AHEAD by Example

Eduardo Figueiredo

http://www.dcc.ufmg.br/~figueiredo

AHEAD

- Algebraic Hierarchical Equations for Application Design
  - GenVoca represents an individual program as an expression
  - AHEAD generalizes GenVoca for multiple programs and representations
  - AHEAD treats all representations (code or noncode) in a uniform way

AHEAD Tool Suite

- AHEAD formalism can be easily supported by tools
  - Current AHEAD implementation has a Java-like syntax
  - AHEAD files have the .JAK extension
- Example of a Calculator product line
  - Two calculators: BigInteger and BigDecimal
  - Math operations: sum, divide, subtract, …

Base Application

```
class calc { }
```

(a) Base/calc.jak

- Features
  - Base, BigI, BigD, Iadd, Idiv, Isub, Dadd, Ddivd, Ddivu, Dsub

Application: BigI ● Base

```
class calc { }
```

(a) Base/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

Application: BigD ● Base

```
class calc { }
```

(a) Base/calc.jak

BigI and BigD are alternative features (XOR)

```
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

Operations ● Base

```
class calc { }
```

(a) Base/calc.jak

Integer

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

(d) Idiv/calc.jak

Integer Decimal

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

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

Decimal

```
import java.math.BigDecimal;

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

(f) Ddivd/calc.jak

Operations ● Base

```
class calc { }
```

(a) Base/calc.jak

Integer

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

(d) Idiv/calc.jak

Idiv and Ddivd are alternative features (XOR)

Decimal

```
import java.math.BigDecimal;

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

(f) Ddivd/calc.jak

Product Configurations

- A software product is a GenVoca expression
- Examples of products
  - i1 = Iadd • BigI • Base
  - i2 = Isub • Iadd • BigI • Base
  - i3 = Idiv • Iadd • BigI • Base
  - d1 = Dadd • BigD • Base
  - d2 = Ddivd • Dadd • BigD • Base

Product i3 (jampack)

1

```
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;
   }
```

2

```
   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(); }
}
```

Figure 2. i3/calc.jak

The Compilation Process

1. Compose a product
   - *composer –target=i3 Base BigI Idiv Iadd*
   - A .JAK file is created (calc.jak)
2. Translate the Jak code to Java
   - *jak2java *.jak* or
   - *jak2java calc.jak*
3. Compile the Java code
   - *javac *.java*

Composers

Jampack vs. Mixin

- AHEAD has two ways of composing
  - jampack: results in a single class with all refinement code inline
  - mixin: each refinement results in a class
- By using mixin, each class is annotated with the SoUrCe keyword
  - This annotation indicates the feature associated with that class

Example of Product (mixin)

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

Feature **Iadd**

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

Feature **Idiv**

Example: Jampack vs. Mixin

```
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;
   }

      ■ ■ ■
```

Jampack

Mixin

```
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;
    }
                          ■ ■ ■
```

Mixin Composed Code

- All classes derived from refinements are abstract
  - Except the last composed feature
- A suffix is added to the name of each class indicating its feature
  - The goal is to keep unique identifiers

Why mixin?

- It is usually better to use mixin (instead of jampack) because you can trace back to source code
  - After you use jak2java, the generated code might have errors
  - Jampack means all code in a single class. Then, you do not know the feature with error
  - Mixin allows you to trace the error to the feature code

Unmixin

- Using mixin you can also reverse you generated code back to the source
  - This mechanism is called unmixin
- For instance, propagate changes in the Java code back to the Jak file
  - After compose your code (mixin to Java), you can change the composed code
  - Then, unmixin propagate your changes

Bibliography

- Don Batory. **A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite**. International Conference on Generative and Transformational Techniques in Software Engineering (GTTSE), 2005.