

A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite

Don Batory

Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
batory@cs.utexas.edu

Abstract. *Feature oriented programming (FOP)* is the study of feature modularity and its use in program synthesis. AHEAD is a theory of FOP that is based on a fundamental concept of generative programming that functions map programs. This enables the design of programs to be expressed compositionally as algebraic expressions, which are suited for automated analysis, manipulation, and program synthesis. This paper is a tutorial on FOP and AHEAD. We review AHEAD's theory and the tool set that implements it.

1 Introduction

Software engineering (SE) is in a perpetual crisis. Software products are increasing in complexity, the cost to develop and maintain systems is skyrocketing, and our ability to understand systems is decreasing. A basic goal of SE is to successfully manage and control *complexity*; the “crisis” indicates that SE technologies are failing to achieve this goal. There are many culprits. One surely is that today's software design and implementation techniques are simply too low-level, exposing far more detail than is necessary to make a program's design, construction, and ease of modification simple. Future software design technologies will need to do better, and it should not be surprising that they will be different from those of today.

Looking to the future, SE paradigms will likely embrace:

- *generative programming (GP)*
- *domain-specific languages (DSLs)*
- *automatic programming (AP)*

GP is about automating software development. Eliminating the task of writing mundane and rote programs is a motherhood to improved programmer productivity and program quality. Program synthesizers will transform input specifications into target programs. These specifications will not be written in Java or C# — which are too low-level — but rather in high-level notations called DSLs that are specific to a particular domain. DSL programs are known to be both easier to write and maintain than their low-level (e.g., Java) counterparts. Ideally, DSLs are declarative, allowing their users to define *what* is needed and leave it up to the DSL compiler to produce an efficient program automatically that does the *how* part. But placing the burden of program synthesis on a DSL compiler should not be taken lightly. This involves the

problem of AP; it is a technical problem of great difficulty, as little progress has been made in the last 25 years to produce demonstrably efficient programs from declarative specs. Advancement on all three fronts (GP, DSLs, and AP) are needed before the crisis in SE will noticeably diminish.

While it is wishful thinking that simultaneous advances on all three fronts is possible, it is worth noting that a spectacular example of this futuristic SE paradigm was realized over *25 years ago* — ironically around the time when most people were giving up on AP. Furthermore, this work had a fundamental impact on commercial applications. The example is relational query optimization. SQL is a prototypical DSL: it is a *declarative language* for retrieving data from tables. An SQL compiler translates an SQL statement into a relational algebra expression. A query optimizer accomplishes the goal of *automatic programming* by applying algebraic identities to automatically rewrite — and hence optimize — relational algebra expressions. The task of translating an optimized expression into an efficient program is an example of *generative programming*.

Relational optimizers revolutionized databases: data retrieval programs that were hard to write, hard to optimize, and hard to maintain are now produced automatically. There is nothing special about data retrieval programs: all interesting programs are hard to write, optimize, and maintain. Thus if ever there was a “grand challenge” for SE, it would be to replicate the success of relational query optimization in other domains.

AHEAD is a theory of *feature oriented programming (FOP)* that shows how the concepts and framework of relational query optimization generalize to other domains. ATS is a suite of tools that implement the AHEAD theory.

1.1 Background

How do you describe a program that you’ve written to a prospective customer? You are unlikely to recite what packages you’re using — because the customer would unlikely have any interest in such details. Instead, you would take a more promising approach of explaining the *features* — increments in program functionality — that your program offers its clients. This works because clients know their requirements and can see how features satisfy requirements.

Programs come in different flavors, e.g., entry-level through deluxe. The differences between these categories are the presence or absence of features (or more commonly, sets of features). Entry-level versions have a minimal feature set; deluxe advertises the most.

But if we describe programs by features or differentiate programs by features, why can’t we build programs (or program families) from feature specifications? In fact, we can. This is the area of research called *product-lines*. The ability to add and remove features suggests that features can be modularized. While it is possible to construct product-lines without modularizing features (e.g., through the extensive use of `#if-#endif` preprocessor declarations), we focus on a particular sub-topic of product-line research that deals with feature modularization. By making features first-class design and implementation entities, it is easier to add and remove features from applications. (In fact, this is a capability that most of us wish we had today — the ability to add and remove features from our programs. We don’t have it now; the purpose of this paper is to explain how it can be done in a general way). It happens that feature modularity

goes far beyond conventional notions of code modularity. This, among other things, makes it a very interesting topic.

Feature oriented programming (FOP) is the study of feature modularity and programming models that support feature modularity. A powerful form of FOP is based on a methodology called *step-wise development (SWD)*. SWD is both simple and ancient: it advocates that complex programs can be constructed from simple programs by incrementally adding details. When incremental units of change are features, FOP and SWD converge. This is the starting point of AHEAD and ATS. But what is a feature? How is it represented? And how are features and their compositions modeled?

1.2 A Clue

Consider any Java class `c`. A class member could be a data field or a method. Class `c` below has four members `m1`—`m4`.

```
class C {
    member m1;
    member m2;
    member m3;
    member m4;
}
(1)
```

Have you ever noticed that there is no unique definition for `c`? The members of `c` could be defined in a single class as above, or distributed over an inheritance hierarchy of arbitrary height. One possibility is to have class `c1` encapsulate member `m1` and `c23` encapsulate members `m2` and `m3`:

```
class C1 { member m1; }
class C23 extends C1 {
    member m2;
    member m3;
}
class C4 extends C23 { member m4; }
class C extends C4 {}
(2)
```

From a programmatic viewpoint, both definitions of `c`, namely (1) and (2), are indistinguishable. In fact, we could further decompose `c23` to be:

```
class C2 extends C1 { member m2; }
class C3 extends C2 { member m3; }
class C23 extends C3 {}
```

and the definition of `c` would not change; it would still have members `m1`—`m4`. Moreover, there's nothing really special about the placement of member `m1` (or `m2` ...) in this hierarchy. If method `m1` references other members, as long as these members are not defined lower in the inheritance chain than `m1`, `m1` can appear in any class of that chain.

If you recall your high school or college courses on algebra, you may recognize these ideas. Consider sets and the union operation. We can define the sets:

```
C1 = { m1 }
C2 = { m2 }
C3 = { m3 }
C4 = { m4 }
C23 = C2 ∪ C3
C = C1 ∪ C23 ∪ C4 = C1 ∪ C2 ∪ C3 ∪ C4
```

Union is commutative, which means that the order in which the union of sets is taken doesn't matter. This is similar to, but not the same as, inheritance because as we saw, a method can be added only as long as members it references are not defined in subclasses.

Something a bit closer to inheritance are vectors and the vector operations of addition(+) and movement(\rightarrow). Suppose we define vectors in 4-space:

```
C1 = (m1, 0, 0, 0)
C2 = (0, m2, 0, 0)
C3 = (0, 0, m3, 0)
C4 = (0, 0, 0, m4)
```

You know about vector addition; vector movement is the path that is followed when laying vectors end-to-end. Vector addition is commutative; vector movement is not:

```
C = (m1, m2, m3, m4) = C1 + C2 + C3 + C4
C1 + C2 + C3 + C4 = C4 + C3 + C2 + C1
C1  $\rightarrow$  C2  $\rightarrow$  C3  $\rightarrow$  C4  $\neq$  C4  $\rightarrow$  C3  $\rightarrow$  C2  $\rightarrow$  C1
```

Inheritance has the flavor of both vector arithmetic and vector movement.

When you think about an *operation* for inheritance, what you are really defining is an operation for class extension. A *class extension* can add new members and extend existing methods of a class. Here's an example. Suppose a program **P** has a single class **B** that initially contains a single data member **x**:

```
class B { int x; } // program P
```

Suppose an extension **R** of program **P** adds data member **y** and method **z** to class **B**. Let us write this extension as:

```
refines class B { // extension R
  int y;
  void z() {...}
} (3)
```

where “**refines**” is a keyword modifier to mean extension. The composition of **R** with **P** defines a new program **N** with a single class **B** with three members:

```
class B { // program N
  int x;
  int y;
  void z() {...}
} (4)
```

In effect, this composition is expressed by the following inheritance chain, called an *extension chain*:

```
class BP { int x; }
class BR extends BP {
  int y;
  void z() {...}
}
class BN extends BR {} (5)
```

where subscripts indicate the program or extension from which that fragment of **B** is defined.

We can express these ideas algebraically by “values” and “functions”. Program **P** is a *value* (or constant function) — it defines a base artifact. An extension is a *function*

that maps programs, so \mathbf{R} is a function. A composition is an expression. We can model (5) as the equation $\mathbf{N} = \mathbf{R}(\mathbf{P})$ or $\mathbf{N} = \mathbf{R} \bullet \mathbf{P}$, where \bullet denotes function composition.

We can express our previous example about class \mathbf{c} in this manner. Here is one way: let $\mathbf{c1}$ be a value and $\mathbf{c2}$, $\mathbf{c3}$, and $\mathbf{c4}$ be the extensions:

```
class C { member m1; } // value C1
refines class C { member m2; } // function C2
refines class C { member m3; } // function C3
refines class C { member m4; } // function C4
```

Now, class \mathbf{c} of can be synthesized by *evaluating* the expression $\mathbf{c4} \bullet \mathbf{c3} \bullet \mathbf{c2} \bullet \mathbf{c1}$. The expression — $\mathbf{c4} \bullet \mathbf{c3} \bullet \mathbf{c2} \bullet \mathbf{c1}$ — is called the *design* of \mathbf{c} . Taking this idea further, we see that $\mathbf{c23}$ has a representation as a composite function or composite extension:

$$\mathbf{c23} = \mathbf{c2} \bullet \mathbf{c3}$$

which represents the code:

```
refines class C {
  member m2;
  member m3;
}
```

There are loose ends to tie up before a bigger picture emerges. First, there's scalability. The effects of a program extension need not be limited to a single Java class. In fact, it is common for a "large-scale" extension to encapsulate multiple class extensions as well as new classes. That is, such an extension would augment existing classes of a program with new members and extend existing methods, but would also introduce new classes that could be subsequently augmented.

Second, program extensions have meaning when they encapsulate the implementation of a feature. Have you ever added a new feature to an existing program? You discover that you often have to extend a number of classes, as well as add new classes to a program. Well, a feature is a "large scale" program extension.

Third, in product-line design, features are stereo-typical units of application design that can be composed with other features to produce customized programs. A model of a product-line — called a *domain model* — is a set of values and functions each representing a particular feature, that can be composed to synthesize customized programs.

Fourth, recall that a key to the success of relational query optimizers is that they use expressions to represent program designs. That is, a data retrieval program is defined by a composition of relational algebra operations. To see the generalization, a domain model is an *algebra* — a set of operations ("values" and "functions") whose compositions define the space of programs that can be synthesized. Given an algebra, there will always be algebraic identities among operations. These identities can be used to optimize algebraic expression definitions of programs, just like relational algebra expressions can be optimized. (Some domains will have more interesting optimizations than others).

Fifth, what is *design*? If you think about it, this is a *really* hard question to answer, because it is asking for a clear articulation of a deeply intuitive idea. Our discussions offer a simple answer: a program is a value. The design of a program is the expression that produces its value. If multiple expressions produce the same value, then these expressions represent equivalent designs of that program.

Now, let's consider a more precise way to express these ideas.

2 A Model of FOP

Salient ideas of FOP as expressed by two models: GenVoca and its successor AHEAD.

2.1 GenVoca

GenVoca is a design methodology for creating application families and architecturally-extensible software, i.e., software that is customizable via feature addition and removal. It follows traditional step-wise development with one major difference: instead of composing thousands of microscopic program extensions (e.g., $x+1 \Rightarrow \text{inc}(x)$) to yield admittedly small programs, GenVoca scales extensions so that each adds a feature to a program, and composing few extensions yields an entire program.

In GenVoca, programs are *values* and program extensions are *functions*. Consider the following values that represent base programs with different features:

```
f           // program with feature f
g           // program with feature g
```

A program extension is a function that takes a program as input and produces a feature-augmented program as output:

```
i•x         // adds feature i to program x
j•x         // adds feature j to program x
```

A multi-featured application is an *equation* that is a named expression. Different equations define a family of applications, such as:

```
app1 = i•f   // app1 has features i and f
app2 = j•g   // app2 has features j and g
app3 = i•j•f // app3 has features i, j, f
```

Thus, the features of an application can be determined by inspecting its equation.

Note that a function represents both a feature *and* its implementation — there can be different functions with different implementations of the *same* feature:

```
k1•x       // adds k with implementation1 to x
k2•x       // adds k with implementation2 to x
```

When an application requires the use of feature k , it is a problem of *expression optimization* to determine which implementation of k is best (e.g., provides the best performance)¹. It is possible to automatically design software (i.e., produce an expression that optimizes some criteria) given a set of constraints for a target application. This is automatic programming.

¹ Different equations represent different programs and equation optimization is over the space of semantically equivalent programs. This is identical to relational query optimization: a query is represented by a relational algebra expression, and this expression is optimized. Each expression represents a different, but semantically equivalent, query-evaluation program.

Although GenVoca values and functions seem untyped, constraints do exist. *Design rules* are domain-specific constraints that capture syntactic and semantic constraints that govern legal compositions. It is common that the selection of a feature will disable or enable the selection of other features. More on design rules later.

2.2 AHEAD

AHEAD, or *Algebraic Hierarchical Equations for Application Design*, embodies four key generalizations of GenVoca. First, a program has many representations besides source code, including UML documents, makefiles, BNF grammars, documents, performance models, etc. A model of FOP must deal with all these representations.

Second, each representation is written in its own language or DSL. The code representation of a program may be represented in Java, a machine executable representation may be bytecodes, a makefile representation could be an `ant` XML file, a performance model may be a set of Mathematica equations, and so on. An FOP model must support an open-ended spectrum of languages to express arbitrary program representations.

Third, when a feature is added to a program, any or all of the program's representations may be updated. That is, the source code of a program changes (to implement the feature), makefiles change (to build the feature), Mathematica equations change (to profile the feature), etc. Thus, the concept of extension applies not only to source code representations, but other representations as well.

Fourth, FOP models must deal with a general notion of modularity: *a module is a containment hierarchy of related artifacts*. A class is a module (1-level hierarchy) that contains a set of data members and methods. A package or JAR file is a module (2-level hierarchy) that contains a set of classes. A J2EE EAR file is a module (3-level hierarchy) that contains a set of packages, HTML files, and descriptor files. Going further, a client-server program is also a module (a multi-level hierarchy) that contains representations of both client and server programs.

Given the above, a generalization of GenVoca emerges. A “value” is a module that defines a containment hierarchy of related artifacts of different types written in potentially different languages. An “extension” is a function that maps containment hierarchies. Thus, whenever an extension is applied to a program (i.e., an AHEAD value), any or all of the representations in this module (containment hierarchy) will be updated and new artifacts added. Thus, as AHEAD extensions are applied, all of the representations of the resulting program remain consistent. This is exactly what we need.

The notations of AHEAD extend those of GenVoca. A model \mathbf{m} is a set of features that are “values” or “functions” called *units*:

$$\mathbf{m} = \{ \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \dots \}$$

Individual units may themselves be sets, recursively:

$$\begin{aligned} \mathbf{a} &= \{ \mathbf{x}, \mathbf{y}, \mathbf{z} \} \\ \mathbf{z} &= \{ \mathbf{r}, \mathbf{q} \} \\ &\dots \end{aligned}$$

The nesting of sets models a containment hierarchy or module. The composition of units is defined by the *Law of Composition*. That is, given units \mathbf{x} and \mathbf{r} :

```

X = { ax, bx, cx }
Y = { ay, cy, dy }

```

The composition of **Y** and **X**, denoted **Y•X**, is formed by “aligning” the units of **x** and **y** that have the same name (ignoring subscripts) and composing:

```

Y•X = { ay•ax, bx, cy•cx, dy } // Law of Composition (6)

```

That is, artifact **a** of **Y•X** is the original artifact **a_x** composed with the extension **a_y**; artifact **b** of **Y•X** remains unchanged from its original definition **b_x**, etc. Composition is recursive: if units represent sets, their compositions are expanded according to (6).

To see the connection with inheritance, consider the following inheritance hierarchy which is a class representation of (6). Assume **a** and **c** are methods, where **a_y** and **c_y** extend (or override) their super-methods **a_x** and **c_x**:

```

class X {
    member ax;
    member bx;
    member cx;
}

class Y extends X {
    member ay;
    member cy;
    member dy;
}

class Y•X extends Y {}

```

How the composition operator **•** is defined depends on the artifact type. **•** is polymorphic: it can be applied to all artifacts (i.e., all artifacts can be composed/extended) but what composition/extension means is artifact type dependent (i.e., how makefile artifacts are extended will be analogous to but not the same as how code artifacts are extended). This means that different tools implement **•** for code and makefiles.

AHEAD representations lead to simple tools and implementations. While there are many ways in which containment hierarchies can be realized, the simplest way is to map containment hierarchies to file system directories. Thus a feature might encapsulate many Java files, class files, HTML files, etc. Feature composition corresponds to directory composition.

Recognize what AHEAD represents: it is a *structural theory of information* — it is not just a theory of code synthesis. Its premise is that if a program can be understood in terms of features, so too can all of its representations — code and otherwise. We can choose to interpret individual terms of AHEAD expressions as code files or code directories, but we are free to consider other representations as well. A familiarity with relational query optimization bares this out: the optimizer reasons about a program in terms of performance representations of relational operations (i.e., cost functions), while the code generator produces a program from code representations of these same operations. Reasoning about programs often relies on different representations of programs. AHEAD provides a mathematical foundation for expressing their inter-relationships.

We’ll explore examples of these ideas in the following sections.

3 A Simple Example

Consider a family of elementary post-fix calculators that are modeled after Hewlett-Packard calculators. Calculators in this family are differentiated on (a) the arithmetic values **BigInteger** (an unlimited precision integer) or **BigDecimal** (an unlimited precision, signed decimal number) that can be specified and (b) the set operations that can be performed on them, which includes addition, division, and subtraction.

An AHEAD model that describes this family is **c**:

C = { Base, BigI, BigD, Iadd, Idiv, Isub, Dadd, Ddivd, Ddivu, Dsub }

The lone value in this model is **Base**, which defines an empty **calc** (short for “calculator”) class (Figure 1a). The extensions **BigI** and **BigD** introduce a 3-level stack of **BigInteger** or **BigDecimal** objects, respectively (Figure 1b-c). **BigI** and **BigD** are mutually exclusive as the stack variables introduced by both have the same name, but are of different types. Thus, calculators either work on **BigInteger** or **BigDecimal** numbers, but not both.

```
class calc { }
```

(a) Base/calc.jak

```
refines class calc {
  void divide() {
    e0 = e0.divide( e1 );
    e1 = e2;
  }
}
```

(d) Idiv/calc.jak

```
refines class calc {
  void add() {
    e0 = e0.add(e1);
    e1 = e2;
  }
}
```

(e) Iadd/calc.jak
and Dadd/calc.jak

```
import java.math.BigDecimal;

refines class calc {
  void divide() {
    e0 = e0.divide( e1,
      BigDecimal.ROUND_DOWN );
    e1 = e2;
  }
}
```

(f) Ddivd/calc.jak

```
import java.math.BigInteger;

refines class calc {
  static BigInteger zero = BigInteger.ZERO;
  BigInteger e0 = zero, e1 = zero, e2 = zero;

  void enter( String val ) {
    e2 = e1;
    e1 = e0;
    e0 = new BigInteger(val);
  }

  void clear() {
    e0 = e1 = e2 = zero;
  }

  String top() { return e0.toString(); }
}
```

(b) BigI/calc.jak

```
import java.math.BigDecimal;

refines class calc {
  static BigDecimal
    zero = new BigDecimal("0");
  BigDecimal e0 = zero, e1 = zero,
    e2 = zero;

  void enter( String val ) {
    e2 = e1;
    e1 = e0;
    e0 = new BigDecimal(val);
  }

  void clear() {
    e0 = e1 = e2 = zero;
  }

  String top() { return e0.toString(); }
}
```

(c) BigD/calc.jak

Fig. 1. Files from the **C** model

The extensions `iadd`, `idiv`, and `isub` respectively introduce the `BigInteger` addition, division, and subtraction methods to the `calc` class (Figure 1d-e). The extensions `Dadd`, `Ddivd`, `Ddivu`, and `Dsub` do the same for `BigDecimal` methods (Figure 1f-g). Note that there are two mutually exclusive `BigDecimal` division extensions: `Ddivd` and `Ddivu`. `Ddivd` rounds answers down, `Ddivu` rounds up.

As you may have already noticed, these files look like Java programs, but the language that we are using is not Java but an extended Java language called *Jak* (short for “Jakarta”). *Jak* files have `.jak` extensions, like Java files have `.java` extensions.

A calculator is defined by an equation. Here are a few calculator definitions:

```
i1 = Iadd•BigI•Base
i2 = Isub•Iadd•BigI•Base
i3 = Idiv•Iadd•BigI•Base

d1 = Dadd•BigD•Base
d2 = Dsub•Dadd•BigD•Base
d3 = Ddivd•Dadd•BigD•Base
```

Calculator `i1` offers `BigInteger` addition. `i2` also supports subtraction. `i3` has `BigInteger` addition and division. `d1`—`d3` are the corresponding calculators for `BigDecimal` numbers using rounded-down division. The code generated for the `i3 calc` class is shown in Figure 2. Note the term “*layer*” in Figure 2 is used interchangeably with “*feature*” in AHEAD.

```
layer i3;

import java.math.BigInteger;

class calc {
    static BigInteger zero = BigInteger.ZERO;
    BigInteger e0 = zero, e1 = zero, e2 = zero;

    void add() {
        e0 = e0.add(e1);
        e1 = e2;
    }

    void clear() {
        e0 = e1 = e2 = zero;
    }

    void divide() {
        e0 = e0.divide( e1 );
        e1 = e2;
    }

    void enter( String val ) {
        e2 = e1;
        e1 = e0;
        e0 = new BigInteger(val);
    }

    String top() { return e0.toString(); }
}
```

Fig. 2. `i3/calc.jak`

Model Exercises

- [1] What other calculator features could be added to `c`? What would be their Jak definitions? Look at the `BigInteger` and `BigDecimal` pages in the J2SDK documentation for possibilities.
- [2] Suppose the size of the stack was variable. How would this be expressed as an extension? What modifications of existing extensions would be needed?
- [3] Modify model `c` so that `BigDecimal` round-up and round-down are features, which could parameterize operations like division.
- [4] How would `c` be modified to permit the synthesis of a program that would invoke the calculator from the command-line? From a GUI?

Tool Exercises

An AHEAD model `c` corresponds to a directory `c`, and each unit `u` in `c` corresponds to a subdirectory of `c`, namely `c/u`. The contents of a unit in our example is merely a `calc.jak` file. The AHEAD directory structure of `c` is:

```
C/Base/calc.jak           // see Figure 1a
C/BigI/calc.jak          // see Figure 1b
C/BigD/calc.jak          // see Figure 1c
C/Iadd/calc.jak          // see Figure 1d
C/Idiv/calc.jak          // see Figure 1e
C/Isub/calc.jak
C/Dadd/calc.jak          // see Figure 1d
C/Ddiv/calc.jak          // see Figure 1f
C/Ddivu/calc.jak
C/Dsub/calc.jak
```

Although we provide no `calc.jak` files for `Isub` and `Dsub`, they are easy to write. In fact, they are almost identical to the `calc.jak` files for `Iadd` and `Dadd`.²

The `composer` tool is used to evaluate equations and has many optional parameters. For our tutorial, we need to reset one of these parameters. Create in the model directory a file called `composer.properties`. Its contents is a single line (which says when composing Jak files, use the `jumpack` tool):

```
unit.file.jak : JamPackFileUnit
```

To evaluate an equation, run `composer` in the model directory. The order in which model units are listed on the `composer` command line are inside-to-outside order, and the name of the composition is given by the `target` option. Thus, to evaluate `i3` use:

```
> cd c
> composer --target=i3 Base BigI Idiv Iadd
```

The result of the composition is the directory `c/i3`, which contains a single file, `calc.jak`, shown in Figure 2. Note that the order in which units are listed on the `composer` command line is in *reverse* order in which they are listed in an equation — base first, outermost extension last. (This is a legacy oddity of AHEAD tools that has never been changed. Sigh.)

- [5] Validate your Model Exercise solutions by implementing them using AHEAD tools.

² So why not just define one layer to represent both? This could be done with our current tools, as they are preprocessors. In future tools, these files will be different, because the types of variables for `e1`—`e3` will need to be explicitly declared. When this occurs, the corresponding files will indeed be different.

3.1 Translating to Java

The `jak2java` tool converts Jak files to their Java counterparts:

```
> cd i3
> jak2java *.jak
```

The above command-line translates all Jak files (in our case, there is only one file — `calc.jak`) to their Java equivalents. Of course, these generated files can be compiled in the usual way:

```
> javac *.java
```

Note there are Jak files (i.e., those that refine classes and interfaces) that cannot be translated to Java, as they have no Java counterpart. `jak2java` translates only Jak classes and interfaces.

3.2 Design Rules

New arithmetic operations could be added to `c` to enlarge the family of calculators. At the same time, it becomes increasingly clear that not all compositions are meaningful. In fact, it is quite easy to deliberately or unintentionally specify meaningless compositions, but `composer` is usually quite happy to produce code for them. We need automated help to detect illegal compositions.

This is not a problem specific to calculators, but rather a fundamental problem in FOP. The use of a feature in a program can enable or disable other features. *Design rules* are domain-specific constraints that define composition correctness predicates for features. *Design rule checking (DRC)* is the process by which design rules are composed and their predicates validated. AHEAD offers two different tools for defining and evaluating design rules: `drc` and `guids1`. `drc` is a first-generation tool ; `guids1` is a next-generation tool that we will highlight here.

The theory behind both tools is the use of grammars to define legal sequences (i.e., compositions) of features. A grammar for model `c` is:

```
C      : Type Base ;
Type  : BigInt+ BigI
         | BigDec+ BigD ;
BigInt : Iadd | Idiv | Isub ;
BigDec : Dadd | Ddivu | Ddivd | Dsub ;
```

(7)

where tokens are units of `c`. A sentence of this grammar specifies a particular sequence or composition of features. The set of all sentences defines the model's *product-line*, i.e., the set of all possible expressions or compositions of features.

Like any grammar, some sentences are semantically invalid. To weed out incorrect sentences, a grammar is augmented with *attributes*. Conditions for correct sentences (or correct compositions) are predicates defined over these attributes. That is, these predicates filter out syntactically incorrect sentences. The core theory behind both tools are *attribute grammars*, a well-understood technology.

In the case of our `c` model, syntactic correctness is almost all that is needed. The only additional constraint — which is simple enough to have been expressed by an additional grammar rule — is the mutually exclusive nature of `Ddivu` and `Ddivd`; at most one of these features can appear in a decimal calculator.

As an aside, product-line researchers are familiar with feature diagrams, i.e., trees whose terminal nodes are primitive features and non-terminal nodes are compound features. So what is the connection between grammars and feature diagrams? Although feature diagrams were introduced by Kang, et al in the early 1990s and “GenVoca” grammars, like (7), were introduced in 1992, it was not until 2002 that de Jonge and Visser noticed that feature diagrams are graphical representations of grammars. In fact, grammars provide an added benefit beyond feature diagrams in that they tell us the *order* in which features are composed, which is important to AHEAD and step-wise development. So if you’re a fan of feature diagrams, you will see that the tools and ideas we present here are directly applicable to your interests.

3.2.1 The `guids1` Tool

`guids1` is a next generation tool for design rule checking. The key idea is that a tree grammar (i.e., a grammar where each token appears at most once in a sentence and which itself can be depicted as a tree) can be represented as a propositional formula. Moreover, *any* propositional constraints on the use of features can be added to this formula. Amazingly, an FOP domain model reduces to a single propositional formula, whose variables correspond to primitive and compound features!

Here’s why this is important. First, we have a compact representation of an FOP domain model: it is a grammar (which encodes syntactic/ordering constraints) plus a set of propositional formulas that constrains sentences to legal compositions. The entire `guids1` specification for the `c` model is shown in Figure 3. The `:: Name` phrase in a `guids1` specification is a way to assign a name to a pattern.

```

C      : Type Base :: Main ;
Type  : BigInt+ BigI :: BigInteger
      | BigDec+ BigD :: BigDecimal ;
BigInt : Iadd | Idiv | Isub ;
BigDec : Dadd | Ddivu | Ddivd | Dsub ;
%% // arbitrary propositional formulas below
Ddivu or Ddivd implies not (Ddivu and Ddivd);

```

Fig. 3. `C.m` -- the `guids1` Model of C

Second, one of the hallmarks of feature oriented designs is the ability to declaratively specify programs in terms of the features that it offers. `guids1` takes a model specification (a `.m` file) and synthesizes a Java GUI. As a user selects features in the GUI, `guids1` uses a logic truth maintenance system to propagate constraints so that users cannot specify incorrect programs. (`guids1` is, in effect, a syntax-directed editor that guarantees compilable programs). Further, because a domain model is a propositional formula, satisfiability solvers (or SAT solvers) can be used to help debug models. (A *SAT solver* is a tool that determines if there is a truth assignment to boolean variables that will satisfy a propositional formula). We believe that SAT solvers will be invaluable assets in future FOP tools.

To generate a declarative language for our calculator, run `guids1` on the `c` model file:

```
> guids1 c.m
```

The GUI that is synthesized is shown in Figure 4.

Model Exercises

- [6] How would you change the `guids1` file if both Add and Subtract operations were always included if either is selected? Similarly for Divide and Multiply?

Tool Exercises

- [7] Implement your solution to [6].
- [8] To see an explanation (in the form of a proof) why certain features have been automatically selected or deselected, run `guids1`, go to **Help**, and select “**Display reason for variable selection**”. Now drag your cursor over a variable that has been greyed out (i.e., whose value was automatically selected). In the text area at the bottom of the selection panel, you’ll see the explanation/proof for that variable’s value.
- [9] Alter the `c.m` file of Figure 3 by eliminating the propositional constraint and modifying the grammar specification to account for the mutual exclusion of `ddivd` and `ddivu`. Test your solution to see the impact of this change. (Hint: your GUI front-end will change).

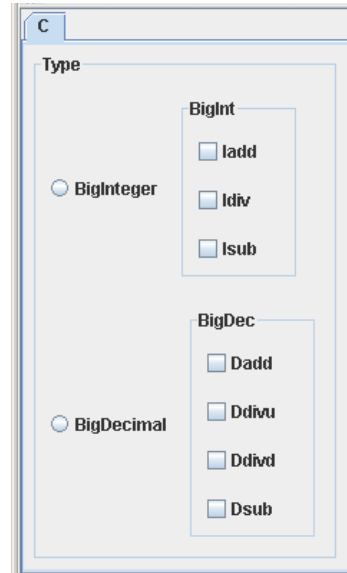


Fig. 4. Declarative GUI for Model C

4 Other ATS Tools and Program Representations

So far, you have seen the `composer`, `jampack` (which is called by `composer`), `jak2java`, and `guids1` tools. Now let’s look at the `mixin`, `unmixin`, and `reform` tools.

4.1 `mixin`

`mixin` is another tool, besides `jampack`, that can compose Jak files. Edit the `unit.file.jak` line in the `composer.properties` file to be:

```
unit.file.jak : MixinFileUnit
```

This is the default setting for `unit.file.jak`. If `composer` doesn’t see a `composer.properties` file, it uses `mixin` to compose Jak files.

Let’s re-evaluate the `i3` equation to see how `mixin` works:

```
> cd C
> composer --target=i3 Base BigI Idiv Iadd
```

This is the same command as before. However, the `calc.jak` file that is produced is quite different and is shown in Figure 5.

```

layer i3;

import java.math.BigInteger;

SoURce Root base "../base/calc.jak";
abstract class calc$$base {}

SoURce BigI "../BigI/calc.jak";
abstract class calc$$BigI extends calc$$base {
    static BigInteger zero = BigInteger.ZERO;
    BigInteger e0 = zero, e1 = zero, e2 = zero;

    void add() {
        e0 = e0.add( e1 );
        e1 = e2;
    }

    void clear() {
        e0 = e1 = e2 = zero;
    }
    void divide() {
        e0 = e0.divide( e1 );
        e1 = e2;
    }

    void enter( String val ) {
        e2 = e1;
        e1 = e0;
        e0 = new BigInteger( val );
    }

    String top() {
        return e0.toString();
    }
}

SoURce Iadd "../Iadd/calc.jak";
abstract class calc$$Iadd extends calc$$BigI {
    // adds BigIntegers
    void add() {
        // adds BigIntegers
        e0 = e0.add( e1 );
        e1 = e2;
    }
}

SoURce Idiv "../Idiv/calc.jak";
class calc extends calc$$Iadd {
    void divide() {
        e0 = e0.divide( e1 );
        e1 = e2;
    }
}

```

Fig. 5. mixin-produced.jak file

The idea behind **mixin** is simple: each extension is mapped to a class in an extension (inheritance) chain. Each class is prefaced by a **source** statement which indicates the name of the feature and the actual file from which the class was derived. Thus, in Figure 5 four Jak files were composed to yield the **calc** class; this class is the terminal class of a four-class extension chain. All other classes are abstract — meaning that they can't be instantiated and whose purpose is only to contribute

members to the final class in the chain. Note that class names are *mangled* (i.e., by appending `$$<featureName>`) to make them unique.

The intent of `mixin` and `jampack` is that you can use either tool to compose Jak files. As you'd expect, the programs of Figure 2 and Figure 5 are functionally equivalent.

Both `mixin` and `jampack` can compose files that they themselves have produced. That is, a `jampack`-produced Jak file can be composed with another `jampack`-produced Jak file. The same holds for `mixin`. Because `jampack`-produced Jak files have the same format as uncomposed Jak files, `mixin` can compose files produced by `jampack`. However, the reverse is not true: `jampack` cannot compose `mixin`-produced files.

4.2 unmixin

So why use `mixin`? Why not always use `jampack`? Consider a typical debugging cycle: you compose files, use `jak2java` to translate Jak files to Java files, compile and run the Java files to discover bugs. The *composed* Jak files are patched and the cycle continues. Eventually, you'll want to back-propagate the changes you made to the composed files to their original feature definitions. Knowing what feature files to update won't always be easy — and the problem becomes worse as the number and size of the Jak files increases. Back-propagation is a tedious and error-prone process.

Because `mixin` preserves feature boundaries, it is easy to know what features to update. In fact, with `source` statements, the propagation of changes can be done automatically. That's the purpose of `unmixin`. The idea is that you compose a bunch of Jak files, edit the *composed* files, and run `unmixin` on the edited files to back-propagate the changes to the original feature files. For example, suppose we add a comment to the bottom-most class in the extension chain of Figure 5:

```
Source Idiv "../Idiv/calc.jak";
class calc extends calc$$Iadd {
  void divide() {
    // *new* divide and pop stack
    e0 = e0.divide( e1 );
    e1 = e2;
  }
}
```

By running `unmixin`, this change is propagated back to the `Idiv/calc.jak` file:

```
> cd C
> unmixin calc.jak
```

See for yourself that the change was made. Here are things to remember about `unmixin`:

- *it can take any number of Jak files on its command line,*
- *the body of the class or interface in the command-line file will replace the body of the class or interface in the original file,*
- *implements declarations are also propagated, and*
- *don't change the contents of the source statements!*

`unmixin` updates the original uncomposed files only if changes to its composed counterpart have been updated.

4.3 reform

reform is a pretty-printing tool that formats unruly Jak files (and Java files!) and makes them unbelievably beautiful. Consider the 1-line `calc.jak` file:

```
refines class calc { void divide() { e0 = e0.divide( e1 ); e1 = e2; } }
```

By running:

```
> reform calc.jak
```

reform copies the original file into `calc.jak~` and updates `calc.jak` to be:

```
refines class calc {
  void divide() {
    e0 = e0.divide( e1 );
    e1 = e2;
  }
}
```

4.4 Equation Files

As we said earlier, AHEAD is a theory for structuring and synthesizing documents of all kinds by composing features. We introduced Jak file (i.e., code) representations earlier, and now we introduce a second.

Typing in equations on the command line to **composer** can be tedious, particularly if equations involve more than a few terms. **composer** takes an alternative specification, called an *equation file*, which is a list of units. The order in which the units are listed is from inside-out, and the name of the equation is the name of the equation file.

For example, the equation $A = B \bullet C$ would be represented by the equation file `A.equation` whose contents is:

```
# base feature listed first!
C
B
```

Where any line beginning with # is a comment. Like other AHEAD artifacts, equation files can be composed. File `A.equation` above is a “value”. An equation file that is an extension has the special term **super** as one of its units. An extension of `A.equation` that puts **E** before **C** and **F** after **B** is `R.equation`:

```
E
super
F
```

A composition of the above files is:

```
> composer --target=c.equation A.equation R.equation
```

and yields file `c.equation` with contents:

```
E
C
B
F
```

Intuitively, an equation file defines an *architectural representation* of a program as an expression. As all program representations are extendable, we now have a means by

which to specify and manipulate program architectures. We will see how such representations are useful later.

5 More Features of Jak Files

There are three additional features of the Jak language that you should know: `Super()` references, extension of constructors, and local identifiers.

5.1 The Super Construct

To invoke a method `m(int x, float y)` of a superclass in Java, you write:

```
super.m(x,y);
```

In Jak files, use the `Super` construct instead:

```
Super(int, float).m(x,y);
```

`Super(<argument types>)` prefaces a `Super` call and lists the argument types of the method to be called. Consider the class `foo` and an extension:

```
class foo {
    void dosomething() { /*code*/ }
}

refines class foo {
    void dosomething() {
        /* more before */
        Super().dosomething();
        /* more later */
    }
}
```

In this example, the `Super` references the `dosomething()` method prior to its extension. A `jumpack` composition of these definitions is shown in Figure 6a. Observe that the original `dosomething()` method is present in `foo`, except that it has been renamed, along with its references. The corresponding `mixin` composition is shown in Figure 6b. When `jak2java` translates Figure 6b, `Super(...)` references are replaced by “`super`”. In general, always use the `Super(...)` construct to reference superclass members; ATS tools do not recognize “`super`”.

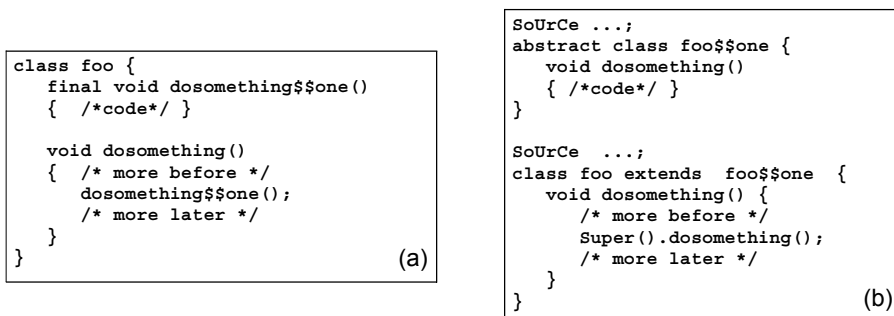


Fig. 6. `jumpack` and `Mixin` compositions

5.2 Extending Constructors

A constructor is a special method and to extend it requires a special declaration in Jak files. Consider the following file that declares a constructor:

```
class test {
  int y;
  test() { y = -1; }
}
```

An extension of `test` and its constructor is:

```
refines class test {
  int x;
  refines test() { x = 2; }
}
```

where “`refines <constructor>`” is the Jak statement that extends a particular constructor. The `jumpack` composition of these files is shown in Figure 7a. That is, the actions of the original constructor are grouped into a block and are performed first, then the actions of the constructor extension are grouped into a block and performed next. The semantically equivalent `mixin` composition is shown in Figure 7b.

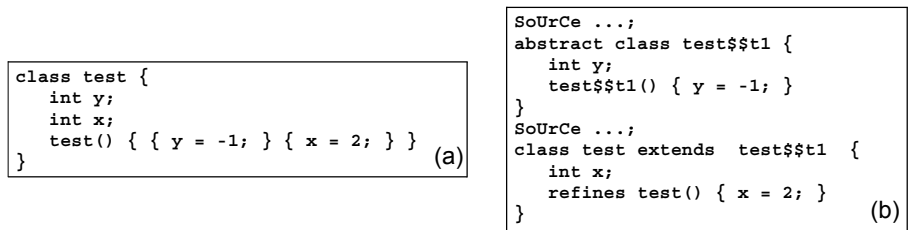


Fig. 7. Another `jumpack` and `Mixin` composition

5.3 Local Identifiers

Variables that are local to a feature are common. Such variables are used only by the feature itself, and are not to be exported or referenced by other features.

Suppose a class `bar` declares a local variable `x`, and an extension of `bar` declares a local variable, also named `x`:

```
class bar {
  int x;
}
```

```
refines class bar {
  float x;
}
```

`jumpack` is smart enough to alert you that multiple definitions of `x` are present; `mixin` isn't that smart — and you will discover the error when you compile the translated Java files and see there are multiple definitions of `x`.

The problem we just outlined isn't specific to AHEAD. In fact, it is an example of a classic problem in metaprogramming, and in particular, macro expansion. The problem is called *inadvertent capture* — i.e., multiple distinct variables are given the same names as identifiers. A general solution is to make sure that distinct variables are given unique names[16].

The way this is done in AHEAD is by using a `Local_Id` declaration. This declaration lists the set of identifiers (i.e., variable names and method names) that are *local* to a feature; ATS tools will mangle their names so that they will always be unique. So a better way to define the above is:

<pre>Local_Id x; class bar { int x; }</pre>	<pre>Local_Id x; refines class bar { float x; }</pre>
--	--

The `jumpack` and `mixin` composition of the above two files are:

<pre>class bar { int x\$\$one; float x\$\$two; }</pre>	<pre>SoURce ...; abstract class bar\$\$one { int x\$\$one; } SoURce ...; class bar extends bar\$\$one { float x\$\$two; }</pre>
--	--

where local names are replaced with their mangled counterparts so that their names no longer conflict.

6 A More Complex Example

Consider model `L`, which defines a set of programs that implement linked lists:

`L = { sgl, dbl, sgldel, dbldel }`

The lone value is `sgl` which contains a pair of classes, `list` and `node`, that implement a bare-bones singly-linked list (Figure 8a-b).

An extension of `sgl` is `dbl`, which converts the program of `sgl` into a doubly-linked list. `dbl` is a crosscut that augments the `node` class with a `prior` pointer, adds a

<pre>class list { node first = null; void insert(node n) { n.next = first; first = n; } }</pre> <p style="text-align: center;">(a) L/sgl/list.jak</p>	<pre>refines class list { node last = null; void insert(node n) { if (last == null) last = n; if (first != null) first.prior = n; Super(list).insert(n); n.prior = null; } }</pre> <p style="text-align: center;">(c) L/dbl/list.jak</p>
<pre>class node { node next = null; }</pre> <p style="text-align: center;">(b) L/sgl/node.jak</p>	<pre>refines class node { node prior = null; }</pre> <p style="text-align: center;">(d) L/dbl/node.jak</p>

Fig. 8. The `sgl` and `dbl` Layers

`last` pointer to the `list` class, and extends the `insert` method so that the values assigned to the `last` and `prior` pointers are consistent (Figure 8c-d).

The composition `both = dbl•sgl` yields the doubly-linked list program of Figure 9. The code underlined originates from the `dbl` extension.

<pre>class list { node first = null; node last = null; final void insert\$\$\$sgl(node n) { n.next = first; first = n; } void insert(node n) { if (last == null) last = n; insert\$\$\$sgl(n); x.prior = null; } }</pre>	<pre>class node { String constant; node next = null; node prior = null; }</pre>
(a) L/both/list.jak	(b) L/both/node.jak

Fig. 9. Composition `dbl•sgl`

Now suppose we want to enhance the design of our list programs by adding a `delete` method. `sgldel` does exactly this for singly-linked lists: it adds a `delete` method to the `list` class (Figure 10a). We can use `sgldel` in a composition `slist` that defines a singly-linked list with both `insert` and `delete` methods:

$$slist = sgldel \bullet sgl$$

To create a doubly-linked list that has both `insert` and `delete` methods requires an extension `dbldel` (Figure 10b). `dbldel` converts the singly-linked list deletion algorithm of `sgldel` to a doubly-linked list deletion algorithm by replacing (or overriding) the `findAndDelete` method.

The following equations yield identical programs for inserting and deleting elements from a doubly-linked list. The reason why they are equivalent is that the extensions `dbl` and `sgldel` are independent of each other, and thus can be composed in any order.

$$dlist = dbldel \bullet dbl \bullet sgldel \bullet sgl \tag{8}$$

$$= dbldel \bullet sgldel \bullet dbl \bullet sgl \tag{9}$$

Model Exercises

- [10] Suppose other operations for traversing the list are added. How would this impact model `L`? What about the operation `reverse()`, which reverses the order in which nodes are listed?
- [11] Suppose an “ordering” feature is added to a list, meaning that nodes have keys and are maintained in ascending key order. How would this feature impact `L`?
- [12] Consider a “monitor” feature, which precludes more than one thread to access a list at a time. How would this feature impact `L`? How would it be defined?

```

refines class list {
    void delete( node n ) {
        if ( n == first ) {
            first = first.next;
        }
        else
            findAndDelete( n );
    }
    void findAndDelete( node n ) {
        node prev = first;
        while ( prev != n )
            prev = prev.next;
        prev.next = n.next;
    }
}

```

(a) L/sgldel/list.jak

```

refines class list {
    void findAndDelete( node n ) {
        if ( n == last )
            last = last.prior;
        if ( n.prior != null )
            n.prior.next = n.next;
        if ( n.next != null )
            n.next.prior = n.prior;
    }
}

```

(b) L/dbldel/list.jak

Fig. 10. The **sgldel** and **dbldel** Layers

Tool Exercises

The directory structure for **L** is:

```

L/sgl/list.jak           // see Figure 8a
L/sgl/node.jak         // see Figure 8b
L/dbl/list.jak         // see Figure 8c
L/dbl/node.jak         // see Figure 8d
L/sgldel/list.jak     // see Figure 8a
L/dbldel/list.jak     // see Figure 8b

```

The files of Figure 9 are the result of evaluating the equation $\text{both} = \text{dbl} \bullet \text{sgl}$ using the **composer** tool:

```

> cd L
> composer --target=both sgl dbl

```

The generated directory structure is:

```

L/both/list.jak        // see Figure 9a
L/both/node.jak       // see Figure 9b

```

[13] What is a **guids1** model of **L**?

6.1 Multi-dimensional Models and Origami

There remains a fundamental relationship among the features of **L** that we have not yet captured. Consider the following incorrect compositions:

```

error1 = dbl•sgldel•sgl
error2 = dbldel•sgldel•sgl

```

Both define programs that are partially and thus incorrectly implemented. **error1** is a program whose **insert** method works on a doubly-linked list, but whose **delete** method works only on a singly-linked list. **error2** is a program whose **insert** method works on a singly-linked list, but whose **delete** method works for a doubly-linked list.

The problem is that if a data structure is extended (i.e., a singly-linked list becomes doubly-linked), then *all* of its operations should be updated to maintain the consistency of this extension, and not just some. That is, if a singly-linked list has both `insert` and `delete` operations, when the structure becomes doubly-linked, both operations must be updated to work on doubly-linked lists. Equivalently, if a feature adds a new method to a data structure, then that method must work for that data structure and not some other structure.

Although this is an elementary example, it is representative of a large class of problems in FOP, namely that a model defines a group of features that are not truly independent and this group must be applied in lock-step — all or nothing — manner. Whenever you notice this phenomena, realize that these groups represent features of multidimensional models, which we explain further and illustrate in the following paragraphs.

Create a matrix, called an *Origami matrix* (which is a 2-dimensional model), where rows represent operations (`insert`, `delete`), and columns represent structure variants (`singleLink`, `doubleLink`). Entries of this matrix are the features of `L` (see Table 1). This matrix can be extended to handle other operations (sort, find) and other structure variants (ordered-lists, monitors, etc.).

Note: what we have done is to identify orthogonal feature sets as ‘data structure operations’ and ‘data structure variants’; these feature sets define the units of different dimensions of a 2-dimensional model or matrix.

Table 1. Origami Matrix for L

	<code>doubleLink</code>	<code>singleLink</code>
<code>insert</code>	<code>dbl</code>	<code>sgl</code>
<code>delete</code>	<code>dbldel</code>	<code>sgldel</code>

Suppose the rows of this matrix are composed (or *folded* — hence the name “Origami”), where the corresponding entries in each column are composed (Table 2):

Table 2. Row-Composed Origami Matrix

	<code>doubleLink</code>	<code>singleLink</code>
<code>delete•insert</code>	<code>dbldel•dbl</code>	<code>sgldel•sgl</code>

Study the entries of Table 2. Consider the entry in the `singleLink` column: `sgldel•sgl` defines a singly-linked program `s` that has both an `insert` and `delete` method. The entry in the `doubleLink` column, `dbldel•dbl`, defines an extension of `s` that converts its `insert` and `delete` methods to work on a doubly-linked list. Thus by composing the `delete` row with the `insert` row of Table 1, we synthesize a data structure that has multiple methods, and an extension of that data structure that consistently updates these methods. This interpretation holds if more rows (operations) or more list features (columns) are added.

The columns of Table 2 can be composed to yield a 1×1 matrix whose entry is an expression that defines a doubly-linked list with insert and delete methods (Table 3). This expression is identical to equation (8).

Table 3. A Completely Folded Matrix

	doubleLink•singleLink
delete•insert	dbl•del•dbl•sgl•del•sgl

Now instead of composing rows of Table 1, let's compose the columns, where corresponding entries in each row are composed (Table 4):

Table 4. Column Composed Origami Matrix

	doubleLink•singleLink
insert	dbl•sgl
delete	dbl•del•sgl•del

The entry in the **insert** row, **dbl•sgl**, defines program **D** that implements a doubly-linked list with an **insert** method. The entry in the **delete** row, **dbl•del•sgl•del**, defines an extension of **D** that adds a **delete** method. By composing the columns of Table 1, we have synthesized a data structure with a single (**insert**) method, and an extension that adds a **delete** method to this structure. Again, this interpretation holds if we add more rows (methods) or more columns (features) to Table 1. By folding the rows of Table 4, a 1×1 matrix is produced whose lone entry is equation (9). As a general rule, as long as the order in which rows and columns (that is, 'data structure operation' features or 'data structure variant' features) are composed is legal, the resulting equations in a fully-folded matrix should be equivalent. (If they are not, then a dimension is missing in the design).

Origami matrices capture fundamental relationships among groups of features: to build consistent and correct programs, it is often necessary to apply an entire group of features at once [6]. A matrix representation of these relationships works because the set of features along one dimension are *orthogonal* to those of another. In our example, the set of methods that can be used with a data structure is orthogonal to the set of data structure variants.

Although this is a simple example, Origami applies at much greater levels of granularity. For example, ATS has five tools — including **jumpack**, **mixin**, and **unmixin** — each having over 30K LOC, and totalling over 150K LOC. These tools are synthesized by folding a 3-dimensional ($8 \times 6 \times 8$) Origami matrix, where the dimensions are: language features \times tool features \times language feature interactions [6].

6.2 The Meaning of Origami

Why is Origami significant? There are several reasons, all of which capture important generalizations of equational program specifications.

In earlier sections, we defined a program by a single equation. Origami generalizes this idea, so that a program is defined by a set of k equations, one per dimension. This has a significant impact on reducing the complexity of a program specification. Suppose each of the k equations has n terms. Thus, a program specification in Origami is of length $O(nk)$. Yet, the matrix that is folded into a single expression would have $O(n^k)$ terms! That is, *Origami exponentially shortens specifications of product-line programs* [6]. Or stated another way, Origami enables very simple specifications for very complex programs.

Here's another interesting question: what is the algebraic meaning of matrix folding? The answer is evident when we interpret the composition operator (\bullet) as addition [12].

Composition in AHEAD is similar to summation. Suppose to build program \mathbf{P} , we start with a base feature \mathbf{F}_0 and progressively add on features \mathbf{F}_1 , \mathbf{F}_3 , and \mathbf{F}_7 . Instead of using \bullet , we use $+$ to denote composition:

$$\mathbf{P} = \mathbf{F}_7 + \mathbf{F}_3 + \mathbf{F}_1 + \mathbf{F}_0 \quad (10)$$

Here's another way to represent \mathbf{P} . Suppose \mathbf{F} is the model that contains features \mathbf{F}_0 , \mathbf{F}_1 , \mathbf{F}_3 , and \mathbf{F}_7 . In general \mathbf{F}_i is the i th feature of model \mathbf{F} . Let \mathbf{E} be the sequence of subscripts whose features we are to sum. For equation (10), $\mathbf{E} = (0, 1, 3, 7)$. We could equivalently write (10) as a summation:

$$\mathbf{P} = \sum_{i \in \mathbf{E}} \mathbf{F}_i$$

Now suppose our model is two dimensional. Let \mathbf{m} denote a two-dimensional Origami matrix, where $\mathbf{m}_{i,j}$ is the element in the i th row and j th column. When the matrix is folded first by rows then by columns, this corresponds to summing the matrix by columns then by rows. When the matrix is folded by columns and then by rows, this corresponds to summing by rows and then columns. Let \mathbf{r} be the sequence of subscripts in which rows are folded; let \mathbf{c} be the sequence of subscripts in which columns are folded. Origami expresses the equivalence of the summation of elements of a matrix in different orders:

$$\mathbf{P} = \sum_{i \in \mathbf{r}} \sum_{j \in \mathbf{c}} \mathbf{m}_{i,j} = \sum_{j \in \mathbf{c}} \sum_{i \in \mathbf{r}} \mathbf{m}_{i,j}$$

That is, an Origami matrix is a k -dimensional ‘‘cube’’, which when summed across different dimensions yields a program in a product-line. Summation of matrix entries and permuting the order in which entities are added, are familiar ideas in mathematics. The name ‘‘Origami’’ is really a visual interpretation of matrix summation.

Finally, it is worth noting that Origami and multi-dimensional models are historically related to a fundamental problem in program design called the ‘‘expression problem’’, which 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 [22][18]. The FOP contribution to this is to show how the idea scales to the synthesis of large programs [6].

6.3 Metamodels and Model Synthesis

How are Origami matrices represented in AHEAD? Before we can answer this question, we need to introduce an important concept in modeling called metamodels. A *metamodel* is a model whose instances are themselves models. Consider model **m**, which has units **a**, **b**, and **c**:

$$\mathbf{m} = \{ \mathbf{a}, \mathbf{b}, \mathbf{c} \}$$

Consider metamodel **mm**, which also has three units, each being a set with a single unit:

$$\begin{aligned} \mathbf{mm} &= \{ \mathbf{AAA}, \mathbf{BBB}, \mathbf{CCC} \} \\ &= \{ \{ \mathbf{a} \}, \{ \mathbf{b} \}, \{ \mathbf{c} \} \} \end{aligned}$$

A model can be synthesized by composing metamodel units. The **mm** equation for model **m** is:

$$\mathbf{m} = \mathbf{AAA} \bullet \mathbf{BBB} \bullet \mathbf{CCC}$$

In this particular case, because there are no units in common with **AAA—CCC**, composition reduces to set-union. The interesting thing about metamodels is that they are identical to models. That is, a model or metamodel is a set of units, where each unit may be a set. Further, the composition operator for units of metamodels (**•**) is the same operator for units of models (**•**).

An Origami matrix is a metamodel. Consider the example of an $n \times m$ matrix **o** where o_{ij} denotes the row *i* column *j* element of **o**. There are two ways to represent this matrix as a tree, i.e., as a model of models. One way is to decompose the matrix first into rows, and then each row into columns (Figure 11a). Another way is to organize by columns first, and then rows (Figure 11b). This idea scales to arbitrary dimensional matrices.

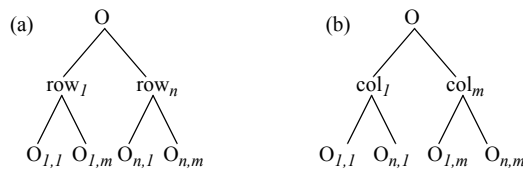


Fig. 11. Matrix Embeddings in Trees

6.4 Representing Origami Matrices

Now lets consider how to represent Origami matrices. Consider a row-dominant decomposition. Figure 12a shows our example matrix, where entries are equation files that have the same name (**eqn.equation**). Entry subscripts denote (to us) their true identity. Figure 12b is the corresponding row-oriented metamodel; Figure 12c is its AHEAD directory structure. Figure 12d-g are the contents of the equation files.

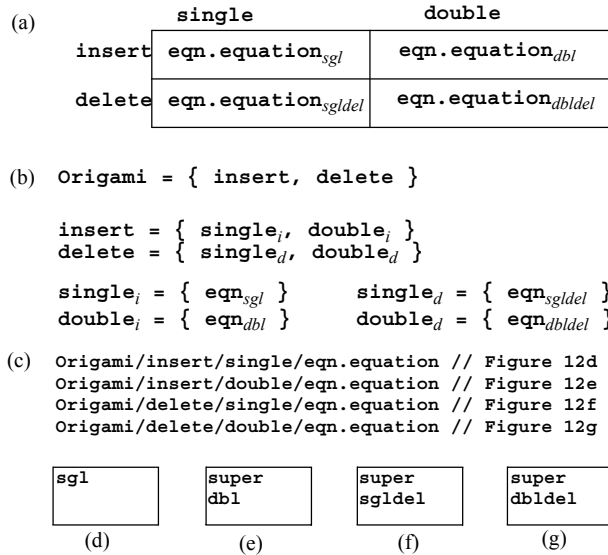


Fig. 12. Row-Dominant Embedding of a Matrix

Why do we use this particular representation of a matrix? Why use equation files, rather than embedding the actual feature directories themselves? The answer: convenience. Try to create such a hierarchical directory yourself, where instead of equation files, you have feature directories. It's hard to navigate such a directory structure, let alone maintain it. The simpler the representation the better. So it is common that we have a flat model directory (where features are immediate subdirectories), and a separate Origami directory which defines the multi-dimensional relationship among features using equation files.

Model Exercises

[14] Expand the Origami matrix to handle more data structure operations and variants, such as list element updates and key ordered lists.

Tool Exercises

To fold a 2-dimensional matrix, you need to invoke **composer** twice: once to compose rows and a second time for columns. (For a k -dimensional matrix, we would invoke **composer** k times). So to produce the AHEAD equivalent of Table 2, we compose the rows of the **Origami** model to produce model **Table2 = delete•insert**:

```

> cd Origami
> composer --target=Table2 insert delete
    
```

The resulting model **Table2** is depicted in Figure 13a, and its synthesized AHEAD directory structure in Figure 13b, and the contents of the equation files in Figure 13c-d.

```
(a) Table2 = { single, double }
      single = { eqnsgldel*eqnsgl }
      double = { eqndbldel*eqndbl }

(b) Origami/Table2/single/eqn.equation // Figure 13c
     Origami/Table2/double/eqn.equation // Figure 13d

(c) 

|        |
|--------|
| sgl    |
| sglde1 |

      (d) 

|        |
|--------|
| super  |
| dbl    |
| dblde1 |


```

Fig. 13. AHEAD Origami Metamodel

To produce the 1×1 matrix of Table 3 or equation (8), we compose the columns (named **single** and **double**) using the following command:

```
> cd Table2
> composer --target=both single double
> cd both
> jak2java *.jak
```

[15] Represent the matrix of Figure 12a by columns, and repeat the above folding.

Model Exercises

- [16] Create two different GUIs for a calculator: one uses the standard 2D keypad, a second uses text fields to enter values and operations. A calculator will use one (but not both) of these GUIs. Operations on both GUIs are buttons. So when a calculator is extended by a new operation, its GUI will be extended also. Express this relationship between operations and GUIs as an Origami matrix.
- [17] Generalize the model in [16] that permits multiple GUIs per calculator. One idea would use tabs, one tab per different GUI. Implement your model.

7 What's Next?

There are many interesting topics and capabilities that we have yet to explore (or develop) for AHEAD. Here are just a few. If you are interested in learning more about these topics, see [1][5].

7.1 Extensible Languages

There are all sorts of non-Java extensions to the Jak language that we haven't talked about, including:

- *metaprogramming* — the ability to assign code fragments to variables, the ability to compose code fragments via escape substitutions, hygienic macros.
- *state machines* — an embedded DSL for supporting the definition and extension of state machines [7].

7.2 Compiler-Compiler Tools

ATS has a sophisticated set of compiler-compiler tools that are used to (a) define base grammars, (b) define grammar extensions, and (c) to synthesize grammars by composing base grammars with extensions. Grammars are yet another representation of a program, in this case, a compiler, and ATS has tools for defining and composing grammars and generating Java files from them [1].

7.3 Generating and Optimizing MakeFiles

The idea of modules as hierarchical collections of related artifacts is powerful. A paradigm of AHEAD that we have explored so far is that of *composition*: that artifacts of a program can be composed from previously defined artifacts. But there is another way in which program artifacts can be produced: by *derivation*. For example, Java files can be produced from Jak files by the `jak2java` tool; class files can be produced from Java files by the `javac` compiler, and so on. A general paradigm is depicted in Figure 14: an artifact can be produced by first composing it from more elementary artifacts, followed by a derivation. Or equivalently, it can be produced by deriving a set of artifacts from more elementary artifacts, and then composing the derived representations³. This leads to the following fundamental distributive algebraic relationship (11).

$$\begin{aligned} \text{derive}(\text{artifact}_i \bullet \text{artifact}_j) &= \\ \text{derive}(\text{artifact}_i) \bullet \text{derive}(\text{artifact}_j) & \end{aligned} \quad (11)$$

Ultimately, we want to specify an entire program — all of its composed and derived representations — as a set of equations. Although ATS does not yet have such a tool, one can imagine a specification like:

$$\begin{aligned} \text{Using } L; \\ i3 = \text{javadoc}(\text{javac}(\text{jak2java}(\text{sg1del}(\text{sg1})))); \end{aligned} \quad (12)$$

where the `Using` clause tells this tool that `sg1del` and `sg1` are units in model `L`, and by inference, `composer` should be used to compose them. The resulting module will have `.jak`, `.java`, `.class` and `.html` files. The `jak2java` tool, when applied to the module of `sg1del(sg1)`, translates all Jak files to Java files and adds them to the module. Similarly, the `javac` operation compiles all Java files and adds their `.class` files to the module. The `javadoc` operation will generate JavaDoc `.html` files from the generated Java files, and so on, progressively enlarging the contents of that module/directory. In effect, (12) is really an equational representation of a makefile! More on this shortly.

The lesson that we learned from relational query optimizers is that an expression represents a program design and expressions (and hence designs) can be optimized. In this particular example, there really isn't anything to optimize. There is, though, a particular sequencing of the application of the `javadoc`, `javac`, and `jak2java`

³ Figure 14 can be generalized further, so that multiple output artifacts can be derived from a single input artifact, and vice versa.

operations that must be imposed. (In fact, this really is the only legal ordering of these operations for this equation). So notions of design rules also apply to tool operations. But as equations become more complicated, there is the possibility of optimization. In some of our larger examples using Origami, generating common subexpressions among different sets of tools arises. Evaluating common subexpressions once, and not many times, is an important optimization that a tool should be able to achieve automatically [14].

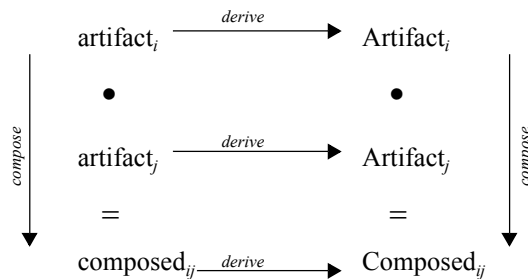


Fig. 14. Compose vs. Derive

The big picture is depicted below. Given an equational representation of a program that specifies both the artifacts that are to be composed and those that are to be derived, a tool will expand the equations and perform optimizations to synthesize the resulting program in an efficient manner. The tool will then produce an optimized set of equations, and a generator will translate these equations into a *makefile* — a functional-like language that efficiently executes equational specifications [14].

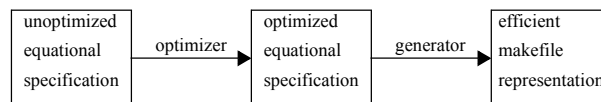


Fig. 15. Generation and Optimization of MakeFiles

7.4 Type Systems

As mentioned earlier, extensions are functions that appear untyped. In fact, function inputs and outputs have definite constraints. Our tools assume that the correct types are both being input and output. In general, this is bad assumption.

Question: how does one type a program? Should Java interfaces be used? How does typing generalize to, say, grammar files or equation files? What is a general mechanism for typing arbitrary artifacts and their extensions? At the present time, there are no solutions to these problems.

7.5 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is closely related to FOP. Both deal with modules that encapsulate crosscuts of classes, and both express program extensions. FOP uses a subset of the “advising” capabilities of AOP, namely those that use execution pointcuts. However, the primary difference between AOP and FOP is their composition models. FOP treats aspects as functions that map programs, and uses function composition as the means to compose aspects. This leads to algebraic representations of programs and a simple means to perform program reasoning with aspects.

In contrast, AOP uses a complex model of aspect composition (e.g., precedence rules) that complicates program reasoning using aspects and makes step-wise development difficult [19]. AOP and FOP are thus not directly comparable, but are instances of a more general paradigm of automated software development — one that composes aspects by FOP function composition and uses the full power of AOP pointcuts.

8 Conclusions

Just as the structure of matter is fundamental to chemistry and physics, so too must the structure of software be fundamental to computer science. Unfortunately, our understanding of software structure is in its infancy. Today, software design is an art. As long as it remains so, our ability to automate rote tasks of program design and synthesis will be limited. And software engineering will be more of a craft than a discipline.

Software designs can be given mathematical precision when expressed as a composition of features. We have presented a simple and elegant theory of program design, backed by years of implementation and experimentation, that brings together key elements in the future of software development: generative programming, domain-specific languages, automatic programming, and step-wise development. Generative programming gives our theory its mathematical backbone: functions can map programs. Domain-specific languages give programming artifacts their form: these are the artifacts that functions transform. Automatic programming underscores AHEAD as a simple model that relates automated reasoning, compositional programming, and design optimization by algebraic reasoning. And step-wise development is a practical way of controlling complexity. AHEAD provides an algebraic foundation for understanding program development on a larger scale.

This paper has explored basic concepts of FOP and a (small) subset of the tools of the AHEAD tool suite. For the most recent results, see our web site [17] and consult the AHEAD documentation [1].

Acknowledgements. This work is sponsored by NSF's Science of Design Project #CCF-0438786. I thank the referees, Ralf Lämmel, João Saraiva, and Joost Visser for their helpful comments.

Suggested Reading

- [1] AHEAD Tool Suite, <http://www.cs.utexas.edu/users/schwartz/ATS.html> **ATS documentation.**
- [2] R. Balzer, "A Fifteen-Year Perspective on Automatic Programming", *IEEE Trans. Software Engineering*, November 1985, pp. 1257-1267. **Mid-80's state-of-art-report on automatic programming.**
- [3] D. Batory and S. O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components". *ACM Trans. Software Engineering and Methodology*, October 1992, pp. 355-398. **The GenVoca Model.**
- [4] D. Batory: "The Road to Utopia: A Future for Generative Programming". *Domain-Specific Program Generation 2003*, Lecture Notes in Computer Science #3016, pp. 1-18. **Relationship of query optimization to AHEAD.**
- [5] D. Batory, J.N. Sarvela, A. Rauschmayer, "Scaling Stepwise Refinement", *IEEE Trans. Software Engineering*, June 2004, 355-371. **The AHEAD model.**
- [6] D. Batory, J. Liu, J.N. Sarvela, "Refinements and Multi-Dimensional Separation of Concerns", *ACM SIGSOFT 2003*, pp. 48 - 57. **A sophisticated example of Origami.**
- [7] D. Batory, C. Johnson, R. MacDonald, and D. von Heeder, "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study", *ACM Transactions on Software Engineering and Methodology*, April 2002, 191-214. **A product-line that needs both extensions and embedded DSLs.**
- [8] D. Batory, G. Chen, E. Robertson, and T. Wang, Design Wizards and Visual Programming Environments for GenVoca Generators, *IEEE Transactions on Software Engineering*, May 2000, 441-452. **Explains relationship between automatic programming and GenVoca equation optimization.**
- [9] D. Batory and B.J. Geraci. Composition Validation and Subjectivity in GenVoca Generators, *IEEE Transactions on Software Engineering*, February 1997, pp. 67-82. **Early form of design rule checking.**
- [10] D. Batory, "Feature Models, Grammars, and Propositional Formulas", *Software Product-Line Conference (SPLC) 2005*, pp. 7-20. **Generalizes results in.**
- [11] M. de Jong and J. Visser, "Grammars as Feature Diagrams", 2002. *Workshop on Generative Programming (GP2002)*, Austin Texas, USA. April 15, 2002. **Relates feature diagrams to grammars.**
- [12] E.J. Jung, "Feature Oriented Programming and Product Line Architectures for Open Architecture Robot Software", M.Sc. Thesis, Dept. Mechanical Engineering, University of Texas at Austin, 2004. **Application of FOP ideas to robotics, Origami.**
- [13] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. "Feature-Oriented Domain Analysis (FODA) Feasibility Study". Technical Report, CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990. **First significant paper on features and product-lines.**
- [14] J. Liu and D. Batory, "Automatic Remodularization and Optimized Synthesis of Product-Families", *Generative Programming and Component Engineering (GPCE)*, October 2004, pp. 379-395. **Shows how sets of equations can be optimized.**
- [15] R. E. Lopez-Herrejon and D. Batory, "A Standard Problem for Evaluating Product-Line Methodologies", *Generative and Component-Based Software Engineering (GCSE 2001)*, Erfurt, Germany. pp. 10-24. **A simple product-line defined using the GenVoca model.**

- [16] E. Kohlbecker, D.P. Friedman, M. Felleisen, and B. Duba, “Hygienic Macro Expansion”, *SIGPLAN '86 ACM Conference on Lisp and Functional Programming*, pp. 151-161. **Classic paper on the inadvertent capture problem.**
- [17] Product-Line Architecture Research Group. <http://www.cs.utexas.edu/users/schwartz/>
- [18] R. Lopez-Herrejon, D. Batory, and W. Cook, “Evaluating Support for Features in Advanced Modularization Technologies”, *European Conference on Object-Oriented Programming (ECOOP)*, July 2005, pp. 169-194. **Using the expression problem to evaluate different modularization technologies.**
- [19] R. Lopez-Herrejon, D. Batory, and C. Lengauer, “A Disciplined Approach to Aspect Composition”, *Program Evaluation and Program Manipulation (PEPM)* 2005, pp. 68-77. **Formalizes the AspectJ composition model.**
- [20] P. Selinger, P. M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, “Access Path Selection in a Relational Database System”, *ACM SIGMOD 1979*, pp. 23-34. **Classic paper on relational query optimizers.**
- [21] T. Teitelbaum and T. Reps, “The Cornell Program Synthesizer: a Syntax-Directed Programming Environment”, *CACM*, v.24 n.9, pp. 563-573, Sept. 1981. **Classic paper on syntax-directed editors.**
- [22] M. Torgersen, “The Expression Problem Revisited. Four New Solutions Using Generics”, *ECOOP 2004*, pp. 123-146. **A recent paper on the Expression problem.**