The Expression Problem as a Product Line and its Implementation in AHEAD

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

Abstract. Software product lines are defined in terms of feature sets. A family member is defined with the set of features it implements. Several modularization techniques have recently appeared so in an effort to evaluate their usability to implement product lines, we propose a simple yet illustrative product line example based on the *expression problem* that we believe captures the most basic requirements to modularize features.

1 Problem Description

The *expression problem* is a fundamental problem of software design where extensibility is defined by adding features to a program to support a mix of new operations and data types [8][14]. It has been widely studied within the context of programming language designs where the focus is to achieve data type and operator extensibility in a type-safe manner, without resorting to code modification or repetition and avoiding run-time type errors [13][9]. Though important issues by themselves, our focus in this report is different as we concentrate on the *design and synthesis* aspects of the expression problem: How can features be modularized and how can they be composed to build variations of a program?

We start with Torgersen's expression problem [13]. The goal is to define data types to represent expressions of the following language:

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

Associated with this grammar is an operation called `Print` that displays the string representation of an expression. For example, the expression `2+3` can be 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`".

This application can be extended by adding new data types. To support the negation of expressions requires the following grammar extension:

```
Exp  :: = ... | Neg
Neg  :: = "-" Exp
```

where `...` denotes the previous right-hand side of `Exp`. The application can also be extended with new operations, such as `Eval`, a function that evaluates expressions and re-

turns their numeric value. Applying the operation `Eval` to the tree of expression `2+3` yields `5` as result.

A natural representation of the expression problem is a two-dimensional matrix [14][8][9]. *Each matrix entry defines a feature module whose code implements the operation given by the column for the data type given by the row*. Consider the original grammar with data types `Lit` and `Add`. Supporting operation `Print` for them can be represented as a 2×1 matrix (Figure 1a) where the vertical dimension specifies data types and the horizontal dimension specifies operations. As a naming convention throughout the report, 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`.

When a new data type is added, the change is reflected by adding a new row to the matrix and implementing the operations in each of the columns. Figure 1b shows the result of adding data type `Neg` to Figure 1a. Symmetrically, adding a new operation modifies the matrix by adding a new column and implementing the operation on the data types specified in each of the rows. The result of adding operation `Eval` to Figure 1b is shown in Figure 1c.

(a)

	Print
Lit	lp
Add	ap

(b)

	Print
Lit	lp
Add	ap
Neg	np

(c)

	Print	Eval
Lit	lp	le
Add	ap	ae
Neg	np	ne

(d)

	Print	Eval
Lit	lp	le
Add	ap	ae

Figure 1. Matrix representation of Expression Problem

Feature sets that can be arranged as matrices of two or more dimensions are *multi-dimensional*. The expression problem is two-dimensional, meaning that it is a design that is extensible along either (data type or operation) dimension. *n*-dimensional means any of the *n* dimensions can be extended [5][6].

A feature matrix is a design for a family of programs (a.k.a. product-line). An architect specifies a particular application by selecting the desired row and column features, and composing the identified modules. In our example, this means trimming the feature matrix of unneeded rows and columns and composing the resulting matrix entries. Figure 1d is the matrix for an application, called `LitAdd`, that supports the `Print` and `Eval` operations on the `Lit` and `Add` data types. An intuitive way to express the composition of modules in a matrix is a *summation*: the desired application is the sum of the contributions of its feature modules, where summation is associative (e.g., left-to-right evaluation yields the same result as right-to-left):

```
LitAdd = ae + ap + le + lp                                    (1)
```

The process by which the contents of a matrix are mapped to a summation is not relevant to this report, but a way to do it is explained in [5][6]. What is important, though, is that it conveys an order in which modules are composed.

2 Implementation in AHEAD

AHEAD (Algebraic Hierarchical Equations for Application Design) is a feature module and feature composition technology based on step-wise refinement [7][5][1]. It was created to address the issues of feature-based development of product-lines, of which the expression problem is a typical example.

AHEAD partitions features into two categories: constants and functions. *Constants* are base programs that modularize any number of classes and interfaces. *Functions* are program deltas, which modularize changes made by the features. Particular applications are *expressions*, which are compositions of feature "constants" and "functions".

`LitAdd` (or really, any program derivable from an expression problem feature matrix) has a simple Java representation: It has an interface `Exp`, whose methods `print()` and `eval()` are the `Print` and `Eval` operations, and classes `Lit`, `Add`, and `Neg` are data types that implement interface `Exp`. Further `LitAdd` has class `Test` that creates instances of the data type classes and invokes their `print()` and `eval()` methods.

AHEAD tools use a language, called *Jak* [7], that is a superset of Java. The implementation of features like `lp`, whose elements are standard classes and interfaces, uses pure Java constructs as follows:

```
public interface Exp { void print(); }
class Lit implements Exp {
    int value;
    Lit (int v) { value = v; }
    void print() { System.out.print(value); }
}
class Test {
     Lit ltree;
     Test() { ltree = new Lit(3); }
     void run() { three.print(); }
}
```

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

```
   }
}
```

This feature extends: a) interface `Exp` with method `eval()`, b) class `Lit` with a implementation of `eval()`, and c) class `Test` with extension to method `run()` that calls method `eval()` on `ltree`. (That is, `Super.run()` invokes the original method and the `System.out.println()` is executed afterwards).

The following is the implementation of feature `ap`. This feature adds new class `Add` and refines `Test` class by adding a field `atree` that is assigned to a new object of class `Add` in the constructor extension of `Test`.

```
class Add implements Exp {
     Exp left, right;
     Add (Exp l, Exp r) { left = l; right = r; }
     void print() { left.print(); System.out.print("+");
                    right.print(); }
}
refines class Test {
     Add atree;
     refines Test() { atree = new Add(ltree, ltree); }
     void run() { Super.run(); atree.print(); }
}
```

Feature `ae` consists of two refinements: a) it refines class `Add` with the implementation of the `eval()` method, b) it refines class `Test` with an extension to method `run()` that calls `eval()` method on the `atree` field defined in feature `ap`.

```
 refines class Add implements Exp {
    int eval() { return left.eval() + right.eval(); }
 }
 refines class Test {
     void run() {
      Super.run();
       atree.print(); System.out.println("= " + atree.eval());
     }
 }
```

Feature `np` is similar to feature `ap`. It defines class `Neg` to represent the negative numbers with the `print()` method. It also refines class `Test` with field `ntree` that is assigned in its constructor refinement to a new object of type `Neg`.

```
class Neg implements Exp {
     Exp expression;
     Neg (Exp e) { expression = e; }
     void print() {
         System.out.print("-("); expression.print();
         System.out.print(")");
     }
 }
refines class Test {
     Neg ntree;
     refines Test() { ntree = new Neg(ltree); }
     void run() { Super.run();ntree.print(); }
}
```

Finally feature ne consists of a refinement to class Neg to implement eval() method and a refinement to class Test with an extension to method run() that calls this method on ntree.

```
refines class Neg implements Exp {
    int eval() { return expression.eval() * -1; }
}
refines class Test {
    void run() {
        Super.run();
        ntree.print(); System.out.println("= " + ntree.eval());
    }
}
```

Composition Specification. Each feature is represented by a directory that contains files for each class and interface definition and extension. Thus, AHEAD implementation uses six directories ne, np, ae, ap, le, and lp.

The LitAdd=ae+ap+le+lp summation is evaluated using composer, the AHEAD tool for feature composition. The command line is:[1]

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

The following are other valid equations:

```
LitNeg      = ne + np + le + lp                          (2)

LitAddNeg   = ne + np + ae + ap + le + lp                (3)

LitNegAdd   = ae + ap + ne + np + le + lp                (4)

EvalPrint   = ne + ae + le + np + ap + lp                (5)
```

An important characteristic of the features implemented with AHEAD is that for all the valid product line members of the extended expression problem family, no changes are made to definition of the features modules, the variability is expressed with the equations themselves.

3 References

[1] AHEAD Tool Suite (ATS). http://www.cs.utexas.edu/users/schwartz

[2] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.

[3] D. Batory, G. Chen, E. Robertson, and T. Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE Trans. on Soft. Engr.*, May 2000, 441-452.

[4] D. Batory, and B.J Geraci, "Composition Validation and Subjectivity in GenVoca Generators". *IEEE Trans. Soft. Engr.*, February (1997) 67-82.

[5] D. Batory, R.E. Lopez-Herrejon, J.P. Martin, "Generating Product-Lines of Product-Families". *Automated Software Engineering Conference*, September 2002.

1. That features are listed in reverse order in which they are summed is an implementation/legacy oddity of composer.

[6] D. Batory, J. Liu, and J.N. Sarvela, “Refinements and Multidimensional Separation of Concerns”. *ACM SIGSOFT*, September 2003.

[7] D. Batory, J.N. Sarvela, and A. Rauschmayer, “Scaling Step-Wise Refinement”, *IEEE Trans. Soft. Engr.* June 2004.

[8] W.R. Cook, “Object-Oriented Programming versus Abstract Data Types”. Workshop on Foundations of Object-Oriented Languages, Lecture Notes in Computer Science, Vol. 173. Spring-Verlag, (1990) 151-178

[9] R.B. Findler and M. Flatt, “Modular Object-Oriented Programming with Units and Mixins”. ICFP, (1998) 94-104

[10] D.L. Parnas, “Information Distribution Aspects of Design Methodology”. TR, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, (1971).

[11] D.L. Parnas, “ On the Criteria To Be Used in Decomposing Systems into Modules”. *CACM* Dec. (1972) 1053-1058.

[12] C. Prehofer, “Feature-Oriented Programming: A Fresh Look at Objects”. *ECOOP* (1997) 419-443

[13] M. Torgensen, “The Expresion Problem Revisited. Four new solutions using generics”. *ECOOP* (2004).

[14] P. Wadler, “The expression problem”. Posted on the Java Genericity mailing list (1998)