamurthi drew my attention first to the field of programming languages, and then to concept analysis in particular. His advice and charisma have spurred each refinement and improvement. Emmanuel Reineris turned many technical roadblocks into detours, and generously shared his bookshelf. My father, Dr. Robert Nachbar, made much-appreciated suggestions for improving the manuscript. And of course, my parents have together supported me from the very beginning.

A Input Program Features

- `ttyscheme` **Features:**

  (A) Scheme language features[KCR98]

  - If a correct Scheme expression is given as input, must print out the correct value of that expression.
  - Handle the types *number, boolean, symbol, list, procedure.*
  - Handle the primitives procedures `+, *, =, car, cdr, cons, null?, eq?, quit`.
  - Handle the special forms `quote, if, define, lambda, load`.
  - Detect interpretation errors (commonly, wrong number or type of arguments are passed to a function).
  - Detect parsing errors, e.g. "`))`".

  (B) Program Infrastructure

  - Provide a read-eval-print loop (REPL) for user input.
  - Ability to read input from a file or from the user.
  - Parsing and interpretation errors must be handled by generating a message and warning the user, not by terminating the program.
  - Perform procedure application based on the stack frame model: each pending application is represented internally as a `(current_procedure, current_environment, parent_frame)` triple; to apply a procedure to its arguments, you extend the birth environment of the procedure with the bindings for its arguments, and then evaluate the body of the procedure in that extended environment. The reference to the parent frame is used for the stack backtrace feature described below.

  (C) Debugging

  - Stack Backtrace: ability to print out the stack of currently pending procedure invocations, upon an error, before returning control to the REPL.
  - When `break` is invoked, print out the stack backtrace before returning control to the REPL.
  - Tracing: `trace-all` and `untrace-all` cause the printing of trace information on every procedure application's arguments and return value.

- `guischeme` **Features:**

  (D) Inspector

  (a) `(inspect` *arg*`)` returns the value of its single argument, and as a side effect, displays a representation of its argument.

  (b) Elements of a list must be selectable and inspectable.

  (E1) Stack Frame Browser — Upon an error, display a window that allows the user to navigate through the call stack and see the procedure that was being evaluated at each stack frame; provide a view of the current environment in each stack frame. (This is just the stack backtrace, but presented interactively — perhaps as a collapsible tree — and with environment information.)

  (E2) Environment Browser — Upon an error, display a window that shows the user all the bindings in the environment of the procedure invocation in which the error occurred; provide a way of accessing and interacting with the parent environment.

(E3) Secondary REPL — Upon an error, display a new interaction window that allows the user to execute code in the environment in which the error occurred. The user must be able to execute code in either the new or original REPL at will (requires a touch of multithreading).

(F) Buffer Editor with *open*, *save*, *execute* buttons. These operations do not extend the language, but make the development environment more useful.

(G) Graphical Primitives: `arc, circle, line, oval, rectangle, string; color-from-name, color-from-rgb, setcolor, getcolor; draw, clear`. These operations extend the language and must be available from within normal Scheme expressions (such as procedure definitions). They should have the side effect of drawing to a window on the screen.

References

[BG98] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3:37–50, 1998.

[Bir40] Garrett Birkhoff. *Lattice Theory*. American Mathematical Society Coll. Pub. 25, 1940.

[CWA] Matt Caswell, Kevin Wilson, and Vijay Aravamudhan. Getting started with jfcunit.

[Foo93] Brian Foote. A fractal model of the life cycle of reusable objects. In *OOPSLA'93 Workshops on Process Standards and Iteration*, 1993.

[GJS00] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java series. Addison-Wesley, 2 edition, 2000.

[GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis : Mathematical foundations*. Springer-Verlag, 1999.

[KCR98] Richard Kelsey, William D. Clinger, and Jonathan Rees. Revised 5 report on the algorithmic language scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

[Lab] AT&T Research Labs. Graphviz: open source graph drawing software.

[Lin00] Christian Lindig. Fast concept analysis. In Bernhard Ganter and Guy W. Mineau, editors, *Proceedings of the 8th International Conference on Conceptual Structures*, volume 1867 of *Lecture Notes in Computer Science*. Springer-Verlag, 8 2000.

[MH02] Alok Mehta and George Heineman. Evolving legacy system features into fine-grained components. In *Proceedings of the International Conference on Software Engineering*, 2002.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241, pages 419–443, Jyväskylä, Finland, 9–13 1997. Springer.

[SR97] Michael Siff and Thomas Reps. Identifying modules via concept analysis. In *Proc. of. the International Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.

[ST00] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. In *ACM Transactions on Programming Languages and Systems*, volume 22, pages 540–582, May 2000.

[TFLW99] C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L. Wolf. A conceptual basis for feature engineering. *The Journal of Systems and Software*, 49(1):3–15, 1999.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.