Feature Modularity in Software Product-Lines

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu
www.cs.utexas.edu/users/dsb/

Copyright is held by the author/owner(s).
Presented at: Lipari School for Advances in Software Engineering
July 8 - July 21, 2007, Lipari Island, Italy
Feature Modularity in Software Product-Lines

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu
www.cs.utexas.edu/users/dsb/

Introduction

• A product-line is a family of similar systems
  - Chrysler mini-vans, Motorola radios, software
• Motivation: economics
  - amortize cost of building variants of program
  - design for family of systems
• Key idea of product-lines
  - members are differentiated by features
  - feature is product characteristic that customers feel is important in describing and distinguishing members within a family
  - feature is increment in product functionality

Very Rich Technical Area...

• Integrates many subjects:
  - compilers • grammars • artificial intelligence
  - databases • algebra
  - category theory • programming languages
  - compositional programming • OO software design
  - metaprogramming • domain-specific languages
  - declarative languages • tensors
  - generative programming • model driven design
  - verification • collaborations
  - refactoring • automatic programming
  - aspect-oriented programming

• Feature Oriented Programming (FOP) is the study of feature modularity in product-lines
  - features are first-class entities in design
  - often implemented by collaborations

• History of applications
  - 1986 database systems
  - 1989 network protocols
  - 1993 data structures
  - 1994 avionics
  - 1997 extensible compilers
  - 1998 radio ergonomics
  - 2000 prog. verification tools
  - 2002 fire support simulator
  - 2003 AHEAD tool suite
  - 2004 robotics controllers
  - 2006 peer-to-peer networks)

Copyright is held by the author/owner(s).
Presented at: Lipari School for Advances in Software Engineering
July 8 - July 21, 2007, Lipari Island, Italy
Overall Goal

• Place automation of large-scale software design and construction on a practical and firm mathematical foundation

• Feature orientation allows us to do this in a simple way

• Tutorial shows how...

Tutorial Overview

• Lecture 1: Introduction to FOP

• Lecture 2a: Tool Demos
• Lecture 2b: Verification of Feature Compositions

• Lecture 3: Program Refactoring, Synthesis, and Model-Driven Design

• Lecture 4: Feature Interactions and Program Cubes

Motivation

• Software products are:
  • increasing in complexity
  • increasing in costs to develop and maintain
  • decreasing in ability to understand

• Goal of SE is to manage and control complexity
  • structured programming to
  • object-oriented programming to
  • component-based programming to...

  • today's design techniques are too low-level, expose too much detail to make application's design, construction and modification simple

• Something is missing...
  • future design techniques generalize today's techniques
  • tutorial to expose a bigger universe
Keys to the Future

• New paradigms will likely embrace:
  – Generative Programming (GP)
    – want software development to be automated
  – Domain-Specific Languages (DSLs)
    – not Java & C#, but high-level notations
  – Automatic Programming (AP)
    – declarative specs → efficient programs

• Need simultaneous advance in all three fronts to make a significant change

Not Wishful Thinking...

• Example of this futuristic paradigm realized 30 years ago
  • around time when many AI researchers gave up on automatic programming

Relational Query Optimization

• Declarative query is mapped to an expression
• Each expression represents a unique program
• Expression is optimized using rewrite rules
• Efficient program generated from expression

Relational Query Optimization

Keys to Success

• Automated development of query evaluation programs
  • hard-to-write, hard-to-optimize, hard-to-maintain
  • revolutionized and simplified database usage

• Used algebra to specify and optimize query evaluation programs

• Identified fundamental operations of a domain
  • relational algebra

• Represented program designs as expressions
  • compositions of relational operations

• Defined algebraic identities among operations to optimize expressions

• Compositionality is hallmark of great engineering models
Looking Back and Ahead

- Query optimization (and concurrency control) helped bring DBMSs out of the stone age
- Holy Grail Software Engineering: **Repeat this success in other domains**
- Not obvious how to do so...
- Subject of this tutorial...
  - series of simple ideas that generalize notions of modularity and lay groundwork for practical compositional programming and an algebra-based science for software design

Towards a Science of Software Design

What motivates FOP and how is it defined?

Today’s View of Software

- Today’s models of software are too low level
  - expose classes, methods, objects as focal point of discourse in software design and implementation
  - difficult (impossible) to
    - reason about construction of applications from components
    - produce software automatically from high-level specifications (distance is too great)
- We need a more abstract way to specify and reason about systems

A Thought Experiment...

- Look at how people describe programs now...
  - don’t say which DLLs are used...
- Instead, say what **features** a program offers its clients
  - Program1 = feature_X + feature_Y + feature_Z
  - Program2 = feature_X + feature_Q + feature_R
  - why? because features align better with requirements
- We should specify systems as compositions of features
  - few do this for software (now)
  - done in lots of other areas
Methodology for Construction

• What methodology builds systems by progressively adding details?

  **Step-Wise Refinement**
  • Dijkstra, Wirth early 1970s
  • abandoned in early 1980s as it didn’t scale...
  • had to compose hundreds or thousands of transforms (rewrites) to produce admittedly small programs
  • recent work shows how SWR scales
    – scale individual transform to a feature
    – composing a few refinements yields an entire system

What is a Feature?

• **Feature**
  • an elaboration or augmentation of an entity(s) that introduces a new service, capability, or relationship
  • increment in functionality

• **Characteristics**
  • abstract, mathematical concept
  • reusable
  • interchangeable
  • (largely) defined independently of each other

• Illustrate in next few slides
Features are Interchangeable

Features are Reusable

Features are Functions!

PersonPhoto beanie(PersonPhoto x)

PersonPhoto uncleSam(PersonPhoto x)

PersonPhoto mustache(PersonPhoto x)

PersonPhoto lincolnBeard(PersonPhoto x)

Composing Features

- Feature composition = function composition

= lincolnBeard( uncleSam( ))
Large Scale Features

- Called **Collaborations (1992)**
  - simultaneously modify multiple objects/entities
  - refinement of single entity is called **role**

- Example: Positions in US Government
  - each defines a role

Composing Collaborations

- At election-time, collaboration remains constant, but objects that are refined are different

Example of dynamic composition of collaborations

Other Collaborations

- Parent-Child collaboration

- Professor-Student collaboration
Same Holds for Software!

Highly complex entities and relationships in software can be synthesized by composing generic & reusable features

Feature Oriented Programming

• Feature Oriented Programming (FOP) is the study of feature modularity and programming models for product-lines
  • a powerful form of FOP based on step-wise development
  • advocates complex programs constructed from simple programs by incrementally adding features

• How are features and their compositions modeled?

The Theory

GenVoca and AHEAD

A Clue...

• Consider any Java class C
  • member could be a data field or method
  • class C below has 4 members m1–m4

```java
class C {
    member m1;
    member m2;
    member m3;
    member m4;
}
```
**Have You Ever Noticed…**

- Contents of C can be distributed across an inheritance hierarchy

```java
class C1 {
    member m1;
}
class C23 extends C1 {
    member m2;
    member m3;
}
class C4 extends C23 {
    member m4;
}
class C extends C4 {}
```

**Another Example…**

- C23 decomposed further as:

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

**Observe…**

- Significance: class definition need not be monolithic, but can be built by incrementally composing reusable pieces via inheritance

- Nothing special about the placement of members m1...m4 in this hierarchy except...
  - **no-forward references**: member can be introduced as long as all members it references are defined
  - requirement for compilation, step-wise development

**Look Familiar?? Remember Algebra??**

- Consider sets and union operation (U)
  - commutative almost like inheritance...

- Vector addition (+)
  - is commutative almost like inheritance

- C1 = { m1 }
- C2 = { m2 }
- C3 = { m3 }
- C4 = { m4 }

- C = C1 \cup C2 \cup C3 \cup C4 = { m1, m2, m3, m4 }
- C = C1 + C2 + C3 + C4 = { m1, m2, m3, m4 }
A Closer Analogy

• Vector join (→)
  • Vector join lays vectors end-to-end to define a path
  • Not commutative! – Order of composition matters!

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

path followed by
A → B is different
than B → A;
end point is the same

Operation We Want...

• Is not quite inheritance...
  • want to add new methods, new fields, and refine existing
    methods like inheritance
  • also want constructors to be inherited and refined as well,
    (inheritance doesn’t provide this)

class C1 {
    constructor1
}

class C2 {
    constructor2
}

class C12 {
    constructor1
    constructor2
}

The operation • we want is called class refinement

Syntax of Class Refinement

• Suppose program P has single class B
• Composition of R with P defines a new program N:

class B { int x; }

class B {
    int x;
    int y;
    void z(){...}
}

refines class B {
    int y;
    void z(){...}
}

Algebraic Formulation

• Base programs are constants
// constant P
class B { int x; }

• Composition is an expression
N = R( P )
= R • P

• yields:

class B {
    int x;
    int y;
    void z(){...}
}

• Refinements are functions
// function R
refines class B {
    int y;
    void z(){...}
}

Treat programs as values
is metaprogramming
Another Example

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

• Composition is an expression or named expression
  
  C = C4( C3( C2( C1 ) ) )
  = C4 ● C3 ● C2 ● C1

  Note: both notations are equivalent
```

Method Refinement ala Inheritance

```cpp
result = method_refinement

// base_method
void foo() {

  super.foo();
  /* after stuff */
}

// method_refinement
void foo() {
  /* before stuff */
  /* do something */
  /* after stuff */
}

is substitution or an equivalent encoding
```

Connecting the Dots...

• Scalability
  • refinement is not limited to a single class
  • collaborations modularize refinements of multiple classes and add new classes
    » adding new classes that can be refined is critical

• A collaboration has meaning when it implements a feature
  • ever add a new feature to an existing OO program?
    • several existing classes may be refined
    • several new classes may be added
Synthesis Paradigm

Program P = featureX • featureY • featureZ

Note: each feature updates multiple classes

class1 class2 class3 class4

By composing features, packages of fully-formed classes are synthesized

Contributors to this View…

- Many researchers have variants of this idea:
  - refinements – Dijkstra, Wirth 68
  - layers – Dijkstra 68, Batory 84
  - product-line architectures – Kang 90, Gomaa 92…
  - collaborations – Reenskaug 92, Lieberherr 95, Mezini 03
  - program verification – Boerger 96
  - aspects – Kiczales 97, et al.
  - concerns – Ossher-Harrison-Tarr 99

Connecting the Dots…

- You can always decompose software in this manner
  - trick is that your refinements are reusable
  - that’s the connection with features, product-lines
  - features are reusable – so too must be their implementations

Design is the Key

- software that is not designed to be reusable, composable, etc. with other software won’t be – this is co-design or designing to a standard
- Architectural Mismatch (ICSE 1995)

Product-Line Design – feature implementations are designed with compositionality, reusability in mind

GenVoca

Genesis + Avoca

The First Generation
GenVoca (1988, 1992)

- Equates constants, functions with features

- Constants:
  - f – base program with feature f
  - h – base program with feature h

- Functions
  - i • x – adds feature i to program x
  - j • x – adds feature j to program x

A domain model or product-line model or GenVoca model M

- set of constants (base programs)
- functions (program refinements)

\[ M = \{ f, h, \ldots, i, j, \ldots \} \]

Function Composition

- Multi-featured applications are expressions

\[ \text{app1} = i \bullet f \quad \text{– application with features f and i} \]
\[ \text{app2} = j \bullet h \quad \text{– application with features h and j} \]
\[ \text{app3} = i \bullet j \bullet f \quad \text{– your turn...} \]

Given a GenVoca model, we can create a family of applications by composing features

Expression Optimization

- Constants, functions represent both a feature and its implementation
  - different functions can be different implementations of the same feature

\[ k_1 \bullet x \quad \text{// adds k with implementation #1 to x} \]
\[ k_2 \bullet x \quad \text{// adds k with implementation #2 to x} \]

- When application requires feature k, it is a matter of optimization to determine the best implementation of k
  - counterpart of relational optimization
  - more complicated rewrites possible too...

- See:

Generalization of Relational Algebra

- Keys to success of Relational Optimizers
  - expression representations of program designs
  - rewrite expressions using algebraic identities

- Here’s the generalization:

  - domain model is an algebra for a domain or product-line
    - is set of operations (constants, functions) that represent stereotypical building blocks of programs/members
    - compositions define 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 expression representations of programs, like relational algebra
Scaling Program Generation

- Generating code for an individual program is OK, but not sufficient
- Today’s systems are not individual programs, but groups of collaborating programs
  - client-server systems, tool suites (IDEs)
- Further, systems are not solely defined by code
  - architects routinely use many knowledge representations
    - formal models, UML models, makefiles, documents, ...
- Need 4 insights to capture these ideas

Insight #1: Platonic Forms and Languages

- Each program representation captures different information in different languages
- We want all these representations in a single module

Insight #2: Generalize Features

- When a program is refined, any or all of its representations may be updated
- Ex: Add a new feature F to program P changes:
  - code (to implement F)
  - documentation (to document F)
  - makefiles (to build F)
  - formal properties (to characterize F)
  - performance properties (to profile F)
  - ...
- This is a collaboration
Vectors and Vector Refinements

- A program is a vector of representations
- Features refine vectors component-wise

\[
\begin{array}{c}
\text{P0} & \text{F} & \text{P1} & \text{G} & \text{P2} \\
\text{code}_0 & \text{binary}_0 & \text{make}_0 & \text{doc}_0 \\
\text{binary}_1 & \text{make}_1 & \text{doc}_1 \\
\text{code}_2 & \text{binary}_2 & \text{make}_2 & \text{doc}_2 \\
\end{array}
\]

Vector Representations

- We are reducing program synthesis to vector composition
- GenVoca model
  - constant P0
  - function F
- Feature composition = vector composition
- Still need another idea

Insight #3: Generalize Modularity

- A module is a containment hierarchy of related artifacts

Modularization of Multiple Programs

- Generalize module hierarchies to arbitrary depth, contents

Don Batory
UT-Austin Computer Sciences
intro 61

Don Batory
UT-Austin Computer Sciences
intro 62

Don Batory
UT-Austin Computer Sciences
intro 63

Don Batory
UT-Austin Computer Sciences
intro 64
## Modules are Nested Vectors

- Program as vector idea recurses: each subrepresentation can itself be a vector

![Diagram of nested vectors]

- Module is a (nested) vector
- Name of a subrepresentation is unique; it defines its index position in a vector

Don Batory
UT-Austin Computer Sciences
intro 65

## Law of Composition

- Consider base program $P$ and refinement $R$:

\[
P = [ A_p, B_p, C_p, ]
\]

\[
R = [ A_R, C_R, D_R ]
\]

- implicit vector padding with blanks
- base programs have nulls ($\emptyset$)
- refinements have identity functions (i)

- What is $R \cdot P$?

Don Batory
UT-Austin Computer Sciences
intro 66

## Law of Composition

- $R \cdot P$ is:

\[
P = [ A_p, B_p, C_p, ]
\]

\[
R = [ A_R, C_R, D_R ]
\]

\[
R \cdot P = [ A_R \cdot A_p, B_p, C_R \cdot C_p, D_R ]
\]

- Says how composition distributes over modularization

- Do you recognize this law?

Don Batory
UT-Austin Computer Sciences
intro 67

## Inheritance!

### class representation

```java
class P {
  member A_p;
  member B_p;
  member C_p;
}
```

### vector representation

\[
P = [ A_p, B_p, C_p, ]
\]

```java
class R extends P {
  member A_R;
  member C_R;
  member D_R;
}
```

\[
R = [ A_R, C_R, D_R ]
\]

```java
class R extends P {} 
```

\[
R \cdot P = [ A_R \cdot A_p, B_p, C_R \cdot C_p, D_R ]
\]

Don Batory
UT-Austin Computer Sciences
intro 68
Simple Implementation

- Module hierarchies = nested vectors

```
A = [ Code, R.drc, Htm ]
Code = [ X.java, Y.java ]
Htm = [ W.htm, Z.htm ]
```

Simple Theory

- Result computed algebraically by **recursively** expanding and applying the law of composition

```
C = B ∙ A

C = [ [ X.java_B, Y.java_B ], [ X.java_A, Y.java_A ], [ R.drc_b, R.drc_a, W.htm_b, Z.htm_a ] ]
```

Note!

- Each expression defines an artifact to be produced
Polymorphism...

- Composition operation • is **polymorphic**
  - law of composition says how vectors are composed
  - different implementation of • for each representation
    » • for code
    » another • for html files, etc.

- But what does refining a non-code artifact mean?
  - what general principle guides refinement?

Example: Makefiles

- Instructions to build parts of a system
  - it is a language for synthesizing programs

- When we synthesize code for a system, we also have to synthesize a makefile for it

- Sounds good, but...
  - what is a refinement of a makefile?????

Makefile

```
mymake

main
- compile A
- compile B
- compile C

common
- compile X
- compile Y
- compile Z

clean
- delete *.class
```

command line> make main

Makefile Refinements

```
mymake

main
- compile A
- compile B
- compile C

common
- compile X
- compile Y
- compile Z
- compile D
- compile E

clean
- delete *.class
- delete *.ser
```

Question: what is a general paradigm for refining non-code artifact types?
Makefiles

```cpp
<project makeMake>
  <target main depends="common">
    <compile A>
      void main {
        ...
      }
      <compile B>
      <compile C>
    </target>
  </target>
  <target common>
    <compile X>
    <compile Y>
    <compile Z>
    </target>
  </target>
</project>
```

Insight #4: Principle of Uniformity

- Treat all artifacts equally, as objects or classes
  - create analog in OO representation

- Refine non-code representations same as code representations

- That is, you can refine any artifact
  - understand it as an object, collection of objects, or classes

- **We are creating a theory of information structure based on features**
  - it works for code and all other representations

Don Batory
UT-Austin Computer Sciences
intro 77

Big Picture

- Most artifacts today (HTML, XML, etc.) have or can have a hierarchical structure
- But there is no refinement relationship among artifacts!
  - what's missing are refinement operations for artifacts
- Need tools to refine instances of each artifact type
  - MS Word?
  - given such tools, scale step-wise refinement scales without bounds...
- Features modularize changes/additions to all representations of a system
  - so all artifacts (code, makefiles, etc.) are updated consistently
- Compositions yield consistent representations of a system
  - exactly what we want
  - simple, elegant theory behind simple implementation

Product Member Synthesis Overview

- declarative DSL • generalizes RQO paradigm
- scales to large systems

- Engineer
  - expression composition and optimization
  - artifacts of specified system
  - generator
  - artifact
  - generator
  - artifact
  - generator
  - ...

Don Batory
UT-Austin Computer Sciences
intro 78

Don Batory
UT-Austin Computer Sciences
intro 80
Recommended Readings


• Lieberherr. Adaptive Object-Oriented Software. PWS publishing, 1995.


• Ossher and Tarr. “Multi-dimensional separation of concerns and the Hyperspace approach”. In Software Architectures and Component Technology (M. Akalt, ed.), 2002


• Smaragdakis, Batory. "Scoping Constructs for Program Generators". GCSE 1999.


The AHEAD Tool Suite

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu
www.cs.utexas.edu/users/dsb/

Copyright is held by the author/owner(s).
Presented at: Lipari School for Advances in Software Engineering
July 8 - July 21, 2007, Lipari Island, Italy

Composer Tool

• Key tool in AHEAD Tool Suite (ATS) is composer
• composer expands AHEAD expression to yield target system

Jak Files

• Program in extended-Java files
  • Jak(arta) files

• Java + feature declarations, etc.
  • Jak is an extensible language

• AHEAD is bootstrapped
  • Most AHEAD tools are written in Jak

Other Tools...

• Besides composer
  • jak2java – translates Jak files to Java files
  • javac – javac compiler
  • reform – Jak or Java file formatter/pretty-printer
  • others...

> composer -target=feat321 feat1 feat2 feat3
> cd <model-directory>
> composer -target=...
> reform *.jak
> jak2java *.jak
> reform *.java
> javac *.java
Jak-File Composition Tools

- composer invokes Jak-specific tools to compose Jak files
  - two tools now: jampack and mixin
  - jak2java translates Jak to Java

jampack

- Flattens “inheritance” hierarchies
  - takes expression as input, produces single file as output
  - basically macro expansion with a twist...

```
class top {
  int a;
  void foo() { ... }
}
```

refines class top {
  int b;
  int bar() { ... }
}

mixin

- Encodes class and its refinements as an inheritance hierarchy

```
abstract class top$$A {
  int a;
  void foo() { ... }
}
```

refines class top {
  int b;
  int bar() { ... }
}

```
public class top extends top$$A {
  int b;
  int bar() { ... }
}
```
unmixin

- Edit, debug composed A.jak files
- *unmixin* propagates changes from composed file to original feature files automatically

```
A.jak        unmixin
            (from feat 1)
```

```
A.jak  unmixin  A.jak  jak2java  A.java
            (from feat 2)         (composed)
```

```
A.jak        translate debug update
            (from feat 3)
```

## Composable Representations

- Current list...
  - *.jak* – extended Java files (Jakarta)
    - class
    - interface
    - state machine (ex: embedded DSL)
  - *.equation* – named expression files
  - *.b* – grammar files
  - *.drc* – design rule files
  - others...

### AHEAD tools
- AHEAD tools are written in extended Java.
- AHEAD has been bootstrapped so that its tools have been written using AHEAD tools.

---

## Cultural Enrichment

- Note algebraic underpinning...

```
P = \text{javac}( \text{jak2java}( f3 \cdot f2 \cdot f1 ) )
```

- Same paradigm as AHEAD
  - progressively elaborating a containment hierarchy
  - can optimize expression (not this one...)
Cultural Enrichment

- To see connection, watch how module hierarchy is transformed...
  - adding new artifacts is example of module refinement

• Big picture: lots of operations on AHEAD modules
  - seems that lots of optimizations are possible too...

Don Batory
UT-Austin Computer Sciences

A Simple Example

to illustrate concepts, tools

Domain of Graph Applications

- A grammar is a simple way to express family of related applications
  - tokens are features
  - sentences are feature compositions

Example Family Members

- cycle checking
- vertex numbering
- connected regions

Don Batory
UT-Austin Computer Sciences
It is Easy to...

Imagine a GUI tool that allows you to specify any possible combination

- declarative language
- tool generates an explanation of your specification
- and identifies errors (and suggests corrections) when combinations of features are not possible

That's Easy...

So too is creating the underlying FOP model:

\[
G_{pl} = \{ \\
\text{DIRECTED} \quad - \quad \text{directed graphs} \\
\text{UNDIRECTED} \quad - \quad \text{undirected graphs} \\
\text{BFS} \quad - \quad \text{breadth first search} \\
\text{DFS} \quad - \quad \text{depth first search} \\
\text{CYCLE} \quad - \quad \text{cycle checking} \\
\text{NUMBER} \quad - \quad \text{vertex numbering} \\
\text{REGIONS} \quad - \quad \text{connected regions} \\
\ldots \\
\} \\
\]

Constructing Applications

Constructing Applications

Constructing Applications

Recommended Readings

Introduction

- Fundamental problem: not all compositions of features are correct
  - but code can still be generated!
  - and maybe code will still compile!
  - and maybe code will run for a while!
  - impossible for users to figure out what went wrong!

Tool Demo

- Illustrate on Graph Product Line
  - has been applied to much larger examples

- Declarative domain-specific language
  - counterpart to Dell web page

- Constraints propagated as selections are made
  - cannot specify incorrect design

- Can debug model specifications
  - by verifying known properties of feature combinations
Feature Diagrams and Grammars
(The Theory Behind The Tool)

Grammar  \( \rightarrow \)  Feature Diagram

Feature Diagrams

- **Feature diagrams** are standard product-line notations
  - declarative way to specify products by selecting features

- FDs are trees:
  - leaves are primitive features
  - internal nodes are compound features
  - parent-child are containment relationships

How To Read Feature Diagrams

- Mandatory – features that are required
- Optional – features that are optional
- And – all subfeatures (children) are selected
- Alternative – only 1 subfeature can be selected
- Or – 1+ or 0+ subfeatures can be selected
Another Example

- What is a legal product specification?
  - E is ?
  - R is ?
  - S is ?

- Sound familiar?
  - de Jonge and Visse 2002: FDs are graphical representations of grammars
  - “GenVoca Grammars” 1992: grammar defines legal orders in which features can be composed

Recall GPL Model

\[ Gpl = \{ \]
\[ \text{DIRECTED} \quad \text{UNDIRECTED} \quad \text{BFS} \quad \text{DFS} \quad \text{CYCLE} \quad \text{NUMBER} \quad \text{STRONGC} \]
\[ \} \]

GPL Grammar

\[ Gpl : \text{Alg}^+ \ [\text{Src}] \ \text{Wgt} \ \text{Gtp}; \]
\[ \text{Gtp} : \text{DIRECTED} \ | \ \text{UNDIRECTED}; \]
\[ \text{Wgt} : \text{WEIGHTED} \ | \ \text{UNWEIGHTED}; \]
\[ \text{Src} : \text{DFS} \ | \ \text{BFS}; \]
\[ \text{Alg} : \text{NUMBER} \ | \ \text{CONNECTED} \ | \ \text{STRONGC} \]
\[ \ | \ \text{CYCLE} \ | \ \text{MSTPRIM} \ | \ \text{MSTKRUSKAL} \ | \ \text{SHORTEST}; \]

A sentence of this grammar defines a composition of features

\[ \text{Prog} = \text{NUMBER} \bullet \text{CYCLE} \bullet \text{BFS} \bullet \text{UNWEIGHTED} \bullet \text{DIRECTED} \]

Mapping of FDs to Grammars

Diagram

Grammar

\[ S : e1 \ [e2] \ en; \]

\[ \ldots S \ldots \]

\[ S : e1 \ | \ e2 \ | \ en; \]

\[ \ldots S+ \ldots \]

\[ S : e1 \ | \ e2 \ | \ en; \]
Example: Convert FD to Grammar

E : R S ;
R : g | h | i ;
S : a [ b ] c ;

- Application defined by Feature Model = sentence of grammar E
- Resulting grammar is a GenVoca grammar (1992)

Grammars and Propositional Formulas

• Set of boolean variables and a propositional logic predicate that constrains values of these variables
• Standard $\neg, \lor, \land, \Rightarrow, \Leftrightarrow$ operations
• Nonstandard:
  - atmost1(e₁...eₖ) – at most one eᵢ is true
• Insight: A grammar is a compact representation of a propositional formula

Mapping Productions to Formulas

• Given production: R : P₁ | ... | Pₙ ;
• R can be referenced in two ways:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>... R+ ...</td>
<td>R $\Leftrightarrow$ P₁ $\lor$ P₂ $\lor$ ... $\lor$ Pₙ</td>
</tr>
<tr>
<td>(choose 1 or more)</td>
<td></td>
</tr>
<tr>
<td>... R ...</td>
<td>R $\Leftrightarrow$ (P₁ $\lor$ P₂ $\lor$ ... $\lor$ Pₙ) $\land$ atmost1(P₁,P₂, ..., Pₙ)</td>
</tr>
<tr>
<td>(choose 1)</td>
<td></td>
</tr>
</tbody>
</table>
Mapping Patterns to Formulas

- T1 T2 ... Tn :: P
  
  formula: \( P \iff T_1 \land P \iff T_2 \land ... \land P \iff T_n \)

- T1 [T2] ... Tn :: Q
  
  formula: \( Q \iff T_1 \land T_2 \land ... \land Q \iff T_n \)

Example: Grammars to Formulas

- Convert each production, pattern to formula
- Take conjunction of all formulas
- Conjoin root of grammar

\[
E: RS; \\
R: g | h | i; \\
S: a [ b ] c; \\
E \iff R \land E \iff S \\
<grammar>
\]

A sentence of E satisfies the propositional formula and vice versa

Example of an additional formula:

- Ex: if features i and b are incompatible

\[
i \lor b \Rightarrow \neg (b \land i)
\]

Last Example

Recap

- We can map any AHEAD model or Feature Diagram to a propositional formula
- But what about constraints?

- Any additional, arbitrary propositional formulas conjoined onto grammar formula
  - Ex: if features i and b are incompatible, add the formula

\[
i \lor b \Rightarrow \neg (b \land i)
\]
### Example: Additional Constraints in GPL

- **Straight from Graph Algorithm Text**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Required Graph Type</th>
<th>Required Weight</th>
<th>Required Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Numbering</td>
<td>Any</td>
<td>Any</td>
<td>BFS, DFS</td>
</tr>
<tr>
<td>Connected Components</td>
<td>UNDIRECTED</td>
<td>Any</td>
<td>BFS, DFS</td>
</tr>
<tr>
<td>Strongly Connected Components</td>
<td>DIRECTED</td>
<td>Any</td>
<td>DFS</td>
</tr>
<tr>
<td>Cycle Checking</td>
<td>Any</td>
<td>Any</td>
<td>DFS</td>
</tr>
<tr>
<td>Minimum Spanning Tree</td>
<td>UNDIRECTED</td>
<td>WEIGHTED</td>
<td>None</td>
</tr>
<tr>
<td>Single-Source Shortest Path</td>
<td>DIRECTED</td>
<td>WEIGHTED</td>
<td>None</td>
</tr>
</tbody>
</table>

### GPL Model Specification

**Grammar**

- `Gtp : DIRECTED | UNDIRECTED ;`
- `Wgt : WEIGHTED | UNWEIGHTED ;`
- `Src : DFS | BFS ;`
- `Alg : NUMBER | CONNECTED | STRONGC | CYCLE | MSTPRIM | MSTKRUSKAL | SHORTEST ;`

**Constraints**

- `% NUMBER implies Gtp and Src; CONNECTED implies UNDIRECTED and Src; CYCLE implies Gtp and DFS; SHORTEST implies DIRECTED and WEIGHTED; STRONGC implies DIRECTED and DFS; MSTKRUSKAL or MSTPRIM implies UNDIRECTED and WEIGHTED;`

### Recap

- An AHEAD Model is a propositional formula!
  - primitive features and compound features are variables
- Grammar:
  - specifies order in which features are composed
  - ordering very important for AHEAD
- Additional propositional constraints:
  - weed out incompatible feature combinations

### Declarative Domain-Specific Languages

- GenVoca Grammar
- Feature Diagram
- Propositional Formula
- DDSLs
Declarative Languages

- Features enable declarative program specifications
  - that’s what feature diagrams are for!
  - counterpart of SQL, Dell web pages

- Want a declarative GUI DSL that acts like a syntax-directed editor
  - user selects desired features
  - tool precludes specifying incorrect programs

Constraint Propagation

- 1980’s result from Artificial Intelligence

  - Logic Truth Maintenance System
    - boolean constraint propagation (BCP) algorithm
    - takes a boolean predicate, set of variable assignments as input, deduces other variable assignments as output
    - very simple, efficient algorithm


  - BDDs (Binary Decision Diagrams) are also popular

Debugging Feature Models

- We know features A and B are compatible
  - let $P_{model}$ be the predicate of our feature model
  - $P_{model} \land A \land B$ must be satisfiable
    - that is, is there a product that has both A and B?

  - Satisfiability (SAT) Solver
    - off-the-shelf tool that automatically determines if a boolean predicate is satisfiable
    - very efficient

  - Basis for feature model debugging
    - provide a script of compatible, incompatible features and verify that our feature model has these properties
    - solver confirms known properties of a model
Experience

• Has worked well...
• Use off the shelf constraint solvers
• Predicates are simple
• Reason: architects think in terms of features
  • if predicates were really complicated
    – architects couldn’t design
    – people couldn’t program
    – because it would be too difficult
• We are making explicit what is implicit now...

There’s More...

• Benavides noticed you could add numerical attributes to grammar

There’s More... and is Very Exciting!

• Allow features to have additional parameters
  • property lists
• Generalize predicates to include constraints on numeric variables
  • select product that maximizes/minimizes criteria
    (performance!)
  • restrict products based on performance requirements, criteria
  • use standard Constraint Satisfaction Problem (CSP) Solvers

Future

• Basic result: software design is a satisfiability problem
  • does there exist a system that satisfies the following set of constraints?
• Research: to find optimal system configurations automatically
  • true automatic programming!
  • counterpart to relational query optimizers
### Recommended Readings

Program Refactoring, Program Synthesis, and Model Driven Design

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu
www.cs.utexas.edu/users/dsb/

Copyright is held by the author/owner(s).
Presented at: Lipari School for Advances in Software Engineering
July 8 - July 21, 2007, Lipari Island, Italy

This Lecture

• Sketch where I see
  automated software design & maintenance is headed

• Essential complexity of software structure
  is exposed when program construction and design is viewed as a computation

• Architectural Metaprogramming
  programs are values
  transformations map programs to programs
  operators map transformations to transformations

Architectural Metaprogramming

• Lies at core of many important areas in software design and maintenance:
  refactorings are behavior-preserving transformations
  feature-based and aspect-based software synthesis use behavior-extending transformations
  model driven design uses both to map platform independent models (PIMs) to platform specific models (PSMs)

• Lecture reveals a bigger world in which FOP lies

Relationship of Design to Set Arithmetic

• Is basic to engineering

• Computer Aided Design (CAD) tools enable engineers to express designs by adding, subtracting, and transforming volumes from which properties of designs are derived

• Architectural metaprogramming offers a program analog: programs can be added, subtracted, and transformed
  • set arithmetic captures essential design concepts
  • accidental complexities and limitations of languages, tools, and implementations are abstracted away
Upcoming Topics – Four “Mini” Talks

• Basics of Architectural Metaprogramming then reflect on 2006 advances in

• Program Refactoring
  • Danny Dig & Ralph Johnson (Illinois)

• Program Synthesis
  • Roberto Lopez-Herrejon (Oxford) & Christian Lengauer (Passau)

• Model Driven Design
  • Salva Trujillo & Oscar Diaz (Basque Country)

• All topics describe systems that have been built
  • step back and give a simple explanation of their results

Architectural Metaprogramming

• Programs are values

• Here is a value
  (Java definition of class C):

```java
class C {
  int x;
  void inc() {...}
  ...
}
```

• Here is another value:

```java
class D {
  void compute()
  {...}
  ...
}
```

• Let D = and C =

```java
class D {
  void compute()
  {...}
}
class C {
  int x;
  void inc() {...}
}
```

• D + C =

```java
class D {
  void compute()
  {...}
}
class C {
  int x;
  void inc() {...}
}
```
Another Example

1. Let \( C_1 = \) and \( C_2 = \)

\[
\text{class } C_1 \{
\text{void comp() \{..\}}
\}
\]

\[
\text{class } C_2 \{
\text{int } x;
\text{void inc() \{..\}}
\}
\]

2. \( C_1 + C_2 = \)

\[
\text{class } C \{
\text{void comp() \{..\}}
\}
\]

\[
\text{class } C \{
\text{int } x;
\text{void inc() \{..\}}
\}
\]

+ (Sum) is Disjoint Set Union

1. Has expected properties
   - 0 is identity (null program)
     \[ P = 0 + P = P + 0 \]
   - Commutative (because disjoint set union is commutative)
     \[ A + P = P + A \]
   - Associative (because disjoint set union is associative)
     \[ (A + B) + C = A + (B + C) \]

2nd Operation: – (Sub)

1. Subtraction is set difference
   \[ (D + C) – C = D \]

2. Has expected properties:
   - Left associative
     \[ P – C – D = ((P – C) – D) \]
   - Not commutative
     \[ P – C \neq C – P \]
   - Identity
     \[ P – 0 = P \]

3rd Operation: Distributive Transformations

1. Transformation is a function that maps programs to other programs

2. Rename\((p,q,r)\) – in program “p” replace name “q” with “r”

   \[
   \text{class } C \{
   \text{int } x;
   \text{void inc() \{..\}}
   \}
   \]

   Rename\((x, C.x, C.z)\) =

   \[
   \text{class } C \{
   \text{int } z;
   \text{void inc() \{..\}}
   \}
   \]
Another Example

```java
class D {
  void compute(
    ..
  )
}
```

```java
class D {
  void compute(
    ..
  )
}
```

**Rename(                              , C.x, C.z) =

- Called a **fixed point**:
  - a value x such that f(x) = x

- Distributive transformations have lots of fixed points

A Key Property of Distributive Transformations

- Transformations we consider **distribute** over + and –

  \[ f( A + B ) = f(A) + f(B) \]

  \[ f( C - D ) = f(C) - f(D) \]

  Here’s an example...

Example of Distributivity

```java
class D {
  void compute() {
    ..
  }
}
```

```java
class D {
  void compute() {
    ..
  }
}
```

**Rename(                              , C.x, C.z) =

**Structures & Properties**

- **Structure** – what are the parts and how are they connected?

  - a solid bounded by six equal squares, the angle between any two adjacent faces is a right angle.

- **Properties** of structure – attributes derivable from structure

  - surface area = \( 6E^2 \); where E is edge length
  - volume = \( E^3 \)
Software Analogs

- **Structure** of a program is a meta expression

  \[ P = f_1(C + D + f_2(E + F)) \]

- Properties of a program – is derived from program structure

  In this lecture, I focus on program structure.

  Results on properties are presented elsewhere.

Refactoring

- **Is** a program transformation that changes the structure of a program, but not its behavior
  - rename methods
  - move method from subclass to superclass
  - ...

- Most design patterns are end-products of refactorings

- Common IDEs (Eclipse, Visual Studio, IntelliJ) have refactoring tools or plug-ins

- Here’s an interesting refactoring problem

Evolution of APIs

- **Use** of components (e.g. frameworks, libraries) are common in software development
  - build systems faster and cheaper

- **Application Program Interface (API)** of a component – set of (Java) interfaces and classes that are exported to application developers
  - ideally, APIs don’t change, but of course they do!
  - when APIs change, client code must also change
  - very disruptive event in program development

- Need an easy and safe way to update applications when component’s API changes
A Common API Change

- **Move Method**
  - Note: although component code changes, client code must also change
  - But a component developer doesn’t have the client code

```
class host {
    static X m(...home f)
    ...
}
```

```
class home {
    X m(..) { ... }
    void b() { host.m(..,f) }
}
```

Component

```
void b() { f.m(); }
```

Client Code

This Change is a Meta-Expression

\[ P_{new} = \rho \cdot \mu \left( P_{old} \right) \]

```
class host {
    static X m(...home f)
    ...
}
```

```
class bar {
    void y() {
        host.m(...f)
    }
}
```

Component

```
void b() { f.m(); }
```

Client Code

Other Common API Changes

- **Move Field**
- **Delete Method**
  - usually done after a method is renamed or moved
- **Change Argument Type**
  - ex: replace argument type with its supertype
- **Replace Method Call**
  - with another that is semantically equivalent and in the same class
- **Lots of others...**
  - preliminary work suggests all can be written as meta expressions

Result

- Dig & Johnson paper:
  - “How do APIs Evolve: A Story of Refactoring”
    Jour. Software Maintenance & Evolution: Research & Practice 2006
  - Manually analyzed change logs, documentation, etc. of different versions of 5 medium to large systems (50K to 2M LOC)
    - Eclipse, Struts, JHotDraw...
  - Found over 80% of API changes are refactorings
    - means LOTS of tedious & error-prone updates can be automated
  - explain elegance of their solution using architectural metaprogramming
In the Future

- Programmers will use advanced IDEs that “mark” API classes, methods, fields
  - only way marked elements can change is by refactorings ($\beta$)
  - “private” component edits modeled by transformations ($e$)

\[ \beta \cdot e \cdot e \cdot \beta = \text{version 1} \]

\[ \beta = \text{transformations to be applied to update client code w.r.t. changes in API} \]

- API updates $\beta$ is a projection of changes where “private” edits are removed

In the Future

- IDEs will be component evolution calculators

- IDEs will create update functions like $U$ for distribution
  - distribute meta-functions, not components

\[ U(x) \]

- IDEs will apply functions to code bases to automatically update them

- Architectural metaprogramming is at the core of this technology

Client Update Meta-Function $U$

\[ U \left( \begin{array}{c}
\text{client program} \\
\text{version 0}
\end{array} \right) = \beta \left( \begin{array}{c}
\text{client program} \\
\text{version 0}
\end{array} \right) - \text{version 0} + \text{version 1} \]

\[ U \left( \begin{array}{c}
\text{client program} \\
\text{version 1}
\end{array} \right) = \beta \left( \begin{array}{c}
\text{client code} \\
\text{version 0}
\end{array} \right) + \text{version 1} \]

This is not how result is presented by Dig and Johnson; it is an architectural metaprogramming expression of their results.

#3: Advances in Program Synthesis

- Architectural metaprogramming is at the core of this technology
Background

- Previous lectures have presented basic ideas on feature modularity and product lines
- But now, let's look inside the structure of features and see how it is related to **aspect-oriented programming (AOP)**
  - find similarities and differences between aspects and features

What Are FOP Features?

- If we peer inside features we see familiar ideas popularized by AOP
  - here I use ideas of AOP
- **Introduction** – adds new members to existing classes
  - corresponds to metaprogramming addition
- **Advice** – modifies methods at particular points, called **join points**
  - quantification means advise all parts of a program – distributivity!
  - advice is a distributive transformation
  - advice is behavior-extending not behavior-preserving
- No “subtraction” in AOP or in FOP

Introduction

- Incrementally add new members, classes

```
Program P

class C {
    void foo() { .. }
    int i;
    String b;
}
class D {
    String bar;
    int cnt() { .. }
}
```

Meta-Algebra Interpretation

```
P = C.b + C.foo + C.i + D.bar + D.cnt
```
Advice

- Defined in terms of events called **join points**
  - when method is called
  - when method is executed
  - when a field is updated
  - ...

- **Advice**: when particular join points occur, execute a given piece of code

- Although advice has a “dynamic” interpretation, we can give it a “static” metaprogramming interpretation

Advice Example

```java
Program P

class C {
    int i,j;
    void setI (int x){ i=x; }
    void setJ (int x){ j=x; }
}

after(): execution (void C.set*(..))
    { print("hi"); }

```

Meta-Algebra Interpretation

```java
Program P

class C {
    int i,j;
    void setI (int x){ i=x; }
    void setJ (int x){ j=x; }
}

after(): execution (void C.set*(..))
    { print("hi"); }

```

```
P = C.i + C.j + C.setI' + C.setJ'
```

Structure of Features

- Features are metaprogramming functions that:
  - advise (a) an existing program (x)
  - introduce new terms (i)

  \[ F(x) = i_f + a_f(x) \]

- Composition:

  \[ G(F(B)) = i_g + a_g(i_f + a_f(b)) \]
In the Future

- Many (narrow) domains will be well-understood
  - know problems, solutions
- Complexity controlled by standardization
  - programs specified declaratively using "standard" features (like Dell)
- Compilers will be program calculators
  - inhale source code
  - generate meta-expression, maybe optimize expression
  - evaluate to synthesize program
- Architectural metaprogramming is at core of these technologies

Big Picture

- Refactorings and advice are both transformations
- Suppose I have a refactoring and advice to apply to a program. What does it mean to compose them?
- Advice does not modify a refactoring
  - a refactoring is not a language construct; there are no join points in a refactoring
- Refactoring can modify programs that include advice

Example

Program P

```java
class C {
    int i,j;
    void SETI (int x){ i=x; }
    void SETJ (int x){ j=x; }
}

after(): execution (void C.set*(..))
print("hi");
```

 naam

- change method names
- change advice declaration
Meta-Algebra

- Remember differential operators in calculus?
  - they transform expressions
    \[ \frac{\partial (a+b+c)}{\partial y} = \frac{\partial a}{\partial y} + \frac{\partial b}{\partial y} + \frac{\partial c}{\partial y} \]
    each term is transformed

- Rename refactoring is similar
  - it transforms each term of a meta expression
    \[ \beta( i + a(x) ) = \beta(i) + \beta(a)( \beta(x) ) \]

Homomorphisms

- Such a mapping is an example of a:
  - structure-preserving map between algebras

- Grounded in **Category Theory**
  - theory of mathematical structures and their relationships
  - more later...

How Meta-Calculation Proceeds

Program P

```java
class C {
    int i,j;
    void SETI (int x){ i=x; }
    void SETJ (int x){ j=x; }
    , C.set*, C.SET*
}

after(): execution (void C.set*(..))
{ print(“hi”); }
```

\[ HI ( C.i + C.j + C.SETI + C.SETJ ) \]

Recap

- Architectural meta-algebra is getting more interesting
  - refactorings are operators on meta expressions that have higher-precedence than advice

- The rewrite rules for a refactoring \( R \) is:
  \[ R( a + b ) = R( a ) + R( b ) \]
  \[ R( a - b ) = R( a ) - R( b ) \]
  \[ R( a \cdot b ) = R( a ) \cdot R( b ) \]
Another Interesting Question

What does AspectJ really do?

Basic Differences of FOP and AspectJ

- Aspects don’t compose
  - to this day, you cannot express all aspect files as a composition of simpler aspect files
  - reason: rules for ordering around, before, after advice are incomprehensible
  - see AspectJ documentation

- Unbounded quantification
  - AspectJ applies advice after all introductions have been made
  - FOP applies advise at different stages of program development

- Why does AspectJ use unbounded quantification?

Tutorial – Method Refinement

- Features refine individual methods by before, around, after advice

Aspects Originate From MetaClasses ~1990

- Don’t think of programs, think of interpreters and refining interpreters with new features
When you define advise or introductions in AspectJ, you are refining (adding features to) the Java interpreter!
- effects of advice are PROGRAM WIDE
- advises entire program (no matter when introductions are made)
- "unbounded" advice basic to AOP

When you refine a program in FOP
- effects of advise limited to the current state of a program’s design
- “bounded advice”

Refining programs ≠ refining language interpreters!

Historically, incremental software design (e.g., agile programming) never “refines” interpreters, only “programs”

Example of UnBounded Quantification

Program P’

```java
class C {
    int i,j,k;
    void setI'(int x){ i=x; }
    void setJ'(int x){ j=x; }
    void setK'(int x){ k=x; }
}
after(): execution (void set*(..))
    { print("hi"); }
```

P’ = hi(C.k + C.setK + C.i+C.j+C.setI+C.setJ )

Example of Bounded Quantification

Program P

```java
class C {
    int i,j,k;
    void setI(int x){ i=x; }
    void setJ(int x){ j=x; }
    void setK(int x){ k=x }
}
after(): execution (void set*(..))
    { print("hi"); }
```

P = C.k + C.setK + hi(C.i+C.j+C.setI+C.setJ )

Different Kinds of Quantification

- May need both because they are doing semantically different things for different purposes
  - bounded advice standard for program synthesis
  - unbounded advice used for invariants – program-wide constraints
- Architectural metaprogramming shows these distinctions
Looking Forward

• Notice:
  • refactoring
  • advice
  • introductions
  • modify structure of code but could also modify structure of grammars, makefiles, xml documents, MDD models ... as well

• Generalizing meta-algebra beyond code structures to non-code structures...
  • theory applies to all documents that can be synthesized

Introduction

• **Model Driven Design (MDD)** is an emerging paradigm for software creation
  • uses domain-specific languages (DSL)
  • encourages automation
  • exploits data exchange standards

• Model is written in a DSL
  • captures particular details of program’s design
  • several models are needed to specify a program
  • models can be derived from other models by transformations

  program synthesis is transforming high-level models into executables (which are also models)
  • Bezivin “Everything is a Model”

MDD Tools

• OMG’s **Model Driven Architecture (MDA)**
  • define models in terms of UML
  • transform models using graph transformations (QVT)

• First and best works I’ve seen is Vanderbilt’s **Model Integrated Computing (MIC)** and Tata’s MDD work and **MasterCraft** tools

• Lots of other groups:
  • Eclipse
  • Microsoft's Software Factories
  • Borland
  • ...

Don Batory
UT-Austin Computer Sciences

Don Batory
UT-Austin Computer Sciences

Don Batory
UT-Austin Computer Sciences

Don Batory
UT-Austin Computer Sciences
**Metaprogramming Connection**

- MDD embraces concept that program development is a *computation*
  - **Claim:** MDD is a metaprogramming paradigm
  - Models are values
  - Transformations are functions that map models to models

**Interesting Question**

- If javac is a transformation, is it distributive?

**javac is not distributive!**

Although there is research by Ancona et. al. on Separate Class Compilation that makes it so...

---

**More Typical MDD Example: PinkCreek**

- Work with S. Trujillo and O. Diaz
- Portlet is a web component
- PinkCreek is an MDD case study for synthesizing portlets
- Uses transformations to map an annotated state chart (SC) to different representations (Java, JSP code)

---

**Portlet Synthesis Metaprogram**

Example of using transformations to derive different models or representations of a program
Another Interesting Question...

As FOP and MDD are both metaprogramming paradigms, how do they combine?

In the Future

- Features “extend” or “refine” models
- An example:

\[
F(x) = i + a(x)
\]

Fundamental Relationship

- Relationship between transformations that derive models and those that refine models

How Commuting Diagrams are Created

- Begin with derivation of representations of base program
- Each feature refines each of these representations
Property of Commuting Diagrams

- Given model in upper left, often want to compute model in lower right
- Any path from upper left to lower right should produce same result
- Each path represents a different metaprogram that produces same result

Example: Refining State Charts in PinkCreek

- Features refine state charts by adding new states, transitions, annotations, etc.

How State Charts are Refined in PinkCreek

Commuting Diagrams in PinkCreek

- Features map space of artifacts by refining them
- Composing features sweeps out the commuting diagrams to traverse to synthesize portlet representations
Portlet Synthesis

- Start at upper left
- compute nodes on lower right
- #1: refine models and then derive
- #2: derive representations and then refine
- #2 is faster by a factor of 2-3
- Diagrams tell us different ways in which programs can be synthesized

Experience

- Our tools initially did not satisfy properties commuting diagrams
  - synthesizing via different paths yielded different results
  - exposed errors in our tools & specifications
- Significance of commuting diagrams
  - validity checks provide assurance on the correctness of our model abstractions, portlet specifications, and our tools
  - applies also to individual transformations (as they too have commuting diagrams)
  - win – better understanding, better model, better tools
  - reduce problem to its essence

In the Future

- Theory, methodology, tools of architectural metaprogramming use elementary ideas from Category Theory
  - where homomorphisms, pushouts, commuting diagrams arise...
  - finding utility in relating software structures to mathematical structures
  - preliminary results are encouraging

Benefit: Interesting Optimization

- Which way is faster?
  - (A) compose transformations
  - (B) transform compositions
- Diagrams tell us different ways in which programs can be synthesized
A Brief Tutorial on Category Theory

**Conclusions**

Don Batory
UT-Austin Computer Sciences

---

**A Brief Tutorial on Category Theory**

- *Category* is a directed graph with special properties
- Nodes are **objects**, edges are **arrows**
- Arrows are maps that compose
- Arrow composition is associative
- Identity arrows

**A Category – Look Familiar?**

- *Category* is a directed graph with special properties
- Nodes are **objects**, edges are **arrows**
- Arrows are maps that compose
- Arrow composition is associative
- Identity arrows are implied

**Functors**

- Structure preserving map between 2 categories
  - embedding of category J into B such that J’s connectivity properties are preserved
- **Manifest functor** between isomorphic categories
  - map each object, arrow in J to the corresponding object, arrow in B

---

Don Batory
UT-Austin Computer Sciences
Functors – Look Familiar?

- Structure preserving map between 2 categories
  - embedding of category J into B such that J's connectivity properties are preserved
- Manifest functor between isomorphic categories
  - map each object, arrow in J to the corresponding object, arrow in B

Next Steps

- Can express many of the ideas of architectural metaprogramming in terms of categorical concepts
- Much more to come...

Conclusions

- Extraordinarily good at:
  - languages
  - compilers
  - optimizations
  - analyses
- Not good at:
  - languages
  - compilers
  - optimizations
  - analyses
- for programming in the small because we:
  - understand abstractions
  - their models
  - their relationships
  - their integration
- programming in the large because we don’t fully:
  - understand abstractions
  - their models
  - their relationships
  - their integration
My Message: Getting Closer

- Fundamental ideas of metaprogramming
  programs are values, transformations, operators

- Provide a simple explanation of technologies that are being developed and built in isolation – there is a lot in common with simple mathematical descriptions

- Recent work in program refactoring, synthesis, and model-driven design are raising level of automation
  success is not accidental
  examples of paradigm called architectural metaprogramming that we are only now beginning to recognize
  many details and connections to other work are still not understood

In the Future...

- Build tools, languages, and compilers to implement metaprogramming abstractions
  improve structure of programs
  higher-level languages & declarative languages
  IDEs will be component evolution calculators
  compilers will be program calculators
  our understanding of programs, their representations, and their manipulation will be greatly expanded beyond source code

- Exciting future awaits us

Recommended Readings


Feature Interactions and Program Cubes

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu
www.cs.utexas.edu/users/dsb/

Copyright is held by the author/owner(s).
Presented at: Lipari School for Advances in Software Engineering
July 8 - July 21, 2007, Lipari Island, Italy

Feature Interactions

• Are unavoidable
• Features interact by changing each others code or behavior
• This lecture looks at one fundamental form of feature interaction called Program Cubes (or Cubes)
  • there are other forms of interaction
• Formalized as tensors (multi-dimensional arrays)

The Calculator Model

• Product line of calculators
  • what operations do you want in your calculator?

\[
C = \{ \\
  \text{Base, // base program} \\
  \text{Add, // add} \\
  \text{Sub, // subtraction} \\
  \text{Form, // format} \\
  \ldots \\
\} \\
\]

• How to express calculators with optional front-ends?
  • none, command-line, GUI1, GUI2, etc

A Micro Example

Don Batory
UT-Austin Computer Sciences
Refactor Model C: Separate Core from GUI

Model Synthesis

- We are synthesizing models!

- A meta model is a model of models
  - product-line of FOP models
  \[ \text{MM} = \{ \text{Core}, \text{Gui}_1, \text{Gui}_2, \ldots, \text{Gui}_n \} \]

- Features of MM are themselves base models or model refinements!

\[
\begin{align*}
\text{Core} &= \{ \text{Base, Add, Sub, Form \ldots} \} & \text{// base calculator model} \\
\text{Gui}_1 &= \{ \text{Base, Add, Sub, Form \ldots} \} & \text{// Gui}_1 \text{ extensions to Core} \\
\text{Gui}_n &= \{ \text{Base, Add, Sub, Form \ldots} \} & \text{// Gui}_n \text{ extensions to Core}
\end{align*}
\]

Don Batory
UT-Austin Computer Sciences

Model Synthesis

- To get desired model, compose Core with desired Gui

\[ C = \text{Gui}_1 \bullet \text{Core} \quad // \text{original model} \]
\[ \text{Desired} = \text{Gui}_n \bullet \text{Core} \quad // \text{desired model} \]

- To specify a calculator need pair of expressions
  - expression to produce a model
  - expression to produce a calculator

- Vast symmetries are fundamental and common to FOP and program families

- Now let's look at the mathematics behind all this

Tensors

Don Batory
UT-Austin Computer Sciences
Tensors

- n-dimensional arrays

- The rank of a tensor is the number of array indices required to describe it
  - cube is a 3D array (tensor of rank 3)
  - matrix is a 2D array (tensor of rank 2)
  - vector is a 1D array (tensor of rank 1)
  - scalar is a 0D array (tensor of rank 0)

- Number of elements along an index is its dimension

- Example: a rank 3 tensor of dimension (2,5,7) is a 3-dimensional array of size $2 \times 5 \times 7$

Basic Tensor Concepts

Tensor Notation

- # of indices indicates rank of tensor
  - name of index is unimportant

Tensor Product

- Cross product of elements of 2 tensors

$$T_{ij} \otimes S_{kn} = M_{ijkm}$$

- $M_{ij}$ – tensor of rank 2
- $M_{ijkn}$ – tensor of rank 4
- Scalar is rank 0

- $T$ is of rank $t$ dim $dt$
- $S$ is of rank $s$ dim $ds$
- $M$ is of rank $t+s$ dim $dt \times ds$

Tensor Product Example

- $R_i = [A, B, C]$ tensor rank 1 dim 3
- $S_k = [D, E, F, G]$ tensor rank 1 dim 4

$$R_i \otimes S_k = \begin{array}{cccc}
AD & AE & AF & AG \\
BD & BE & BF & BG \\
CD & CE & CF & CG \\
\end{array}$$

Result is tensor of rank 2 = 1+1

Dimension 3X4

Tensor Product Example

- $T_l = [X, Y]$ tensor rank 1 dim 2

$$\begin{array}{cccc}
AD & AE & AF & AG \\
BD & BE & BF & BG \\
CD & CE & CF & CG \\
\end{array}$$

Result is tensor of rank 3 = (1+1)+1

Dimension (3X4)X2
**Tensor Contraction**

- Aggregation of entries of a tensor reduces its rank.

- Example: contracting k index of tensor $T_{ikm}$ yields $S_{im}$

$$S_{im} = \sum_k T_{ikm}$$

- Order of aggregation does not matter!
  
  $$\text{scalar} = \sum_{ikm} T_{ikm} = \sum_i \sum_k \sum_m T_{ikm} = \sum_m \sum_i \sum_k T_{ikm} = \ldots$$

**Tensor Projection**

- Remove elements from dimensions
  - not a classical operation in tensor calculus
  - similar to data cubes of database systems

**Program Cubes**
Program Cubes (PCs)

- Are a fundamental design technique in FOP
- Given model \( F = [ F_n, \ldots, F_2, F_1 ] \) // notice vector
- Let program \( G = F_8 + F_4 + F_2 + F_1 \)
  - where + denotes composition operator
- Can write \( G \) as:
  \[
  G = \sum_{i \in (8,4,2,1)} F_i
  \]

Generalize Interpretation

- An FOP model is a vector
  - \( F = [ F_n, \ldots, F_2, F_1 ] \)
    - no longer a set
    - tensor of rank 1, dimension \( n \)
  - A program \( G = \sum_{i \in (8,4,2,1)} F_i \)
    - is a projection of model \( F \)
      - that includes only the needed features
      - features in the vector are in composition order
      - vector is then contracted to a scalar

Program Cubes

- Use \( n \) rank-1 FOP models called dimension models to specify features or indices along a dimension
- A 3-D model \( M \) with \( A, B, C \) as dimension models
  - \( A = [ A_1, \ldots, A_a ] \)
  - \( B = [ B_1, \ldots, B_b ] \)
  - \( C = [ C_1, \ldots, C_c ] \)

Program Cubes

- \( M \) is a tensor product: \( A \otimes B \otimes C \)
- \( M \) has \( axbxc \) entries
  - Entry \( M_{ijk} \) implements the interaction of features \( (A_i, B_j, C_k) \)
    - examples shortly
N-Dimensional Models

- A program is now specified by $n$ expressions
  - 1 per dimension

- Program $P$ in product-line of $M$ has 3 expressions:
  
  $$P = A_6 + A_3 + A_1 = \sum_{i \in (6,3,1)} A_i$$
  
  $$P = B_7 + B_4 + B_3 + B_2 = \sum_{j \in (7,4,3,2)} B_j$$
  
  $$P = C_9 + C_1 = \sum_{k \in (9,1)} C_k$$

Contracting Tensors

- The 3-expression specification of $P$ is translated into an $M$ expression scalar by contracting $M$ along each dimension
  
  $$P = \sum_{i \in (6,3,1)} \sum_{j \in (7,4,3,2)} \sum_{k \in (9,1)} M_{ijk}$$

- Really a projection and contraction to a scalar:
  
  $$P = \sum_{ijk} \left( \Pi_{i \in (6,3,1)} \Pi_{j \in (7,4,3,2)} \Pi_{k \in (9,1)} M_{ijk} \right)$$

Contracting Tensors

- Order in which dimensions are summed (contracted) does not matter!

  $$P = \sum_{k \in (9,1)} \sum_{j \in (7,4,3,2)} \sum_{i \in (6,3,1)} M_{ijk}$$

- Commutativity property of tensor contraction

- Provided that dimensions are orthogonal
  - this needs to be proven

Significance is Scalability!

- Complexity of program is # of features

- Given $n$ dimensions with $d$ features per dimension
  
  - program complexity is $O(d^n)$
  - using cubes $O(d \times n)$

  - ex: program $P$ specified by $3 \times 4 \times 2$ features of $M$ or only $3 + 4 + 2$ dimensional features!

- FOP program specifications are exponentially shorter when using cubes
Academic Legacy

• “Extensibility Problem” or “Expression Problem”
  • classical problem in Programming Languages
  • see papers by: Cook, Reynolds, Wadler, Torgensen
  • focus is on achieving data type and operation extensibility in a type-safe manner

Don Batory
UT-Austin Computer Sciences

Academic Legacy

• Multi-Dimensional Separation of Concerns (MDSoC)
  • Tarr, Ossher IBM

• Cubes are tensor formulation of MDSoC and Expression Problem
  • review a micro example (~35 line programs)
  • then a large example (~35K line programs)
  • synthesis of the AHEAD Tool Suite
  • finally techniques to prove orthogonality of dimensions

Don Batory
UT-Austin Computer Sciences

Calculator Matrix

• View product-line as a matrix
• Tensor product of Calc_r ⊗ GUI_r = CT_{rc}

Calc model

GUI model

<table>
<thead>
<tr>
<th>GUI_1</th>
<th>Cmd</th>
<th>GUI_2</th>
<th>...</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Form_1</td>
<td>Form_c</td>
<td>Form_2</td>
<td>...</td>
<td>Form</td>
</tr>
<tr>
<td>Sub</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>Sub</td>
</tr>
<tr>
<td>Sub_1</td>
<td>Sub_c</td>
<td>Sub_2</td>
<td>...</td>
<td>Sub</td>
</tr>
<tr>
<td>Add</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>Add</td>
</tr>
<tr>
<td>Add_1</td>
<td>Add_c</td>
<td>Add_2</td>
<td>...</td>
<td>Add</td>
</tr>
<tr>
<td>Base</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>Base</td>
</tr>
<tr>
<td>Base_1</td>
<td>Base_c</td>
<td>Base_2</td>
<td>...</td>
<td>Base</td>
</tr>
</tbody>
</table>
Calculator Synthesis is Tensor Contraction

• Define which GUI features to compose
  - MyCalc = GUI₁ + Core
  - project and contract the matrix

<table>
<thead>
<tr>
<th>GUI₁</th>
<th>Cmd</th>
<th>GUI₂</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Form₁</td>
<td>Form₂</td>
<td>Form</td>
<td></td>
</tr>
<tr>
<td>Sub₁</td>
<td>Sub₂</td>
<td>Sub</td>
<td></td>
</tr>
<tr>
<td>Add₁</td>
<td>Add₂</td>
<td>Add</td>
<td></td>
</tr>
<tr>
<td>Base₁</td>
<td>Base₂</td>
<td>Base</td>
<td></td>
</tr>
</tbody>
</table>

• Define which GUI features to compose
  - MyCalc = GUI₁ + Core
  - project and contract the matrix

<table>
<thead>
<tr>
<th>GUI₁</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>+ ...</td>
</tr>
<tr>
<td>Form₁</td>
<td>+ Form</td>
</tr>
<tr>
<td>Sub₁</td>
<td>+ Sub</td>
</tr>
<tr>
<td>Add₁</td>
<td>+ Add</td>
</tr>
<tr>
<td>Base₁</td>
<td>+ Base</td>
</tr>
</tbody>
</table>

process is symmetrical get equivalent result if rows are contracted first

Calculator Synthesis is Tensor Contraction

• Define which Calc features to compose
  - MyCalc = Add + Base
  - project and contract the matrix

<table>
<thead>
<tr>
<th>GUI₁</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>+ ...</td>
</tr>
<tr>
<td>Form₁</td>
<td>+ Form</td>
</tr>
<tr>
<td>Sub₁</td>
<td>+ Sub</td>
</tr>
<tr>
<td>Add₁</td>
<td>+ Add</td>
</tr>
<tr>
<td>Base₁</td>
<td>+ Base</td>
</tr>
</tbody>
</table>

MyCalc = Add₁ + Add + Base₁ + Base
Calculator Synthesis is Tensor Contraction

• Define which Calc features to compose
  • MyCalc = Add + Base
  • project and contract the matrix

 Don Batory
 UT-Austin Computer Sciences
  
  tensors 33

Calculator Synthesis is Tensor Contraction

• Define which GUI features to compose
  • MyCalc = GUI₁ + Core
  • project and contract the matrix

 Don Batory
 UT-Austin Computer Sciences
  
  tensors 34

Calculator Synthesis is Tensor Contraction

• Define which Calc features to compose
  • MyCalc = Add + Base
  • project and contract the matrix

 Don Batory
 UT-Austin Computer Sciences
  
  tensors 35

Calculator Synthesis is Tensor Contraction

• Define which GUI features to compose
  • MyCalc = GUI₁ + Core
  • project and contract the matrix

 MyCalc = Add₁ + Base₁ + Add + Base

 Don Batory
 UT-Austin Computer Sciences
  
  tensors 36
Calculator Synthesis is Tensor Contraction

- Note generated expressions are not syntactically identical
  - columns, rows:
    \[ \text{Mycalc} = \text{Add}_1 + \text{Add} + \text{Base}_1 + \text{Base} \]
  - rows, columns:
    \[ \text{Mycalc} = \text{Add}_1 + \text{Base}_1 + \text{Add} + \text{Base} \]

- Expressions are equal because \( \text{Add} \) and \( \text{Base}_1 \) are commutative (orthogonal)
  - see how we prove this property later…

When to Use Multiple Dimensions?

- Rule: When adding a feature requires the lock-step updating of many other features
  - row feature updates all columns
  - column feature updates all row features

Perspective

- So far, our models customize individual programs
  - set of all such programs is a product-line

- Tool Suite is an integrated set of programs, each with different capabilities
  - MS Office (Excel, Word, Access, ...)

- Question: Do features scale to tool suites?
  - product-line of tool suites
IDEs: A Tool Suite

**Integrated Development Environment (IDE)**
- suite of tools to write, debug, document programs
- AHEAD variant: Java language extensibility

Don Batory  
UT-Austin Computer Sciences  
tensors 41

The Problem – Declarative IDE

From this declarative DSL spec, how do we generate AHEAD tools?

Don Batory  
UT-Austin Computer Sciences  
tensors 42

Define Dimensional Model #1

- AHEAD Model of Java Language Dialects

\[ J = [ \text{Java, Sm, Tmpl, Ds, ...} ] \]

Don Batory  
UT-Austin Computer Sciences  
tensors 43

Define Orthogonal Model #2

- Tools can be specified by a different, orthogonal model

\[ IDE = [ \text{Parse, ToJava, Harvest, Doclet, ...} ] \]

Don Batory  
UT-Austin Computer Sciences  
tensors 44

constant  functions (optional features)

\[ \text{J} = [ \text{Java, Sm, Tmpl, Ds, ...} ] \]

constant  functions (optional features)

\[ \text{IDE} = [ \text{Parse, ToJava, Harvest, Doclet, ...} ] \]

constant  functions (optional features)

\[ \text{Jak} = \text{Tmpl} + \text{Sm} + \text{Java} \]
\[ \text{jak2java} = \text{ToJava} + \text{Parse} \]
\[ \text{jedi} = \text{Doclet} + \text{Harvest} + \text{Parse} \]

...
Tool Specification

- Defined by a pair of expressions
  - one defines tool language
  - other defines tool actions

- ex: \texttt{jedi} (i.e., javadoc) for the Jak dialect of Java
  \[
  \text{jedi} = \text{Tpl} + \text{Sm} + \text{Java} \quad \text{// using J Model}
  \]
  \[
  \text{jedi} = \text{Doclet} + \text{Harvest} + \text{Parse} \quad \text{// using IDE Model}
  \]

- Synthesize \text{jedi} by projecting and contracting the tensor product of the J and IDE models

Tensor for \text{jedi}

- Rows are language features
- Columns are tool features
- Entries are modules (refinements) that implement a language feature for a tool feature
- Shows relationship between IDE and J models

\[
\begin{array}{ccc}
\text{Doclet} & \text{Harvest} & \text{Parse} \\
\text{Java} & \text{JDoclet} & \text{JHarvest} & \text{JParse} \\
\text{Sm} & \text{SDoclet} & \text{SHarvest} & \text{SParse} \\
\text{Tmpl} & \text{TDoclet} & \text{THarvest} & \text{TParse}
\end{array}
\]

Tensor for \text{jedi}

- Composition of these modules yields \text{jedi}
- Synthesize \text{jedi} expression by contracting the tensor according to its dimensional expressions

Contract the Tensor!

- IDE expression
  \[
  \text{jedi} = \text{Doclet} + \text{Harvest} + \text{Parse}
  \]
- Tells us the column summation order
Now Contract the Rows

- **J expression**
  \[ jedi = Tmpl + Sm + Java \]
- Tells us the row summation order

\[
\begin{array}{ccc}
\text{Doclet} & \text{Harvest} & \text{Parse} \\
\hline
\text{Java} & ( JDoclet + JHarvest + JParse ) & \\
\text{Sm} & ( SDoclet + SHarvest + SParse ) & \\
\text{Tmpl} & ( TDoclet + THarvest + TParse ) & \\
\end{array}
\]

Resulting Expression

\[ jedi = ( TDoclet + THarvest + TParse ) + ( SDoclet + SHarvest + SParse ) + ( JDoclet + JHarvest + JParse ) \]

Using Cubes we can synthesize an expression for a language-dialect specific tool

Using Cubes to Generate

- Tool Suites...

Product-Line Tensor

- That relates J and IDE models
- Rows are language features
- Columns are tool features
- Entries implement feature interactions (refinements)

<table>
<thead>
<tr>
<th></th>
<th>Parse</th>
<th>ToJava</th>
<th>Harvest</th>
<th>Doclet</th>
<th>Signat</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Java</strong></td>
<td>JParse</td>
<td>J2Java</td>
<td>JHarvest</td>
<td>JDoclet</td>
<td>JSig</td>
</tr>
<tr>
<td><strong>Sm</strong></td>
<td>SParse</td>
<td>S2Java</td>
<td>SHarvest</td>
<td>SDoclet</td>
<td>SSig</td>
</tr>
<tr>
<td><strong>Tmpl</strong></td>
<td>TParse</td>
<td>T2Java</td>
<td>THarvest</td>
<td>TDoclet</td>
<td>TSig</td>
</tr>
<tr>
<td><strong>Ds</strong></td>
<td>DParse</td>
<td>D2Java</td>
<td>DHarvest</td>
<td>DDoclet</td>
<td>DSig</td>
</tr>
</tbody>
</table>
To Synthesize IDE Tools

- Project unneeded rows and columns
  - directly from IDE GUI input
  - example: jedi, jak2java for Java + Sm + Tmpl

<table>
<thead>
<tr>
<th></th>
<th>Parse</th>
<th>ToJava</th>
<th>Harvest</th>
<th>Doclet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>JParse</td>
<td>J2Java</td>
<td>JHarvest</td>
<td>JDoclet</td>
</tr>
<tr>
<td>Sm</td>
<td>SParse</td>
<td>S2Java</td>
<td>SHarvest</td>
<td>SDoclet</td>
</tr>
<tr>
<td>Tmpl</td>
<td>TParse</td>
<td>T2Java</td>
<td>THarvest</td>
<td>TDoclet</td>
</tr>
</tbody>
</table>

Tensor for IDE Tools

- Contract rows
- Note the semantics of the result...

Yields Expression For Each Tool Feature!

- Parse = TParse + SParse + JParse
- ToJava = T2Java + S2Java + J2Java
- Harvest = THarvest + SHarvest + JHarvest
- Doclet = TDoclet + SDoclet + JDoclet

And we know expressions for each tool!

jak2java = ToJava + Parse
jedi = Doclet + Harvest + Parse
...

IDE Generator is Simple

- For each selected tool, evaluate its expression
  - And generate the code for each tool automatically!
**Generator of IDE Tool Suite**

- Engineer
- Cube generator
- $h_3 + g_3 + f_3$ to generator $\rightarrow$ jak2java
- $h_2 + g_2 + f_2$ to generator $\rightarrow$ jedi
- $h_1 + g_1 + f_1$ to generator $\rightarrow$ ...

**Experimental Results**

**Bootstrapping AHEAD**

- We contracted a tensor of rank 3, dimension $(8 \times 6 \times 8)$ to generate 5 tools of the AHEAD Tool Suite

**Tool Features**

- Lang
- 3rd dimension captures language feature interactions

**Bootstrapping AHEAD**

- Contract tensor to produce IDE model, from which we can generate tool expression

**Tool Features**

- Sum 3rd dimension
- Sum rows

**IDE Model**
Results of AHEAD Bootstrap

- 90 distinct features
- Typical tool contains 20-30 features
  - most tools share 10 features
- Generated Java for each tool is ~35K LOC
- Generating well close to 150K from simple, declarative specifications
  - exactly what we want
- Making designs for multiple tools to conform to a tensor
  - controlling the complexity of tool suites

Don Batory
UT-Austin Computer Sciences

tensors 61

Tensor Representations Scale!!

- Micro example ~150 LOC total
- AHEAD example ~150K LOC total
- 3 orders of magnitude!
- Cubes apply to all levels of abstraction equally
- Cubes scale to much larger systems

Don Batory
UT-Austin Computer Sciences

tensors 62

Proving Commutativity Properties of Tensors

On going work…

Don Batory
UT-Austin Computer Sciences

tensors 63

Contracting Tensors

- We assumed a basic property of tensors
- Order in which dimensions are contracted does not matter
  - commutativity property that we have to verify
- Cubes need not be orthogonal, as next example shows

Don Batory
UT-Austin Computer Sciences

tensors 64
Example of Non-Orthogonal Cube

- A non-orthogonal Cube

| void run() { | void run() { |
| lit.print(); | Super().run(); |
| } | lit.eval(); |
| } | } (a)
| void run() { | void run() { |
| Super().run(); | lit.eval(); |
| add.print(); | add.print(); |
| } | } (b)
| by rows first | by columns first |

So What?

- Contract tensors differently to provide different views of software
  - viewing modules from language feature viewpoint or tool feature view point is occasionally useful

- Properties derived in one view (contraction), might not hold in other views

- Edits or code repairs performed in one view might not work correctly in other views

- Need consistent views!!
  - simple design changes can make a cube orthogonal

A Fix: An Orthogonal Cube

- An orthogonal cube and its contraction

| void run() { | void run() { |
| pr(); | pr(); |
| } | ev(); |
| pr() | ev() |
| { | { |
| Super().pr(); | Super().ev(); |
| add.print(); | add.eval(); |
| } | } (a)
| void ev() { | void ev() { |
| Super().ev(); | Super().ev(); |
| add.eval(); | add.eval(); |
| } | } (b)
| void pr() { | void pr() { |
| lit.print(); | lit.print(); |
| } | add.print(); |
| void add.print(); | } |

Properties to Preserve

- Same program must be synthesized when tensor dimensions are contracted in any order

- For a tensor A of rank 2:
  \[ \Sigma_i \Sigma_k A_{ik} = \Sigma_k \Sigma_i A_{ik} \]

- For a tensor of rank n, there are n! summation orders, all must produce equivalent results

- Need algorithms to verify these properties
Orthogonality Property

- Reduces to testing 2D matrix

$$\sum_{i \in \{1, 2\}} \sum_{j \in \{1, 2\}} a_{ij} = \sum_{i \in \{1, 2\}} \sum_{j \in \{1, 2\}} a_{ij}$$

$$a_{11} + a_{12} + a_{21} + a_{22} = a_{11} + a_{12} + a_{21} + a_{22}$$

- For the above to be equal, the following must hold

$$a_{21} + a_{12} = a_{12} + a_{21}$$

- composition of the bottom left and upper right quadrants must commute

Don Batory
UT-Austin Computer Sciences
tensors 71

a_{21} and a_{12} commute if

- (1) they do not add or refine the same member
  - they add or refine non-overlapping sets of methods and variables

- (2) they do not refer to members added by each other

- Both conditions are easy to verify;
  the hard part is doing so efficiently
  - brute force doesn’t work as it would be hideously slow

Don Batory
UT-Austin Computer Sciences
tensors 70

Essence of the Algorithm

- For an arbitrary rank, dimension tensor T

- For every member m added or refined in feature F, store it along with the coordinates of F in T in a hash table

- If a prior definition of m exists (meaning it was added or refined by another feature G), see if the coordinates of F and G conflict and if they do, see if F and G can belong in the same product
  - if so, T is not orthogonal

- Similar analysis for references

- Almost linear in the size of the code base

Don Batory
UT-Austin Computer Sciences
tensors 69

Example: Bali Tools of ATS

<table>
<thead>
<tr>
<th>TOOLS</th>
<th>Base</th>
<th>CodGen</th>
<th>Bali2Jak</th>
<th>Bali2javacc</th>
<th>Bali Composer</th>
<th>Bali2Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANG</td>
<td>kernel, bali, visitor, collect</td>
<td>codegen</td>
<td>bali2jak</td>
<td>bali2javacc</td>
<td>composer</td>
<td>bali2layer, b2Optns, b2GUI</td>
</tr>
<tr>
<td>G</td>
<td>WebReqFeature</td>
<td>require</td>
<td>require</td>
<td>require</td>
<td>require</td>
<td>require</td>
</tr>
<tr>
<td></td>
<td>Without</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bali : Lang Tools ;
Lang : core CoreFeatures ;
CoreFeatures = ::baliBaliComposerFeature | without ;

Tools: base [codeGen] Tool :: BaliTools ;
Tool : baliJava : bali2jakTool | bali2javacc | baliComposerTool |
| bali2layer : bali2layerTool |

%%
BaliComposerTool implies not codeGen ;
Bali2JakTool or Bali2LayerTool or Bali2javaccTool iff codeGen ;

Don Batory
UT-Austin Computer Sciences
tensors 72
Example Error

- Require refines method defined in Composer

```java
composer.main
public Object driver(String[] args) throws Throwable {
    setVersion("v2003.02.17");
    Collector collector = collectSources(inpFiles);
    return collector;
}
```

```java
require.main
public Object driver(String args[]) throws Throwable {
    setVersion("v2002.09.03");
    return Super(String[]).driver(args);
}
```

```
require.main
public Object driver(String[] args) throws Throwable {
    setVersion("v2002.09.03");
    return Super(String[]).driver(args);
}
```

Another Error

<table>
<thead>
<tr>
<th>TOOLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>Core</td>
</tr>
<tr>
<td>WithReqFeature</td>
</tr>
</tbody>
</table>

```java
bali.main
public Object driver(String args[]) throws Throwable {
    setVersion("v2002.09.04");
    return parseTree;
}
```

```java
require.main
public Object driver(String args[]) throws Throwable {
    setVersion("v2002.09.03");
    return Super(String[]).driver(args);
}
```

```java
codegen.main
public Object driver(String args[]) throws Throwable {
    setVersion("v2002.09.04");
    return Super(String[]).driver(args);
}
```

```java
require.main
public Object driver(String args[]) throws Throwable {
    setVersion("v2002.09.03");
    return Super(String[]).driver(args);
}
```

Other Statistics

- Fast – didn’t find errors in JPL

```java
<table>
<thead>
<tr>
<th>Product Line</th>
<th>Baseline # of Features</th>
<th>Baseline # of Programs</th>
<th>Code Base Java LOC</th>
<th>Program Java LOC</th>
<th>Time to Run (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bail Product Line</td>
<td>2</td>
<td>17</td>
<td>8</td>
<td>12K/16K</td>
<td>8K/12K</td>
</tr>
<tr>
<td>Graph Product Line</td>
<td>2</td>
<td>18</td>
<td>80</td>
<td>1800/1800</td>
<td>760/700</td>
</tr>
<tr>
<td>Java Product Line</td>
<td>3</td>
<td>70</td>
<td>56</td>
<td>34K/48K</td>
<td>22K/35K</td>
</tr>
</tbody>
</table>
```
Insights

• Oddly, we didn’t find serious errors in the ATS designs
  • only benign (inconsequential) errors were found

• Created these designs long before we had any analysis tools
  • suggests that creating orthogonal tensors is not difficult

Final Comments

Future Work

• Commutativity or “orthogonal” properties have a simple description in category theory
  • deep interconnection with our use of tensors

• Other forms of feature interactions
  • generalization of the ideas presented here seem to account for many of such interactions
  • developing theories and supporting tools for this

• Additional analyses
  • want to analyze product-lines to ensure that all legal compositions of features yield type safe programs


Recommended Readings


Recommended Readings


