Evaluating Support for Features in Advanced Modularization Technologies

Roberto E. Lopez-Herrejon, Don Batory, and William Cook
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{rlopez, batory, wcook}@cs.utexas.edu

Abstract. A *software product-line* is a family of related programs. Each program is defined by a unique combination of features, where a *feature* is an increment in program functionality. Modularizing features is difficult, as feature-specific code often cuts across class boundaries. New modularization technologies have been proposed in recent years, but their support for feature modules has not been thoroughly examined. In this paper, we propose a variant of the expression problem as a canonical problem in product-line design. The problem reveals a set of technology-independent properties that feature modules should exhibit. We use these properties to evaluate five technologies: AspectJ, Hyper/J, Jiazzi, Scala, and AHEAD. The results suggest an abstract model of feature composition that is technology-independent and that relates compositional reasoning with algebraic reasoning[1].

1 Introduction

A *feature* is an increment in program functionality [52]. Researchers in software product-lines use features as a defacto standard in distinguishing the individual programs in a product-line, since each program is defined by a unique combination of features [24]. Features are the semantic building blocks of program construction; a product-line model is a set of features and constraints among features that define legal and illegal combinations. Product-line architects reason about programs in terms of features.

Despite their crucial importance, features are rarely modularized. The reason is that feature-specific code often cuts across class and package boundaries, thus requiring the use of preprocessors to wrap feature-specific code fragments in `#if-#endif` statements. While the use of preprocessors works in practice, it is hardly an adequate substitute for proper programming language support. Among the important properties sacrificed are: static typing of feature modules, separate compilation of feature modules, and specifications of feature modules that are independent of the compositions in which they are used (a property critical for reusability). This sacrifice is unacceptable.

In recent years, new technologies have been proposed that have the potential to provide better support for feature modularity. These technologies have very different notions of modularity and composition, and as a consequence are difficult to compare and unify. Thus it is increasingly important to advance standard problems and metrics for technol-

1. This research is sponsored in part by NSF's Science of Design Project #CCF-0438786.

ogy evaluation. A few attempts have been made to compare technologies and evaluate their use to refactor and re-implement systems that are not part of a product family [13][19][29][36]. But for a few studies [16][51][34], the use of new technologies to modularize features in a product line is largely unexplored.

In this paper we present a standard problem that exposes common and fundamental issues that are encountered in feature modularity in product-lines. The problem reveals technology-independent properties that feature modules should exhibit. We use these properties to evaluate solutions written in five novel modularization technologies: AspectJ [1][25], Hyper/J [40][47], Jiazzi [30][31][51], Scala [44][37][38][39], and AHEAD [2][6]. The results suggest a technology-independent model of software composition where the definition and composition of features is governed by algebraic laws. The model provides a framework or set of criteria that a rigorous mathematical presentation should satisfy. Further, it helps reorient the focus on clean and mathematically justifiable abstractions when developing new tool-specific concepts.

2 A Standard Problem: The Expressions Product-Line

The *Expressions Product-Line (EPL)* is based on the extensibility problem also known as the "expression problem" [15][49]. It is a fundamental problem of software design that consists of extending a data abstraction to support a mix of new operations and data representations. It has been widely studied within the context of programming language design, where the focus is achieving data type and operation extensibility in a type-safe manner. Rather than concentrating on that issue, we consider the *design and synthesis* aspects of the problem to produce a family of program variations. More concretely, what features are present in the problem? How can they be modularized? And how can they be composed to build all the programs of the product-line?

2.1 Problem Description

Our product-line is based on Torgersen's expression problem [48]. Our goal is to define data types to represent expressions of the following language:

```
Exp  :: = Lit | Add | Neg
Lit  :: = <non-negative integers>
Add  :: = Exp "+" Exp
Neg  :: = "-" Exp
```

Two operations can be performed on expressions of this grammar:

1) `Print` displays the string value of an expression. The expression `2+3` is represented as a three-node tree with an `Add` node as the root and two `Lit` nodes as leaves. The operation `Print`, applied to this tree, displays the string "`2+3`".

2) `Eval` evaluates expressions and returns their numeric value. Applying the operation `Eval` to the tree of expression `2+3` yields `5` as result.

We add a class `Test` that creates instances of the data type classes and invokes their operations. We include this class to demonstrate additional properties that are important for feature modules. Figure 1 shows the complete Java code for a program of the prod-

uct-line that implements all the data types and operations of EPL. Shortly we will see what the annotations at the beginning of each line mean.

```
lp  interface Exp {
lp    void print();
le    int eval();
lp  }

ap  class Add implements Exp {
ap    Exp left, right;
ap    Add (Exp l, Exp r) {
ap     left = l; right = r; }
ap    void print() {
ap      left.print();
ap      System.out.print("+");
ap      right.print();
ap    }
ae    int eval() {
ae      return left.eval()
ae           + right.eval();
ae    }
ap  }

np  class Neg implements Exp {
np    Exp expr;
np    Neg (Exp e) { expr = e; }
np    void print() {
np      System.out.print("-(");
np      expr.print();
np      System.out.print(")");
np    }
ne    int eval() {
ne      return expr.eval() * -1;
ne    }
np  }
```

```
lp  class Lit implements Exp {
lp    int value;
lp    Lit (int v) { value = v; }
lp    void print() {
lp      System.out.print(value);
lp    }
le    int eval() { return value; }
lp  }

lp  class Test {
lp    Lit ltree;
ap    Add atree;
np    Neg ntree;
lp    Test() {
lp      ltree = new Lit(3);
ap      atree = new Add(ltree, ltree);
np      ntree = new Neg(ltree);
lp    }
lp    void run() {
lp      ltree.print();
ap      atree.print();
np      ntree.print();
le      System.out.println(ltree.eval());
ae      System.out.println(atree.eval());
ne      System.out.println(ntree.eval());
lp    }
lp  }
```

Figure 1. Complete code of the Expressions Product Line

From a product-line perspective, we can identify two different feature sets [17]. The first is that of the operations {`Print`, `Eval`}, and the second is that of the data types {`Lit`, `Add`, `Neg`}. Using these sets, it is possible to synthesize all members of the product-line described in Figure 2 by selecting one or more operations, and one or more data types. For instance, row 4 is the program that contains `Lit` and `Add` with operations `Print` and `Eval`. As with any product-line design, in EPL there are constraints on how features are combined to form programs. For example, all members require `Lit` data type, as literals are the only way to express numbers.

	Operations		Data types		
Program	Print	Eval	Lit	Add	Neg
1	✓		✓		
2	✓	✓	✓		
3	✓		✓	✓	
4	✓	✓	✓	✓	
5	✓		✓		✓
6	✓	✓	✓		✓
7	✓		✓	✓	✓
8	✓	✓	✓	✓	✓

Figure 2. Members of the EPL

A common way to implement features in software product-lines is to use preprocessor declarations to surround the lines of code that are specific to a feature. If we did this for the program in Figure 1, the result would be unreadable. Instead, we use an annotation

at the start of each line to indicate the feature to which the line belongs. This makes it easy to build a preprocessor that receives as input the names of the desired features and strips off from the code of Figure 1 all the lines that belong to unneeded features. As can be imagined, this approach is very brittle for problems of larger scale and complexity. Never the less, the approach can be used as a reference to define what is expected from feature modules in terms of functionality (classes, interfaces, fields, methods, constructors), behaviour (sequence of statements executed), and composition.

2.2 Feature Modularization

A natural representation of the expression problem, and thus for EPL, is a two-dimensional matrix [15][49][22]. The vertical dimension specifies data types and the horizontal dimension specifies operations. Each matrix entry is a *feature module* that implements the operation, described by the column, on the data type, specified by the row. As a naming convention throughout the paper, we identify matrix entries by using the first letters of the row and the column, e.g., the entry at the intersection of row `Add` and column `Print` is named `ap` and implements operation `Print` on data type `Add`. This matrix is shown in Figure 3 where module names are encircled.

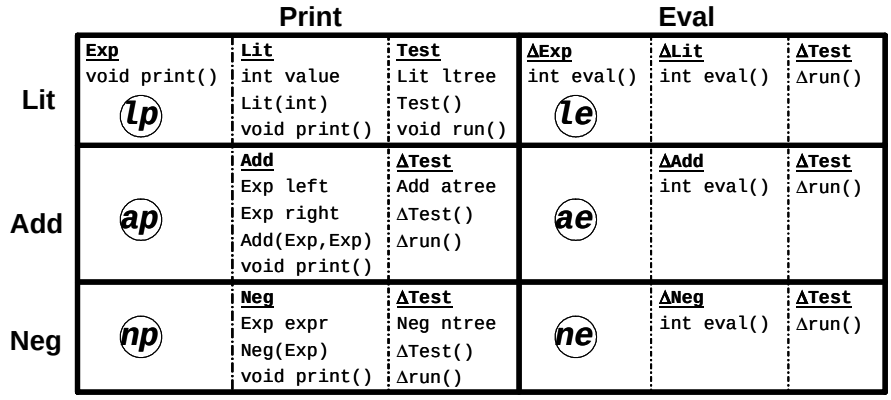


Figure 3. Matrix representation and Requirements

To compose any program from Figure 2, the modules involved are those at the intersection of the selected columns and the selected rows. For example, program number 1, that provides `Print` operation on `Lit`, only requires module `lp`. Another example is program 6, that implements operations `Print` and `Eval` on `Lit` and `Neg` data types, requires modules `lp`, `le`, `np`, and `ne`.

The source code of a feature module are the lines that are annotated with the name of the module. For instance, the contents of feature `ap` include:

a) Class `Add` with `Exp` fields `left` and `right`, a constructor with two `Exp` arguments, and method `void print()`, and

b) An increment to class `Test`, because it is adding something to the class as opposed to contributing a brand new class as is the case of class `Add`. This increment is symbolized by `ΔTest` in Figure 3. It adds: field `atree`, a statement to the body

of the constructor expressed with ΔTest(), and a statement to the body of method run expressed as Δrun().

For clarity we decided to put the Exp interface inside module lp instead of creating a separate row for it. This decision makes sense since the other data types are built using Lit objects. Also, we put the constructors and fields of the data types in column Print instead of refactoring them into a new column and have columns Print and Eval implement only their corresponding methods. Later we will see an interesting consequence of these two design decisions. Additionally, from the design requirements we can infer dependencies and interactions among the feature modules. For instance, if we want to build a program with module ap, we also need to include module lp because ap increments the Test class which is introduced in lp. Later, we briefly discuss this issue as compositional constraints, which are not the focus of this paper. Constraints are discussed in [2][3][9].

3 Basic Properties for Feature Modularity

To give structure to our evaluation, we identify a set of basic properties about features that can readily be inferred from, illustrated by, and assessed in EPL and its solutions in the five technologies evaluated. Conceivably, there are other desirable properties that feature modules should exhibit such as readability, ease of use, etc. However, for sake of simplicity and breadth of scope, they are not part of this evaluation as their objective assessment would require a larger case study that would prevent us from comparing all five technologies together.

The properties are grouped into two categories, covering the basic definition of features and their composition to create programs. The first properties in each category follow from the structure of EPL, while the others come from the studied solutions to EPL and are desirable from the software engineering perspective.

3.1 Feature Definition Properties

The first category of properties relate to the definition of the basic building blocks of EPL, the representation of each piece, and their organization into features.

Program deltas. The code in Figure 1 can be decomposed into a collection of *program deltas* or program fragments. The kinds of program deltas required to solve EPL are summarized in Figure 3, and include:

- *New Classes*, for example Lit in module lp.
- *New Interfaces*, for example Exp in module lp.
- *New fields* that are added to existing classes, like field atree in module ap is added to class Test.
- *New methods* that are added to existing interfaces, like eval() in module le is added to interface Exp.
- *Method extensions* that add statements to methods. For example, extension to method run(), expressed by Δrun(), in all modules except lp.

- *Constructor extensions* that add statements to constructors. For instance, extensions to constructor `Test()`, expressed by `ΔTest()`, in modules `ap` and `np`.

There are other program deltas, such as new constructors, new static initializers, new exception handlers, etc. that are not needed for implementing EPL and thus are not considered in this paper. Nonetheless, we believe that EPL contains a sufficient set of program deltas for an effective evaluation.

Cohesion. It must be possible to collect a set of program deltas and assign them a name so that they can be identified and manipulated as a cohesive module.

Separate compilation. Separate compilation of features is useful for two practical reasons: a) it allows debugging of feature implementation (catching syntax errors) in isolation, and b) it permits the distribution of bytecode instead of source code.

3.2 Feature Composition Properties

Once a set of feature modules has been defined, it must be possible to compose them to build all the specific programs in the Expression Product Line.

Flexible Composition. The implementation of a feature module should be syntactically independent of the composition in which it is used. In other words, a fixed composition should not be hardwired into a feature module. Flexible composition improves reusability of modules for constructing a family of programs.

Flexible Order. The order in which features are composed can affect the resulting program. For instance, in EPL, the order of test statements in method `run()` affects the output of the program. The program in Figure 1 is the result of one possible ordering of features, namely (`lp`, `ap`, `np`, `le`, `ae`, `ne`). Another plausible order in EPL is to have expressions printed and evaluated consecutively, as in order (`lp`, `le`, `ap`, `ae`, `np`, `ne`). Hence, feature modules should be composable in different orders.

Closure under Composition. Feature modules are closed under composition if one or more features can be composed to make a new composite feature. Composite features must be usable in all contexts where basic features are allowed. In EPL, it would be natural to compose the `Lit` and `Neg` representations to form a `LitNeg` feature which represents positive and negative numbers.

Static Typing. Feature modules and their composition are subject to static typing which helps to ensure that both are well-defined, for example, preventing method-not-found errors. We base the evaluation of this property on the availability of a formal typing theory or mechanism behind each technology.

Using these properties we evaluate AspectJ, Hyper/J, Jiazzi, Scala, and AHEAD in the following sections. The complete program listings are presented in the Appendix. We use a concrete example to illustrate these alternatives, i.e. the program that supports `Print` and `Eval` operations in `Lit` and `Add` data types (program number 4 in Figure 2). Thus, the program has four modules: `lp`, `ap`, `le`, and `ae` that we compose in this order (the same as in Figure 1). Throughout the paper, we call this program `LitAdd`.

4 AspectJ

An aspect, as implemented in *AspectJ*[2] [1][25], modularizes a *cross-cut* as it contains code that can extend several classes and interfaces.

4.1 Feature modules and their composition

The implementation of module `lp` is straightforward as it consists of Java interface `Exp` and classes `Lit` and `Test`. In AspectJ literature, programs written using only pure Java code are called *base code*. In Figure 4a, the names of files that are base code are shown in italics, while those of *aspect code* are shown in all capital letters.

	Print	Eval
Lit	*Exp, Lit, Test*	LE
Add	*Add*, AP	AE
Neg	*Neg*, NP	NE

(a)

```
public aspect LE {

  // ΔExp interface
  public abstract int Exp.eval();

  // ΔLit class
  public int Lit.eval() { return value; }

  // ΔTest, advice that implements Δrun()
  pointcut LPRun(Test t):
     execution (public void Test.run())
     && target(t);

  void around(Test t) : LPRun(t) {
    proceed(t);
    System.out.println("= "  + t.ltree.eval());
  }
}
```

(b)

Figure 4. AspectJ Solution

Alternatively, we could have declared the new classes and interfaces as nested elements of an aspect. However, they would be subject to the instantiation of their containing aspect, and their references would be qualified with the aspect name where they are declared. For these reasons, we decided to implement classes and interfaces in separate files.

From Figure 3, module `le`:

1) adds method `eval()` to interface `Exp`,

2) adds the implementation of `eval()` to class `Lit`, and

3) appends a statement to method `run()` of class `Test` that calls `eval()` on field `ltree`.

The entire code of module `le` is implemented with the aspect shown in Figure 4b. The first two requirements use AspectJ's *inter-type declaration*, which is part of its *static crosscutting* model [1][25][3]. Method extensions, like that of the third requirement, can-

2. We used AspectJ version 1.1 for our evaluation.

not be implemented as inter-type declarations because members with the same signature can be introduced only once. Hence, to implement the last requirement it is necessary to utilize AspectJ's *dynamic crosscutting* model which permits adding code (*advice*) at particular points in the execution of a program (*join points*) that are specified through a predicate (*pointcut*).

Since it is required to execute an additional statement when method `run()` is executed, we must capture the join point of the `execution` of that method. Also, since the statement to add is a method call on field `ltree` of class `Test`, we must get a hold of the object that is the `target` of the execution of method `run()` to access its `ltree` field. These two conditions are expressed in pointcut `LPRun` of Figure 4b, where `t` is the reference to the target object. Lastly, to add the extension statement we use an `around` advice. This type of advice executes instead of the join points of the pointcut, but it allows its execution by calling AspectJ's special method `proceed`. We add the new statement to `run()` after the call to method `proceed(t)`.[4]

The implementation of feature module `ap` (not shown in Figure 4) uses two files. The first is a Java class to implement data type `Add`. The second is an aspect to implement the extensions to class `Test`. The first extension adds a new field to class `Test`. This is done also using inter-type declaration in the following way:

```
public Add Test.atree;
```

The other two extensions of module `ap`, Δ`Test()` and Δ`run()`, are implemented in a similar way to those of module `le`. The other modules `ae`, `np`, and `ne` have an analogous implementation.

To compose program `LitAdd`, the AspectJ compiler (*weaver*) `ajc`, requires the file names of the base code and the aspects of the feature modules. The composition is specified as follows, where the order of the terms is inconsequential:

```
ajc Exp.java Lit.java Test.java LE.java Add.java AP.java AE.java
   -outjar LitAdd.jar                                              (1)
```

The static crosscutting model of AspectJ has a simple realization that does not depend on order, namely, members can only be introduced once. However, in the case of dynamic crosscutting, i.e. pointcuts and advice, several pieces of advice can apply to the same join point. In such cases, the order in which advice code is executed is in general undefined[5]. This means that a programmer cannot know a priori, by simply looking at the pointcut and advice code, in what order advice is applied. In program `LitAdd`, this

---

3. We could also implement the first requirement as follows:

   ```
   public int Exp.eval() { return 0; }
   ```

   This alternative defines a default value for the method which can be subsequently overridden by each class that implements Exp.

4. Method `proceed`, has the same arguments as the advice where it is used.

5. There are special rules that apply for certain types of advice when advices are defined in either the same aspect or in others [1]. These rules help determine the order in few cases but not in general.

issue is manifested in the order of execution of method `run()` and its extensions. The order that we want is that of Figure 1, namely, first the statement from `lp` followed by those of `ap`, `le` and `ae`. However, the order obtained by executing the program is statements from `lp`, `ae`, `ap`, and `le`[6].

AspectJ provides a mechanism to give precedence to advice, thus imposing an order, at the aspect level. In other words, it can give precedence to all the advice of an aspect over those of other aspects. To obtain the order that we want for method `run()`, we must define the following aspect:

```
public aspect Ordering {
   declare precedence : AE, LE, AP;
}
```

and add it to the list of files in the specification `(1)`. For further details on how precedence clauses are built, consult [1][26].

4.2 Evaluation

Feature definition. AspectJ can describe all program deltas required for EPL. However, in cases like module `ap` which is implemented with class `Add` and aspect `AP`, there is no way to express that both together form feature `ap`. In other words, AspectJ does not have a cohesion mechanism to group all program deltas together and manipulate them as a single module. Nonetheless, this issue can be addressed with relatively simple tool support. Aspects cannot be compiled separately, as they need have base code in which to be woven.

Feature composition. AspectJ provides flexible composition and order. It can be used to build all members of EPL in the order described in an auxiliary aspect that contains a `declare precedence` clause. This type of clause can also be used inside aspects that implement feature modules, like `LE`, but doing that could reduce order flexibility as the order could be different for different programs where `LE` is used. Feature modules implemented in AspectJ are not closed under composition for two reasons: the absence of a cohesion mechanism and the lack of a general model of aspect composition. The latter is subject of intensive research [18]. Static typing support for AspectJ is also an area of active research [23][50].

5 Hyper/J

Hyper/J [47] is the Java implementation of an approach to *Multi-Dimensional Separation of Concerns* (*MDSoC*) called Hyperspaces [40][46]. A *hyperspace* is a set of units. A unit can be either primitive, such as a field, method, and constructor; or compound such as a classe, interface, and package.

A *hyperslice* is a modularization mechanism that groups all the units that implement a *concern* (a feature in this paper) which consists of a set of classes and interfaces. Hyperslices must be *declaratively complete*. They must have a declaration, that can be

6. In AspectJ version 1.1

incomplete (stub) or abstract, for any unit they reference. Hyperslices are integrated in *hypermodules* to build larger hyperslices or even complete systems.

5.1 Feature modules and their composition

The Hyper/J weaver performs composition at the bytecode level which makes a natural decision to implement each hyperslice (feature module) as a package that can be compiled independently. Hyperslices that contain only new classes and interfaces, like module `lp`, have a straightforward implementation as Java packages. The interesting case is hyperslices that extend units in other hyperslices. For example, Figure 5a shows the package that implements feature `le`. It adds method `eval()` to `Exp` (new method in an interface), the implementation in `Lit` (new method in a class), and a call in method `run()` of class `Test` (method extension).

```
interface Exp {
   int eval();
}
```

```
class Lit implements Exp {
  public int value;  // stub lp
  public Lit (int v) { } // req constructor
  public int eval() { return value; }
}
```

```
class Test {
  Lit ltree; // stub lp
  public void run() {
   System.out.println(ltree.eval());
  }
}
```

(a) Package LE of feature le

```
Hyperspace (hs)
hyperspace  LitAdd
 composable class LP.*;
 composable class LE.*;
 composable class AP.*;
 composable class AE.*;
```

```
Concern Mapping (cm)
package LP : Feature.LP
package LE : Feature.LE
package AP : Feature.AP
package AE : Feature.AE
```

```
Hypermodule  (hm)
hypermodule LitAdd
 hyperslices:
   Feature.LP,
   Feature.AP,
   Feature.LE,
   Feature.AE;
 relationships:
   mergeByName;
end hypermodule;
```

(b) Composition Specification

Figure 5. Hyper/J Implementation

However, extra code is required to make a hyperslice declaratively complete so that it can be compiled. For instance, variable `value` that is introduced in feature `lp` is replicated in class `Lit` so that it can be returned by method `eval()`. Something similar occurs with variable `ltree` in `Test`. Additionally, the Hyper/J weaver requires stubs for non-default constructors. When the package is compiled, the references of these variables are bound to the definitions in the package; however, when composed with other hyperslices that also declare these variables, all the references are bound to a single declaration determined by the composition specification. The extension of methods and constructors is realized by appending the code of their bodies one after the other. The rest of the feature modules are implemented similarly.

The `LitAdd` composition is defined by the three files of Figure 5b: hyperspace `LitAdd.hs`, concern mapping `LitAdd.cm`, and hypermodule `LitAdd.hm`. The hyperspace file lists all the units that participate in the composition. The concern mapping

divides the hyperspace into features (hyperslices) and gives them names. Finally, the hypermodule specifies what hyperslices are composed and what mechanisms (operators) to use. Our example merges units that have the same name.

5.2 Evaluation

Feature definition. Hyper/J's hyperslices can modularize all deltas, treat them as a cohesive unit, and compile them separately. Though, separate compilation requires manual completion of the hyperslices.

Feature composition. Hyper/J provides flexible composition. The order is specified in the hypermodule and can be done using several composition operators [47], thus composition order is flexible. Hyperslices are by definition closed under composition. To the best of our knowledge there is no theory to support static typing of hyperslices.

6 Jiazzi

Jiazzi [30][31][51] is a component system that implements units [21][22] in Java. A *unit* is a container of classes and interfaces. There are two types of units: *atoms*, built from Java programs, and *compounds* built from atoms and other compounds. Units are the modularization mechanism of Jiazzi. Therefore they are the focus of our evaluation.

6.1 Feature modules and their composition

Jiazzi programs use pure Java constructs. Jiazzi groups classes and interfaces in *packages* that are syntactically identical to Java packages. Implementation of modules like `lp` are thus standard Java packages with normal classes and interfaces. Consider the following code contained in package `le` that implements the feature of the same name[7]:

```
public interface Exp extends lp.Exp {
   int eval();
}

public class Lit extends lp.Lit
 implements fixed.Exp {
   public int eval() { return value; }
}

public class Test extends lp.Test {
   public void run() {
     super.run();
     System.out.println(ltree.eval());
   }
}
```

7. Definition of non-default constructors is required but not shown.

Two important things to note are: a) `Exp`, `Lit` and `Test` *extend* their counterparts of feature `lp`, and b) class `Lit` implements `fixed.Exp` which refers to the version of `Exp` that contains all the extensions in a composition.

Package `le` shows how methods can be added to existing classes and interfaces, and how existing methods can be extended. Jiazzi also supports adding new classes, interfaces, constructor extensions, and fields in a similar way to that of normal Java inheritance. The rest of the feature modules are implemented along the lines of module `le`.

Composition in Jiazzi is elaborate. For simplicity, we illustrate unit composition with units `lp` and `le` instead of `LitAdd`. From this readers can infer what the composition of `LitAdd` entails.

We start with the definition of a *signature* which describes the structure of a package, i.e., the interface it exports. The following code is the signature of package `le`[8]:

```
signature leS = l : lpS + {
  package fixed;
  public interface Exp { int eval(); }
  public class Lit { public int eval(); }
  public class Test { public void run(); }
}
```

Two relevant points are: a) the expression `l:lpS +` indicates that `leS` is an extension of signature `lpS`, meaning that `Exp`, `Lit`, and `Test` of `le` extend their counterparts in `lp`, and b) `fixed` is a *package parameter* that is used, as we have seen, in the implementation of `le`. How this parameter is bound is explained shortly.

A unit definition consists of import and export packages followed (if necessary) by a series of statements that establish relations among the packages which, in the case of compound units, determines the order in which units are composed. Each of the features in our problem is implemented by an atom, and a program in the EPL is expressed by a compound unit. The following code defines unit `le`:

```
atom le {
  import lp : lpS;
  export le extends lp : leS;
  import fixed extends le;
}
```

It asserts that atom `le` imports package `lp` with signature `lpS` and that it exports package `le` of signature `leS` which is an extension of `lp`. It also states that it imports package `fixed`, an extension of `le` which is bound, at composition time, to the package parameter of the same name in the signature.

Jiazzi supports composition through the *Open Class Pattern* [30][31]. The key element of this pattern is the creation of a package, called `fixed` in our example, that contains all the extensions made by the units. This package is imported by the atom units, creat-

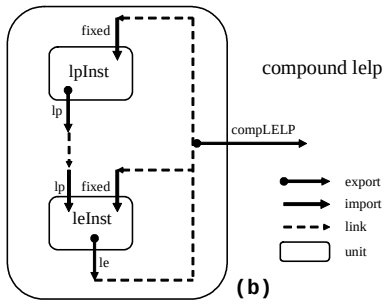
8. For convention in this section, we form signature names with the names of the packages they described followed by a S.

ing a feedback loop that permits them to refer to the most extended version of the classes and interfaces involved in a composition.

Figure 6a shows the code that composes these two units. Figure 6b illustrates this composition. Consider the second part of the specification first. It states that the composition contains two units (line 3): `lpInst` an instance of unit `lp`, and `leInst` an instance of unit `le`. The packages of these two units are linked as follows: a) line 4 states that the exported package `le` of `leInst` is bound to all the `fixed` packages in the compound, b) line 5 sets the link between the export package `lp` of `lpInst` to the import package `lp` of `leInst`.

```
compound lelp {
  export compLELP : leS;                    (1)
  bind package compLELP to compLELP@fixed; (2)
  }{
    link unit lpInst : lp, leInst : le;     (3)
    link package
        leInst@le to *@fixed,               (4)
        lpInst@lp to leInst@lp,             (5)
        leInst@le to compLELP;              (6)
}
```

(a)



(b)

Figure 6. Jiazzi Composition of le and lp

To be useful, compound packages must export something, in our case it exports a package that we named `compLELP` with signature `leS` (line 1) which is linked to package `le` of unit `leInst` in line 6. Since `compLELP` has signature `leS` that contains package parameter `fixed` we must bind it, in this case to itself, as done in line 2.

Signatures allow separate unit compilation. Jiazzi provides a stub generator that uses the unit's signature to create the packages and the code skeletons of the classes and interfaces required to compile the unit. It also provides a linker that checks that the compiled unit conforms to the unit's signature and stores the unit's binaries and signature into a Java archive (`jar`) file that can be used to compose with other units. For further details on the stub generator and linker refer to [32].

6.2 Evaluation

Feature definition. Jiazzi units can modularize all program deltas of EPL in a cohesive way. Furthermore, signatures allow separate compilation.

Feature composition. Jiazzi separates clearly the implementation of features from their composition thus provides a flexible composition. The order of unit composition is determined by the linking statements in compound units and therefore it is flexible. By definition, units are closed under composition. Jiazzi is backed up with a formal theory for type checking units and their compositions [21][22]. This theory permits the linker to statically check and report errors in program composition.

Jiazzi's type checking and separate compilation come with a price. Defining signatures and wiring the relationships between units is a non-trivial task, especially when dealing with multiple units with complex relations among them [51].

7 Scala

Scala is a strongly-typed language that fuses concepts from object-oriented programming and functional programming [44][37]. Though Scala borrows from Java, it is not an extension of it. We included Scala[9] in our evaluation because it supports two non-traditional modularization mechanisms: traits [43] and mixins [10].

7.1 Feature modules and their composition

A trait in Scala can be regarded as an abstract class without state and parameterized constructors. It can implement methods and contain inner classes and traits. We implemented each feature module by a trait. Consider the implementation of feature `lp` shown in Figure 7a. The trait contains:

- Abstract type `exp` with upper bound `Exp`. This means that `exp` is at least a subtype of `Exp` and thus it leaves `exp` open for further extensions by other features.
- Trait `Exp` declares method `print()`. A trait is used in this context because it is roughly equivalent to a Java interface, as it declares a type with methods whose implementations are not yet defined.
- Class `Lit` extends `Exp`.[10] It has a *primary constructor* (or main constructor) that receives an integer which is assigned to field `value`. It also provides an implementation for method `print()` that displays this field.
- Class `Test` contains abstract field `ltree` of abstract type `exp`. Because of this, class `Test` is also `abstract`. `Test` also contains method `run()` that calls method `print()` on `ltree`.

Trait `ap` is implemented as an extension of trait `lp` , shown in Figure 7b, that contains:

- Class `Add` that extends trait `Exp` of module `lp`. It has a two parameter constructor to initialize the expression fields and the implementation of method `print()`.
- Extension to class `Test`, that adds field `atree` and extends method `run()` with the call to `print()` on this field[11]. This class is also `abstract` because `atree`'s type is abstract.

Trait `le` is also implemented as an extension to trait `lp` and is shown in Figure 7c. This trait has:

---

9. We used version 1.3.0.10 for our evaluation.

10. Scala traits are conceptually not different from classes so that is why we use an extends clause instead of implements.

11. To prevent inadvertent overriding, Scala requires overriding methods to include an `override` modifier as part of their definitions. Notice also that the overridden method can still be called using super as in Java.

```
package epl;
trait lp {
  type exp <: Exp;
  trait Exp {
    def print(): unit;
  }
  class Lit(v: int) extends Exp {
    val value = v;
    def print(): unit = System.out.print(value);
  }
  abstract class Test {
    val ltree: exp;
    def run(): unit = { ltree.print(); }
  }
}
```

(a)

```
package epl;
trait ap extends lp {
  class Add(l: exp, r: exp) extends super.Exp {
    val left = l; val right = r;
    def print(): unit = {
      left.print(); System.out.print("+");
     right.print();
    }
  }
  abstract class Test extends super.Test {
    val atree: exp;
    override def run(): unit = {
      super.run(); atree.print();
    }
  }
}
```

(b)

```
package epl;
trait le extends lp {
  type exp <: Exp;
  trait Exp extends super.Exp {
    def eval(): int
  }
  class Lit(v: int) extends super.Lit(v) with Exp {
    def eval(): int = value;
  }
  abstract class Test extends super.Test {
    override def run(): unit = {
      super.run();
      System.out.println(ltree.eval());
    }
  }
}
```

(c)

```
package epl;
trait ae extends ap with le {
  class Add(l: exp, r: exp) extends super.Add(l, r)
                                  with Exp   {
    def eval(): int = left.eval() + right.eval()
  }

  abstract class Test extends super.Test {
    override def run(): unit = {
      super.run();
      System.out.println(atree.eval());
    }
  }
}
```

(d)

```
package epl;
abstract class Test1 extends lp with ap {
  abstract class Test extends super.Test with super[ap].Test;
}
abstract class Test2 extends Test1 with le {
  abstract class Test extends super.Test with super[le].Test;
}
abstract class Test3 extends Test2 with ae {
  abstract class Test extends super.Test with super[ae].Test;
}
object LitAddObj extends Test3 {
  type exp = Exp;
  class Test  extends super.Test {
    val ltree = new Lit(3);
    val atree = new Add(ltree, new Lit(7));
  }
  def main(args: Array[String]) : unit = {
    var test = new Test();
    test.run();
  }
}
```

(e)

Figure 7. Scala Solution

- Trait `Exp` extends `Exp` of feature `lp` by adding method `eval()`.
- Abstract type `exp` that extends `exp` of feature `lp`, meaning that `exp` is now at least a subtype of `Exp` that has `print()` and `eval()` methods.

- An extension of class `Lit`. This class uses *mixin composition* (expressed as `with Exp` in the figure) to indicate that `Lit` is also a subtype of `Exp` and thus it must implement both of its methods. Since it inherits `print()` from trait `lp` it only needs to implement `eval()`.
- An extension of class `Test` that modifies `run()` to invoke `eval()` on `ltree`.

Feature `ae` is implemented as an extension of feature `ap` and a mixin composition with feature `le` because it provides an implementation of method `eval()` for class `Add`. The code is shown in Figure 7d. Additionally this trait extends method `run()` of class `Test`. The other two feature modules of EPL, `np` and `ne`, are implemented similarly.

To define program `LitAdd` is necessary to: a) specify the order in which method extensions are composed, and b) to create an `object`, a singleton object of a new class, to run the program. Figure 7e illustrates this. For the first part, we use *deep mixin composition* [53] (mixin composition at trait level and nested class level), to establish a linear order of `Test` classes as they contain extensions of method `run()`. For the second part, we define `LitAddObj` that extends `Test3` (the most refined abstract `Test` class), binds abstract type `exp` to concrete type `Exp` as defined by `Test3`, and makes concrete class `Test` by creating instances for the test objects `ltree` and `atree`. The main method creates an instance of `Test` and calls method `run()` on it.

7.2 Evaluation

Feature definition. Scala can implement all program deltas of EPL. Regarding cohesion, traits provide a mechanism to collect program deltas under a single name. Separate compilation in Scala requires traits and classes to be placed in named packages, as it is illustrated by package `epl` in Figure 7.

Feature composition. Scala provides flexible composition and flexible order mechanism for implementing EPL. Scala uses inheritance and mixin composition to compose program deltas that add new classes, traits, fields, methods and simple constructor extensions. However, specifying the order of method extensions is a verbose and non-trivial task. Scala traits are closed under composition. Scala is supported by a sophisticated nominal type theory called vObj calculus [39].

8 AHEAD

AHEAD (Algebraic Hierarchical Equations for Application Design) is a feature modularization and composition technology based on step-wise development [6][4][2]. It was created to address the issues of feature-based development of product-lines.

8.1 Feature modules and their composition

AHEAD partitions features into two categories: *constants* that modularize any number of classes and interfaces, and *functions* that modularize classes, interfaces and their extensions.

AHEAD tools use a language, called *Jak* [4][5], that is a superset of Java. The implementation of constant features like `lp`, whose elements are standard classes and interfaces, uses pure Java constructs. To distinguish extensions of these elements, Jak

provides modifier keyword *refines*. Also, to refer to the method being extended, Jak uses the construct `Super.methodName(args)`. For example, here is the Jak code of feature module `le`:

```
refines interface Exp { int eval(); }
refines class Lit implements Exp {
   public int eval() { return value; }
}

refines class Test {
   public void run() {
      Super.run();
      System.out.println( ltree.eval() );
   }
}
```

As described in Figure 3, this feature extends interface `Exp` with method `eval()`, extends class `Lit` with the corresponding implementation, and extends class `Test` by extending method `run()` with a call to `eval()` on `ltree`. `Super.run()` invokes the previously defined method `run()`. In the case of `LitAdd` it calls the `run()` method of `ap`. Constructor extensions follow a similar pattern, as illustrated in the following example, which extends the constructor of `Test` of feature `ap` by assigning variable `atree` a value:

```
refines Test() {
   Add atree = new Add( ltree, ltree );
}
```

The remaining feature modules are implemented in a similar way. Each feature is represented by a directory that contains files for each class and interface definition and extension. The command line to compose these directories to form `LitAdd` is:

```
composer -target=LitAdd lp ap le ae
```

8.2 Evaluation

Feature definition. AHEAD can modularize all EPL program deltas into a cohesive unit. AHEAD provides tools to compile feature modules to bytecode and compose byte-code representations; however, this is not accomplished by separate compilation. Compilation uses global knowledge of all possible classes, interfaces, and members that can be present in a product-line [2].

Feature composition. AHEAD feature modules are independent of the composition. The order in which features are composed is the order in which they are listed on the *composer* command line. AHEAD features are by definition closed under composition. A static typing model of feature modules for AHEAD is under development.

9 Perspective Beyond Individual Technologies

Let us step back from these implementation details to assess the fundamental nature of the problems that are being solved. We have seen that all five technologies can be used to implement EPL and how they satisfy, in different degrees, the properties required by

feature modules. None of these technologies provide a satisfactory solution to the problem of building product lines, that is, they do not meet all the feature properties or express them in a verbose way. However, many common themes can be identified, even as each technology has particular strengths in meeting one or more of the properties.

In this section we show how the properties of feature definition and feature composition can be understood in terms of an algebra of program deltas. This simple algebra is an abstraction designed to express the underlying structure of feature modularization in product-line development. By hiding the details of particular technologies, this abstraction makes it easier to compare and contrast different technologies and suggests areas where the technologies could be improved or generalized. This discussion will, we hope, help encourage reliance on mathematically justifiable abstractions when developing new tool-specific or language-specific concepts [28].

A fundamental concept of metaprogramming is that programs are data and functions (a.k.a. *transforms*) map programs [7][41]. From this starting point, a program delta can be seen as a function that receives a program as input, adds something to it, and returns the extended program as output. Consider `Δrun()` of module `ap`. This delta adds a statement to method `run()` of class `Test` of the program received as input. Another example from `ap` is delta "`Add atree`", which adds member `atree` to class `Test`. For convenience, we refer to functions associated with program deltas by a single name. Thus we omit return types, parameters and their types in our function declarations. Using a mathematical notation, these two deltas are represented as:

```
Δrun(P)-> P′ where P′ is program with Δrun added to run() of Test of P

atree(Q)->Q′, where Q′ has field atree added to class Test of Q
```

When viewed in this way, a feature module like `lp` can be defined by:

```
lp = Test( Lit( Exp( Empty )))                                    (2)
```

where `Empty` is the empty program, and `Exp`, `Lit`, and `Test` are program deltas that add a new interface, and two new classes. To simplify notation further, we write expressions like this using the + operation, because it intuitively conveys the notion that we are building programs incrementally by adding program deltas. `(2)` now becomes:

```
lp = Test + Lit + Exp                                             (3)
```

where evaluation is from right to left. + denotes function composition; base terms are to the right and extensions are to the left. The choice of operator + was deliberately selected as (we will see) it exhibits composition properties that resemble those of elementary algebra. Next, we examine properties of this operator and relate them to the feature properties of Section 3.

Commutativity and Flexible Order. The order in which program deltas can be composed follows two simple rules. First, a program delta that references a data member or method must be composed *after* (to the left of) the delta that introduces that member or method. `(3)` is an example: `Exp` defines an interface, `Lit` adds a class that references this interface, and `Test` adds a class that references the class of `Lit`.

Second, program deltas that extend the same method are not commutative, because if their order is swapped, a different program will result. For example, changing the order in which `print` methods are added to method `run()` of class `Test` alters the output of a program. Summation is *commutative* (`A+B=B+A`) for arbitrary program deltas `A` and `B` if the first rule is not violated and `A` and `B` do not extend the same method. The evaluation property of *flexible order* relies on the non-commutativity property of operation +.

Substitution, Cohesion, and Closure. Module `ap` is defined by:

```
ap = Δrun + ΔTest + atree + Add                                    (4)
```

That is, (reading from right to left) it adds class `Add`, member `atree` to class `Test`, extends the `Test` constructor, and extends method `run`. When we compose `ap` with `lp`, we know the following equality holds because of *substitution* (i.e., replacing equals with equals):

```
ap + lp = (Δrun + ΔTest + atree + Add) + (Test + Lit + Exp)
```

That is, we know that the program produced by adding `ap` to `lp` must equal the sum of their deltas. *Cohesion* is the property that we can assign names `ap` and `lp` to summation expressions. *Closure* is the property that summation of deltas is itself a delta.[12]

Associativity and Flexible Composition. A common situation in product-line design is not only the addition of new features, but a refactoring of existing features into more primitive features.

Recall that in our EPL design, the `Print` operation is implemented in the `Print` column along with the declaration of the data types' fields and constructors. This design prevents, among other things, our ability to build programs without the `Print` operation. The solution is to refactor the `Print` column into two columns: `Print`' that implements operation `Print` exclusively, and `Core` that declares the data types with their fields and constructors. Figure 8 shows the refactoring of module `lp` into its core and non-core parts.

Class `Test` of module `lp` can be decomposed as:

```
Test = Δrun + run + Test^C + ltree + Test^S where
Test^S   is class Test { };
ltree    is Lit ltree;
Test^C   is Test() { ltree = new Lit(3); };
run      is  void run() { };
Δrun     is ltree.print();                                         (5)
```

Superscript `S` stands for skeleton which is the declaration of the class without any members, and superscript `C` stands for constructor. Class `Lit` has a similar decomposition. Interface `Exp` can be decomposed as:

12. Object-oriented classes contain methods that are mutually referential. One can factor each method into an empty (base) method and a program delta that adds the body. In this way, simple algebraic expressions can be written for mutually referential methods.

```
Exp = print^I + Exp^S where
Exp^S     is interface Exp { };
print^I   is void print();                                   (6)
```

Our refactoring `lp` in Figure 8 is captured by the following algebraic derivation:

```
1) lp = Test + Lit + Exp
2) lp =(Δrun + run + Test^C + ltree + Test^S) + (print + Lit^C+ value + Lit^S)
       + (print^I + Exp^S)
3) lp = (Δrun + print + print^I) +
        (run + Test^C + ltree + Test^S + Lit^C+ value + Lit^S + Exp^S)
4) lp = lp' + lp_Core
```

The first step recites `(3)`. The second step substitutes the definitions of `Test`, `Lit` and `Exp` as in `(5)` and `(6)`. The third step rearranges terms using the commutativity properties of summations. The last step uses an *associativity* property of summations (whose proof is simple)[13] and cohesion to express `lp` as a sum of `lp'` and $\texttt{lp}_{\texttt{Core}}$. Similar reasoning is applied to the other modules in the `Print` column to refactor them into a core and non-core part. The ability to refactor expressions is the property of *flexible composition.*

Compositional Reasoning, Static Typing, and Separate Compilation. Compositional reasoning is the ability to prove properties of a program from the properties of its components, which in our case are features, without reference to their implementation [35]. By equating program deltas with functions (summations), we are relating *compositional reasoning* with *algebraic reasoning*. Doing so can be a substantial win for several reasons. First, an algebra provides a clean mathematical foundation for compositional reasoning and automation — both of which are needed in product-line development. Second, it changes our orientation on tool development and creation. Instead of inventing new tools with new abstractions and new conceptual models — e.g., the AspectJ, Hyper/J, Jiazzi, Scala, and AHEAD models are hardly similar and are difficult to compare — we have a single simple algebraic model that imposes clean abstractions *onto* tools, so that we can reason about programs in a tool-implementation-independent way.

(a) lp'

```
// added to Exp
void print();

// added to Lit
void print() {
    System.out.print(value);
}

// added to run() of Test
ltree.print();
```

(b) lp_{Core}

```
interface Exp { }

class Lit implements Exp {
 int value;
 Lit (int v) { value = v; }
}

class Test {
  Lit ltree;
  Test() {
    ltree = new Lit(3);
   }
  void run() {  }
}
```

Figure 8. Refactoring of lp to lp' + lp_{Core}

13. + denotes function composition. Function composition is associative.

Jiazzi provides an example of compositional reasoning: each feature module is statically typed. Jiazzi ensures that the composition of statically typed modules is itself a statically typed module. So not only does Jiazzi compose the *code* of individual features, it also computes (or verifies) an important property of a composition. Similar examples can be given from other technologies. All of this can be given an algebraic foundation. If we want property *p* of a summation, we need a composition operator $+_p$ (read *p*-sum) that tells us how to compose properties of constituent terms. So property *p* of module `lp`, denoted $\texttt{lp}_\texttt{p}$, is a *p*-sum of the *p* properties of its terms:

$$\texttt{lp}_p = \texttt{Test}_\texttt{p} +_p \texttt{Lit}_\texttt{p} +_p \texttt{Exp}_\texttt{p}$$

This idea (although not in an algebraic form) is common in the software architecture and product-line communities [45], and has been demonstrated elsewhere [6]. In the product-line and software architecture literature, feature modules map to *functional* requirements, and properties of modules and their compositions (such as the property of being statically typed) correspond to *non-functional* requirements.

The remaining property in our evaluation, *separate compilation*, is not a property of an algebraic model, but rather an engineering requirement of any implementation of the model.

10 Related Work

Relational query optimization is a classic example of the importance that algebra can play in program specification, construction, and optimization. SQL queries are translated to relational algebra expressions (i.e., compositions of relational algebra operators). A query optimizer rewrites the expression into semantically equivalent expressions where the goal is to minimize the expression (program) execution time. Readers will see that this is an example of compositional reasoning: the relational algebra expression defines the program, the optimizer composes a performance model of each operator to produce a performance model of that program [6].

The expression problem originated in the works of Reynolds [42] and Cook [15]. Torgersen [48] presents a concise summary of the research on this problem and four solutions that utilize Java generics. Though extensive, this literature focuses only on programming language design and separate compilation issues, and not about the requirements of feature modularity.

Masuhara et al. describe a framework to model the crosscutting mechanisms of AspectJ and Hyper/J [29]. Both are viewed as weavers parameterized by two input programs plus additional information such as where, what, and how new code is woven. Their focus is on the implementation of crosscutting semantics rather than on the broader software design implications that these mechanisms have.

Murphy et al. [36] present a limited study that uses AspectJ and Hyper/J to refactor features in two existing programs. The emphasis was on the effect on the program's structure and on the refactoring process, not in providing a general framework for comparison. Along the same lines, Driver [19] describes a re-implementation of a

web-based information system that uses Hyper/J and AspectJ, but the evaluation is subjective and expressed in terms of factors such as extensibility, plugability, productivity, or complexity. Clarke et al. [13] describe how to map crosscutting software designs expressed as composition patterns (extended UML models) to AspectJ and Hyper/J, and evaluate their crosscutting capabilities to implement such patterns.

Coyler et al. [16] focus on refactoring tangled and scattered code into base code and aspects that could be considered as the features of a product line. They indicate that, based on their experience implementing middleware software, concerns (features) are usually a mixture of classes and aspects; a finding that corroborates the importance of feature cohesion.

For our evaluation we considered MultiJava, an extension of Java that supports symmetric multiple dispatch and modular open classes[11][12]. However, its focus is on solving the *augmenting method problem*, that consists on adding operations (methods) to existing type hierarchies. Given this constraint, it is not possible to implement EPL as it cannot add new fields, add new classes and interfaces, and extend existing methods and constructors. Similarly, *Classboxes* [8] are modules that provide method addition and method replacement (overriding without `super` reference). However, it is unclear if classboxes can support other program deltas such as adding new fields, or methods and constructor extensions.

The *Concern Manipulation Environment (CME)* [14] is a project that builds on the experience of Hyper/J and MDSoC. Among its goals is to provide support for the identification, encapsulation, extraction, and composition of concerns (features in this paper). CME architecture is geared towards supporting multiple modularization approaches. Thus it would be interesting to evaluate whether the software composition model we propose in this paper can benefit from the tool support that CME provides.

Mezini and Ostermann [34], present a comparison of variability management in product lines between *Feature-Oriented Programing (FOP)*, as in AHEAD, and Aspect-Oriented Programming, as in AspectJ. They identify as weaknesses in these technologies: a) features are purely hierarchical (extensions are made to some base code), b) support for reuse (extensions are tied to names not functionality), c) support for dynamic configuration (in FOP composition is static), and d) support for variability (aspects are either applied or not to an entire composition). They propose Caesar[33] to address these issues. Caesar relies on Aspect Collaboration Interfaces, or ACIs, which are interface definitions for aspects (Caesar's aspects are similar to AspectJ's) whose purpose is to separate an aspect implementation from its binding. The association between these two is implemented with a *weavelet*, which must be deployed to activate advice either statically, when the object is created, or dynamically, when certain program block is executed. How these ideas could be applied to solve EPL is subject of an ongoing evaluation.

11 Conclusions and Future Work

Features express the kinds of variations product-line developers encounter in program development, because features represent increments in program functionality. Thus, it is natural to consider modularizing features as a way to modularize programs. Unfortunately, the code for features often cuts across classes, and thus traditional modularization schemes do not work well. New program modularization technologies have been proposed in recent years that have shown promise in supporting feature modularity. We have presented a classical problem in product-line design — called the Expressions Product-Line — to identify properties that feature modules should have. We have used these properties to compare and contrast five rather different technologies: AspectJ, Hyper/J, Jiazzi, Scala, and AHEAD. Our results showed that none of these technologies provide a satisfactory solution to the problem of building product-lines.

Instead of debating the merits of particular technologies, we focused on a topic that we believe has greater significance. Namely, product-line architects reason about programs in terms of their features, not in terms of their code or implementing technologies. We proposed an abstract model of features where compositional reasoning was related to algebraic reasoning. We showed how virtually all of the evaluation properties we identified in EPL were actually properties of an algebra. Namely: program deltas are functions that map programs, cohesion and closure under composition are associativity properties of function composition, flexible composition and flexible order is a consequence of the non-commutativity of certain functions, static typing is a property of a function (program delta) and is a property that can be predicted from an expression (i.e., a composition of deltas). Only the property of separate compilation dealt with engineering considerations of the algebra's implementation.

We believe the time has come for programming languages to play a more supportive role in product-lines and feature-based development. A consolidation of different modularization efforts is essential to this objective. We argued that such a consolidation should relate compositional reasoning with algebraic reasoning, because of its clean abstractions, the ability to automate compositional reasoning, and for giving an algebraic justification when adding new modularization concepts.

To continue this effort and because the full potential of the five technologies was not required, we foresee extending EPL and designing other case studies to help derive and illustrate further properties of feature modules (e.g. AOP quantification [28]). We are currently collaborating with proponents of other modularization technologies, such as Composition Filters [20], Caesar [33], and Framed Aspects [27], for this purpose.

Acknowledgments. We thank Sean McDirmid and Bin Xin for their help with Jiazzi, and Martin Odersky for his help with Scala. We are grateful to Axel Rauschmayer and Awais Rashid for their feedback on drafts of the paper, and the anonymous reviewers for their comments.

12 References

1. AspectJ. Programming Guide. `aspectj.org/doc/proguide`

2. AHEAD Tool Suite (ATS). www.cs.utexas.edu/users/schwartz
3. Batory, D., Geraci, B.J,: Composition Validation and Subjectivity in GenVoca Generators. IEEE Trans. Soft. Engr., February (1997) 67-82
4. Batory, D., Lopez-Herrejon, R.E., Martin, J.P.: Generating Product-Lines of Product-Families. Automated Software Engineering Conference (2002)
5. Batory, D., Liu, J., Sarvela, J.N.: Refinements and Multidimensional Separation of Concerns. ACM SIGSOFT, September (2003)
6. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Soft. Engr. June (2004)
7. Baxter, I.D.: Design Maintenance Systems. CACM, Vol. 55, No. 4 (1992) 73-89
8. Bergel, A., Ducasse, S., Wuyts, R.: Classboxes: A Minimal Module Model Supporting Local Rebinding. Joint Modular Languages Conferences JMLC (2003)
9. Beuche, D.:Composition and Construction of Embedded Software Families. Ph.D. Otto-von-Guericke-Universität Magdeburg (2003)
10. Bracha, G., Cook, W.: Mixin-based inheritance. OOPSLA (1990)
11. Clifton, C., Leavens, G.T., Millstein, T., Chambers, G.: MultiJava: Modular Open classes and Symmetric Multiple Dispatch for Java. OOPSLA (2000)
12. Clifton, C., Millstein, T., Leavens, G.T., Chambers, G.: MultiJava: Design Rationale, Compiler Implementation, and User Experience. TR #04-01, Iowa State University (2004)
13. Clarke, S., Walker, R.: Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J. Technical Report UBC-CS-TR-2001-05, University of British Columbia, Canada (2001)
14. Concern Manipulation Environment (CME).www.eclipse.org/cme/
15. Cook, W.R.: Object-Oriented Programming versus Abstract Data Types. Workshop on FOOL, Lecture Notes in Computer Science, Vol. 173. Spring-Verlag, (1990) 151-178
16. Coyler, A., Rashid, A., Blair, G.: On the Separation of Concerns in Program Families. TRCOMP-001-2004, Computing Department, Lancaster University, UK (2004)
17. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
18. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. AOSD (2004)
19. Driver, C.: Evaluation of Aspect-Oriented Software Development for Distributed Systems. Master's Thesis, University of Dublin, Ireland, September (2002)
20. Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley (2004)
21. Flatt, M., Felleisen, M.: Units: Cool modules for HOT languages. PLDI (1998)
22. Findler, R.B., Flatt, M.: Modular Object-Oriented Programming with Units and Mixins. ICFP, (1998) 94-104
23. Jagadeesan, R., Jeffrey, A., Riely, J.: A Typed Calculus of Aspect Oriented Programs. Submitted for publication.
24. Kang, K., et al.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. CMU/SEI-90-TR-21, Carnegie Mellon Univ., Pittsburgh, PA, Nov. (1990)
25. Kiczales, G., Hilsdale, E., Hugunin, J., Kirsten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. ECOOP (2001)
26. Laddad, R.: AspectJ in Action. Practical Aspect-Oriented Programming. Manning (2003)
27. Loughran, N., Rashid, A., Zhang, W., Jarzabek, S.: Supporting Product Line Evolution with Framed Aspects. ACP4IS Workshop, AOSD (2004)

28. Lopez-Herrejon, R.E., Batory, D.: Improving Incremental Development in AspectJ by Bounding Quantification. SPLAT Workshop at AOSD (2005)
29. Masuhara, H., Kiczales, G.: Modeling Crosscuting Aspect-Oriented Mechanisms. ECOOP (2003)
30. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New age components for old-fashioned Java. OOPSLA (2001)
31. McDirmid, S., Hsieh, W.C.: Aspect-Oriented Programming with Jiazzi. AOSD (2003)
32. McDirmid, S., The Jiazzi Manual (2002)
33. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. AOSD (2003)
34. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. SIGSOFT04/ FSE-12 (2004)
35. Misra, J.: A Discipline of Multiprogramming. Springer-Verlag (2001)
36. Murphy, G., Lai, A., Walker, R.J., Robillard, M.P.: Separating Features in Source Code: An Exploratory Study. ICSE (2001)
37. Odersky, M., et al.: An Overview of the Scala Programming Language. September (2004), `scala.epfl.ch`
38. Odersky, M., et al.: The Scala Language Specification. September (2004), `scala.epfl.ch`
39. Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. ECOOP (2003)
40. Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the Hyperspace approach. In Software Architectures and Component Technology, Kluwer (2002)
41. Partsch, H., Steinbrüggen, R.: Program Transformation Systems. ACM Computing Surveys, September (1983)
42. Reynolds, J.C.: User-defined types and procedural data as complementary approaches to data abstraction. Theoretical Aspects of Object-Oriented Programming, MIT Press, (1994)
43. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. ECOOP (2003)
44. Schinz, M.: A Scala tutorial for Java programmers. September (2004), `scala.epfl.ch`
45. Software Engineering Institute. Predictable Assembly from Certified Components. `www.sei.cmu.edu/pacc`
46. Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE (1999) 107-119
47. Tarr, P., Ossher, H.: Hyper/J User and Installation Manual. IBM Corporation (2001)
48. Torgersen, M.: The Expresion Problem Revisited. Four new solutions using generics. ECOOP (2004)
49. Wadler, P.: The expression problem. Posted on the Java Genericity mailing list (1998)
50. Walker, D., Zdancewic, S., Ligatti, J.: A Theory of Aspects. ICFP (2003)
51. Xin, B., McDirmid, S., Eide, E., Hsieh, W.C.: A comparison of Jiazzi and AspectJ. Technical Report TR UUCS-04-001, University of Utah (2004)
52. Zave, P.: FAQ Sheet on Feature Interaction. `www.research.att.com/~pamela/faq.html`
53. Zenger, M., Odersky, M.: Independently Extensible Solutions to the Expression Problem. Technical Report TR IC/2004/33, EPFL Switzerland (2004)

13 Appendix

13.1 AspectJ implementation

Feature lp

```
// Exp.java
public interface Exp {
  void print( );
}
```

```
// Lit.java
class Lit implements Exp {
  public int value;
  Lit (int v) { value = v; }
  public void print() { System.out.print(value); }
}
```

```
// Test.java
public class Test {
  public Lit ltree;
  public Test( ) {   ltree = new Lit(10); }
  public void run( ) { ltree.print( ); }

  public static void main(String[] args) {
    Test test = new Test();
    test.run();
  }
}
```

Feature le

```
// LE.java
public aspect LE {
  public abstract int Exp.eval();
  public int Lit.eval() { return value; }
  pointcut LPRun(Test t):execution(public void Test.run()) &&
                          target(t);
  void around(Test t) : LPRun(t) {
     proceed(t);
     System.out.println(t.ltree.eval());
  }
}
```

Feature ap

```
// Add.java
class Add implements Exp {
  public Exp left, rigth;
  Add (Exp l, Exp r) { left = l; rigth = r; }
  public void print() {
     left.print(); System.out.print("+"); rigth.print();
  }
}
```

```
// AP.java
public aspect AP {
  public Add Test.atree;
  pointcut APTest(Test t): execution(public Test.new()) && target(t);
  void around(Test t) : APTest(t) {
    proceed(t); t.atree = new Add(t.ltree, t.ltree);
 }
 pointcut APRun(Test t): execution (public void Test.run(..)) &&
                         target(t);
 void around(Test t) : APRun(t) {
    proceed(t); t.atree.print();
 }
}
```

Feature ae

```
// AE.java
public aspect AE {
  public int Add.eval() { return left.eval() + rigth.eval(); }
  pointcut AERun(Test t): execution (public void Test.run(..))
                          && target(t);
  void around(Test t) : AERun(t) {
    proceed(t); System.out.println(t.atree.eval());
  }
}
```

Feature np

```
// Neg.java
class Neg implements Exp {
   public Exp expression;
   Neg (Exp e) { expression = e; }
   public void print() { System.out.print("-"); expression.print(); }
}

// NP.java
public aspect NP {
  public Neg Test.ntree;
  pointcut NPTest(Test t): execution(public Test.new(..)) &&
                           target(t);
  void around(Test t) : NPTest(t) {
    proceed(t); t.ntree = new Neg(t.ltree);
  }
  pointcut NPRun(Test t): execution (public void Test.run(..)) &&
                            target(t);
  void around(Test t) : NPRun(t) { proceed(t); t.ntree.print(); }
}
```

Feature ne

```
// NE.java
public aspect NE {
  public int Neg.eval() { return expression.eval() * -1; }
  pointcut NERun(Test t): execution (public void Test.run(..)) &&
                          target(t);
```

```
  void around(Test t) : NERun(t) {
   System.out.println(t.ntree.eval());}
}
```

13.2 Hyper/J Implementation

Feature lp

```
// Exp.java
package lp;
public interface Exp { void print();}

// Lit.java
package lp;
class Lit implements Exp {
   int value;
   Lit (int v) { value = v; }
   public void print() { System.out.print(value); }
}

// Test.java
package lp;
class Test {
   Lit ltree;
   Test() { ltree = new Lit(3); }
   void run() { ltree.print();}
}
```

Feature le

```
// Exp.java
package le;
interface Exp { int eval();}

// Lit.java
package le;
class Lit implements Exp {
   int value;        // stub lp
   Lit (int v) { }   // req constructor
   public int eval() { return value; }
}

// Test.java
package le;
class Test {
   Lit ltree;   // stub lp
   void run() {
      System.out.println(ltree.eval());
   }
}
```

Feature ap

```
// Add.java
package ap;
class Add implements Exp {
  Exp left, right;
  Add (Exp l, Exp r) { left = l; right = r; }
  public void print() {
     left.print();System.out.print("+"); right.print();
  }
}

// Test.java
package ap;
class Test {
   Lit ltree;   // stub lp
   Add atree;
   Test() { atree = new Add(ltree, ltree); }
   void run() { atree.print(); }
}

// Exp.java All interface is stub
package ap;
interface Exp { void print(); }

// Lit.java All class is stub
package ap;
class Lit implements Exp { public void print() { } }
```

Feature ae

```
// Exp.java All interface is stub
package ae;
interface Exp { int eval();}

// Add.java
package ae;
class Add implements Exp {
   Exp left, right;        // stub ap
   Add (Exp l, Exp r) {  } // req constructor
   public int eval() {return left.eval() + right.eval(); }
}

// Test.java
package ae;
class Test {
  Add atree;    // stub ap
  public void run() { System.out.println(atree.eval()); }
}
```

Feature np

```
// Neg.java
package np;
class Neg implements Exp {
   Exp expression;
   Neg (Exp e) { expression = e; }
   public void print() {
    System.out.print("-(");expression.print();System.out.print(")");
   }
}

// Test.java
package np;
class Test {
   Lit ltree;   // stub lp
   Neg ntree;
   Test() { ntree = new Neg(ltree); }
   void run() { ntree.print(); }
}

// Exp.java All interface is stub
package np;
interface Exp { void print();}

// Lit.java All class is stub
package np;
class Lit implements Exp { public void print() { } }
```

Feature ne

```
// Neg.java
package ne;
class Neg implements Exp {
   Exp expression;   // stub np
   Neg (Exp e) { }   // req constructor
   public int eval() {return expression.eval() * -1; }
}

// Test.java
package ne;
class Test {
   Neg ntree;   // stub np
   void run() { System.out.println(ntree.eval()); }
}

// Exp.java All this interface is a stub
package ne;
interface Exp { int eval();}
```

Composition Example LitAdd

```
// LitAdd.hs
hyperspace  LitAdd
  composable class LP.*;
  composable class LE.*;
  composable class AP.*;
  composable class AE.*;

// LitAdd.cm
package LP : Feature.LP
package LE : Feature.LE
package AP : Feature.AP
package AE : Feature.AE

// LitAdd.hm
hypermodule LitAdd
  hyperslices:
   Feature.LP,
   Feature.AP,
   Feature.LE,
   Feature.AE;
  relationships:
   mergeByName;
end hypermodule;
```

13.3 Jiazzi Implementation

Feature lp

```
// Exp.java
package lp;
public interface Exp { void print(); }

// Lit.java
package lp;
public class Lit implements fixed.Exp {
  public int value;
  public Lit (int v) { value = v; }
  public void print() { System.out.print(value); }
}

// Test.java
package lp;
public class Test extends java.lang.Object{
   public fixed.Lit ltree;
   public Test() { ltree = new fixed.Lit(3); }
   public void run() { ltree.print(); }
   public static void main(String[] args) {
     fixed.Test test = new fixed.Test();
     test.run();
   }
}
```

```
// lp.unit
atom lp {
  export lp : lpS;
  import fixed extends lp : lpS;
}

// lpS.sig the signature of the unit
signature lpS = {
  package fixed;
  interface Exp { void print(); }
  public class Lit implements fixed.Exp {
    public int value;
    public Lit (int v);
    public void print();
  }
  public class Test {
    public fixed.Lit ltree;
    public Test();
    public void run();
    static void main(String[] args);
  }
}
```

Feature le

```
// Exp.java
package le;
public interface Exp extends lp.Exp { int eval(); }

// Lit.java
public class Lit extends lp.Lit implements fixed.Exp {
  public Lit(int n) { super(n); }     // problem with constructors
  public int eval() { return value; }
}

// Test.java
package le;
public class Test extends lp.Test{
   public Test() { super(); }
   public void run() {
     super.run();
     System.out.println("= " + ltree.eval());
   }
}

// le.unit
atom le {
  import lp : lpS;
  export le extends lp : leS;
  import fixed extends le;
}
```

```
// leS.sig the signature of unit le
signature leS = l : lpS + {
  package fixed;
  public interface Exp { int eval(); }
  public class Lit {
    public Lit(int n);
    public int eval();
  }
  public class Test { public void run(); }
}
```

Composition of le and lp

```
// lelp.unit
compound lelp {
   export compLELP : leS;
   bind package compLELP to compLELP@fixed;
  }{
  link unit lpInst : lp, leInst : le;
  link package
   leInst@le to *@fixed,
   lpInst@lp to leInst@lp,
   leInst@le to compLELP;
  }
```

13.4 Scala Implementation

Feature lp

```
// lp.scala
package epl;
trait lp {
  type exp <: Exp;
  trait Exp { def print(): unit; }
  class Lit(v: int) extends Exp {
    val value = v;
    def print(): unit = System.out.print(value);
  }
  abstract class Test {
    val ltree: exp;
    def run(): unit = { ltree.print(); }
  }
}
```

Feature le

```
// le.scala
package epl;
trait le extends lp {
  type exp <: Exp;
  trait Exp extends super.Exp { def eval(): int; }
  class Lit(v: int) extends super.Lit(v) with Exp {
    def eval(): int = value;
  }
```

```
  abstract class Test extends super.Test {
    override def run(): unit = {
      super.run();
      System.out.println(ltree.eval());
    }
  }
}
```

Feature ap

```
// ap.scala
package epl;
trait ap extends lp {
  class Add(l: exp, r: exp) extends super.Exp {
    val left = l; val right = r;
    def print(): unit = {
      left.print(); System.out.print("+"); right.print();
    }
  }
  abstract class Test extends super.Test {
    val atree: exp;
    override def run(): unit = { super.run(); atree.print(); }
  }
}
```

Feature ae

```
// ae.scala
package epl;
trait ae extends ap with le {
  class Add(l: exp, r: exp) extends super.Add(l, r) with Exp {
    def eval(): int = left.eval() + right.eval();
  }
  abstract class Test extends super.Test {
    override def run(): unit = {
      super.run(); System.out.println(atree.eval());
    }
  }
}
```

Feature np

```
// np.scala
package epl;
trait np extends lp {
  class Neg(o: exp) extends super.Exp {
    val od = o;
    def print(): unit = { System.out.print("-"); od.print(); }
  }
  abstract class Test extends super.Test {
    val ntree: exp;
    override def run(): unit = {
      super.run();
      System.out.print("("); ntree.print(); System.out.print(")");
    }
```

```
  }
}
```

Feature ne

```
// ne.scala
package epl;
trait ne extends np with le {
  class Neg(o: exp) extends super.Neg(o) with Exp {
    def eval(): int = - od.eval();
  }
  abstract class Test extends super.Test {
    override def run(): unit = {
      super.run();
      System.out.print(ntree.eval());
    }
  }
}
```

Composition Example LitAdd

```
// LitAdd.scala
package epl;
abstract class Test1 extends lp with ap {
  abstract class Test extends super.Test with super[ap].Test;
}

abstract class Test2 extends Test1 with le {
  abstract class Test extends super.Test with super[le].Test;
}

abstract class Test3 extends Test2 with ae {
  abstract class Test extends super.Test with super[ae].Test;
}

object LitAdd extends Test3 {
  type exp = Exp;
  object test extends super.Test {
    val ltree = new Lit(3);
    val atree = new Add(ltree, new Lit(7));
  }

def main(args: Array[String]) : unit = { test.run(); }

}
```

13.5 AHEAD Implementation

Feature lp

```
// Exp.jak
public interface Exp { void print(); }

// Lit.jak
class Lit implements Exp {
   int value;
```

```
   Lit (int v) { value = v; }
   void print() { System.out.print(value); }
}

// Test.jak
class Test {
   Lit ltree;
   Test() { ltree = new Lit(3); }
   public void run() { three.print(); }
}
```

Feature le

```
// Exp.jak
refines interface Exp { int eval(); }

// Lit.jak
refines class Lit implements Exp {
   public int eval() { return value; }
}

// Test.jak
refines class Test {
   public void run() {
     Super().run();
     System.out.println(ltree.eval());
   }
}
```

Feature ap

```
// Add.jak
class Add implements Exp {
   public Exp left, right;
   Add (Exp l, Exp r) { left = l; right = r; }
   public void print() {
     left.print(); System.out.print("+"); right.print();
   }
}

// Test.jak
refines class Test {
   public Add atree;
   refines Test() { atree = new Add(ltree, ltree); }
   public void run() { Super.run(); atree.print(); }
}
```

Feature ae

```
// Add.jak
refines class Add implements Exp {
   public int eval() { return left.eval() + right.eval(); }
}

// Test.jak
refines class Test {
```

```
   public void run() {
     Super.run(); System.out.println(atree.eval());
   }
}
```

Feature np

```
// Neg.jak
class Neg implements Exp {
   Exp expression;
   Neg (Exp e) { expression = e; }
   void print() {
     System.out.print("-("); expression.print();
     System.out.print(")");
   }
}

// Test.jak
refines class Test {
   Neg ntree;
   refines Test() { ntree = new Neg(ltree); }
   public void run() { Super.run(); ntree.print(); }
}
```

Feature ne

```
// Neg.jak
refines class Neg implements Exp {
   int eval() { return expression.eval() * -1; }
}

// Test.jak
refines class Test {
   void run() {
     Super.run();
     System.out.println(ntree.eval());
   }
}
```