

# Product Lines of Theorems

Don Batory and Ben Delaware

work with William Cook

Department of Computer Science

University of Texas at Austin

Austin, Texas 78712

# Introduction

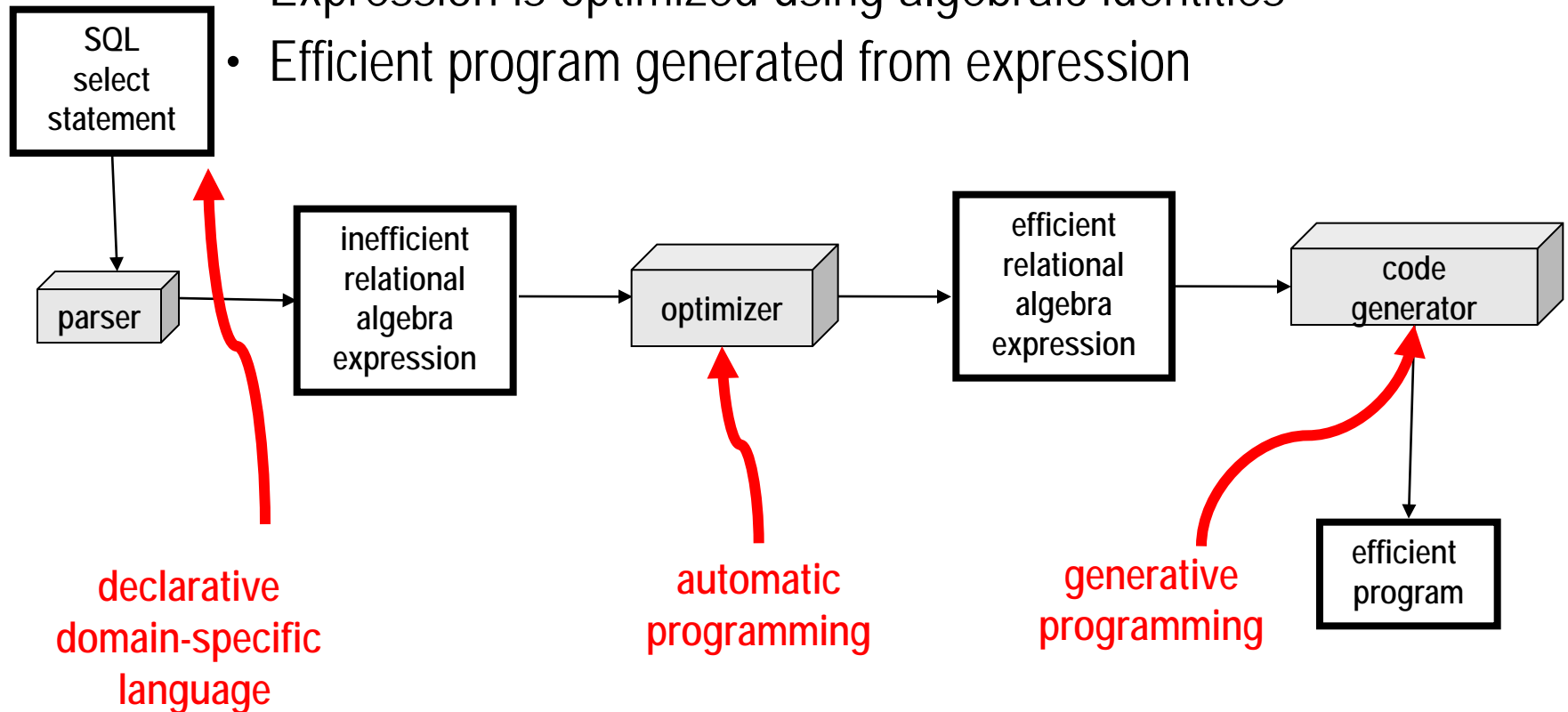
- My background is in database management, not theorem proving
- My interests have always been in software design
  - early work on DBMS implementations
  - transitioned in early 1990s to Software Engineering
  - databases fundamentally shaped my view of software design
- My work focused on **software product lines (SPLs)**
  - set of related programs that are differentiated by “features”
  - **feature** is an “increment in program functionality”
  - different compositions of features yield different programs

# My Contribution

- Understand and explain feature-based software design by simple mathematics
- Easiest way for me to express, conceive, and explain my ideas
  - provided me with a different view of software design whose underpinnings are in categories
  - clear and precise notion of “composition” (function composition)
- My inspiration for automated program generation...

# Relational Query Optimization (RQO)

- Declarative query is mapped to an relational algebra expression
- Each expression is a program
- Expression is optimized using algebraic identities
- Efficient program generated from expression



# Keys to Success of RQO

- Automated development of query evaluation programs
  - hard-to-write, hard-to-optimize, hard-to-maintain
  - revolutionize and simplify database usage
- Represented program designs as **expressions**
- Use algebraic identities to optimize expressions – can optimize program designs
- Compositional: hallmark of great engineering
- **Paradigm to replicate in other domains**

# Purpose of This Talk

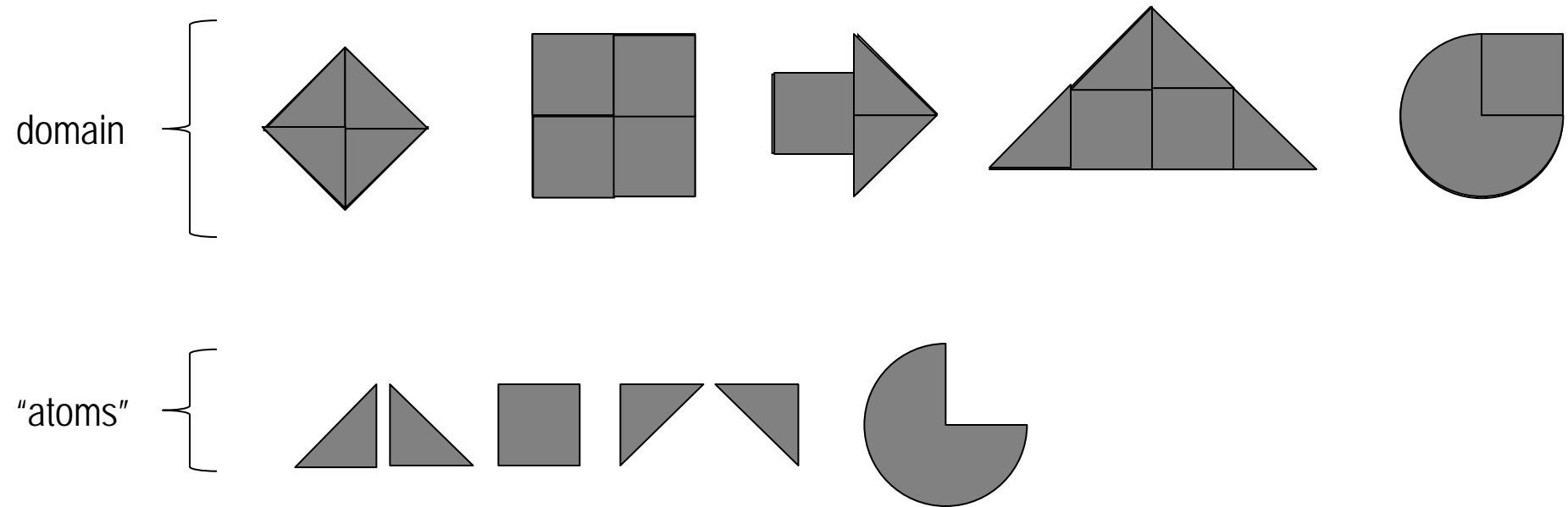
- Explain how RQO paradigm generalizes to SPLs
- Also show how proofs scale from a single program to families of programs – big win
- Within an algebraic framework of automated program generation and SPLs – general approach

quick tutorial on

# SOFTWARE PRODUCT LINES (SPLs)

# Domain Analysis

- Set of structures (programs) from which we want to decompose into more fundamental structures and their compositions

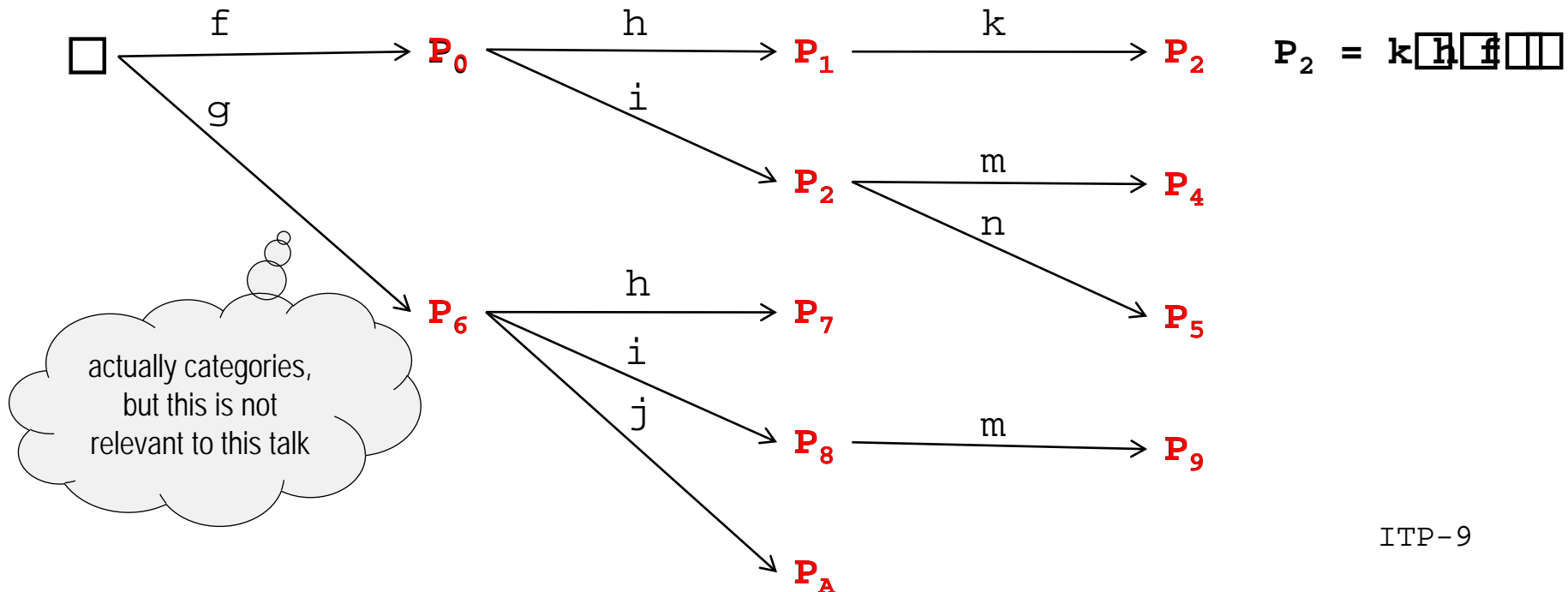


- Standard engineering activity called **Domain Analysis**
- Resulting set of atoms is not necessarily unique



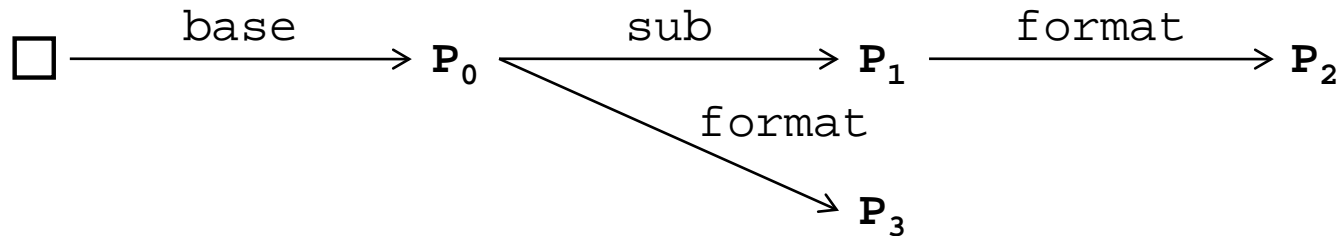
# In Software

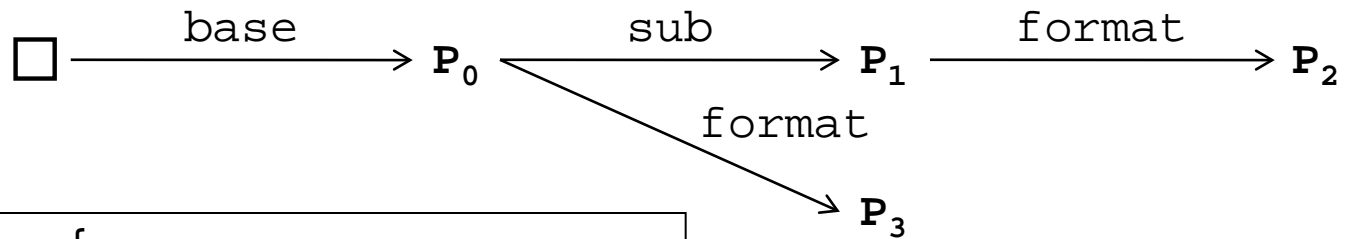
- **Features** are semantic increments in program functionality
- View features as transformations (arrows)
- Programs are defined by a composition of transformations (arrows)
- SPL is a tree whose nodes are programs and arrows are features



# Example: a 4-Program SPL

- Elementary product line of Java calculators





```

class calculator {
    float result;
    void clear() { result=0; }
    void add( float x ) { result+=x; }
    void sub( float x ) { result=-x; }
}
  
```

```

class gui {
    JButton format = new JButton("format");
    JButton add     = new JButton("+");
    JButton sub     = new JButton("-");

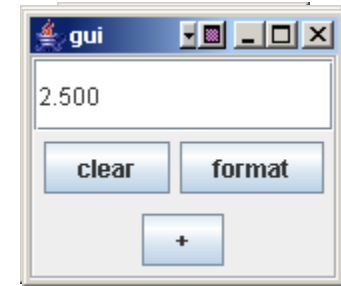
    void initGui() {
        ContentPane.add( format );
        ContentPane.add( add );
        ContentPane.add( sub );
    }

    void initListeners() {
        add.addActionListener(...);
        sub.addActionListener(...);
    }

    void formatResultString() {...}
}
  
```

new methods

new fields



new fields

extend existing methods

extend existing methods

format ☐ sub ☐ base ☐ ☐ = P<sub>3</sub>

new methods

# Ideas Scale...

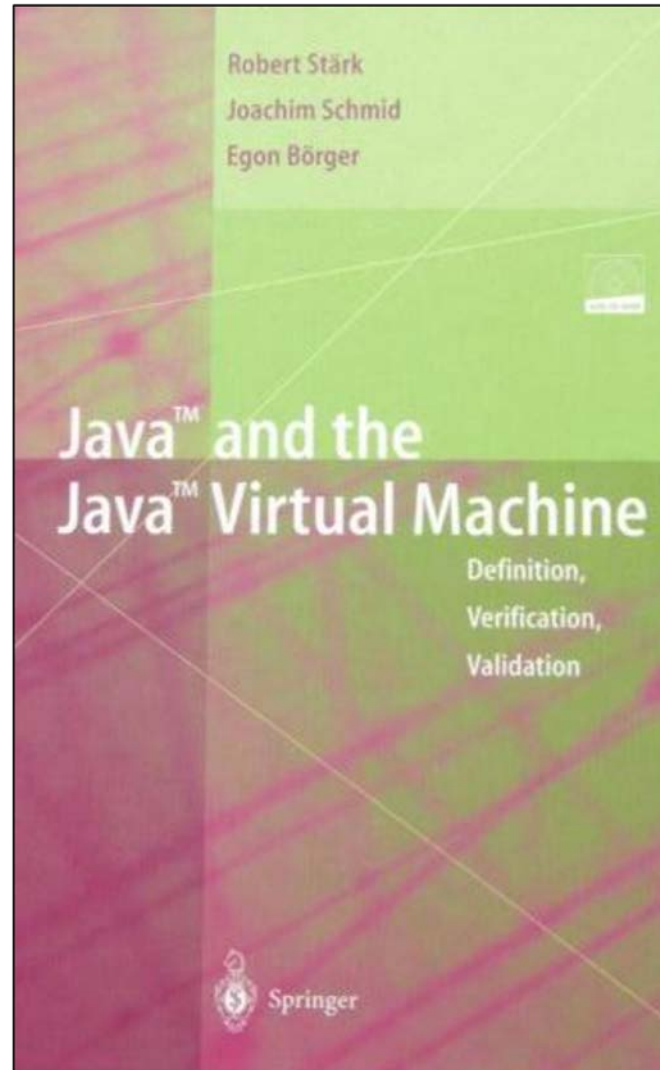
- 1986 database systems 80K LOC
- 1989 network protocols
- 1993 data structures
- 1994 avionics
- 1997 extensible Java preprocessors 40K LOC
- 1998 radio ergonomics
- 2000 program verification tools
- 2002 fire support simulators
- 2003 AHEAD tool suite 250K LOC
- 2004 robotics controllers
- 2006 web portlets
- Others have picked up on these ideas...

# Quick Summary on SPLs

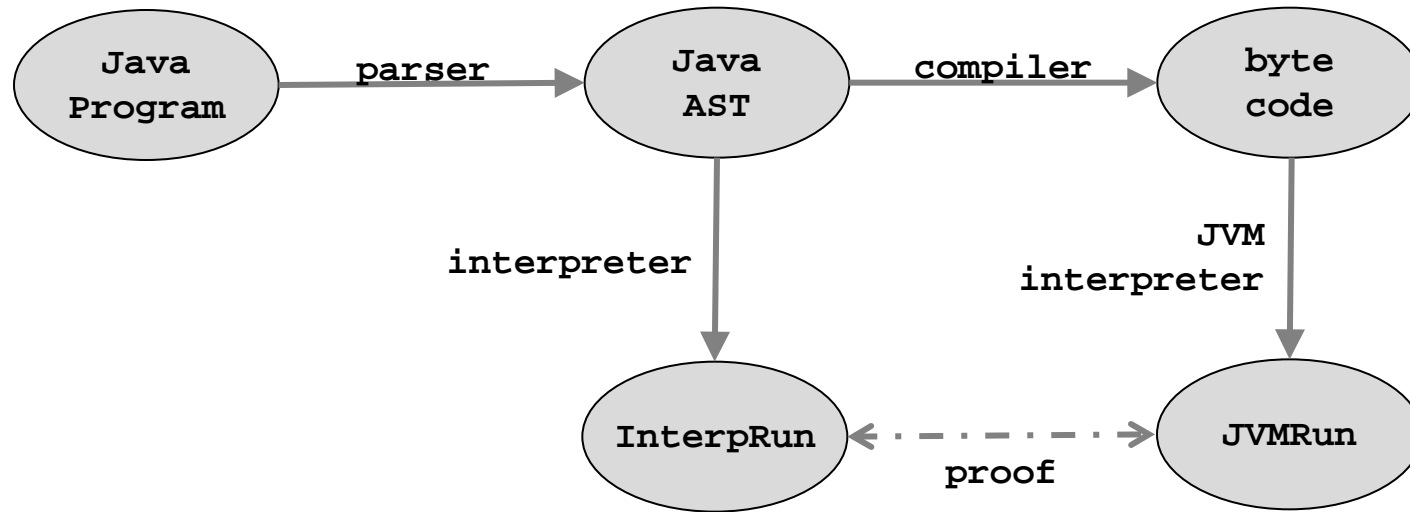
- Using features has right look and feel
  - standard idea in software product lines
  - features as transformations is key to a modern approach
  - feature composition is function composition
  - a generalization of RQO – program designs are expressions
  - design optimization is expression optimization
  - program generation is expression evaluation
- First connection (that I know of) to theorem proving...

Egon Börger's 2001

**JB**BOOK



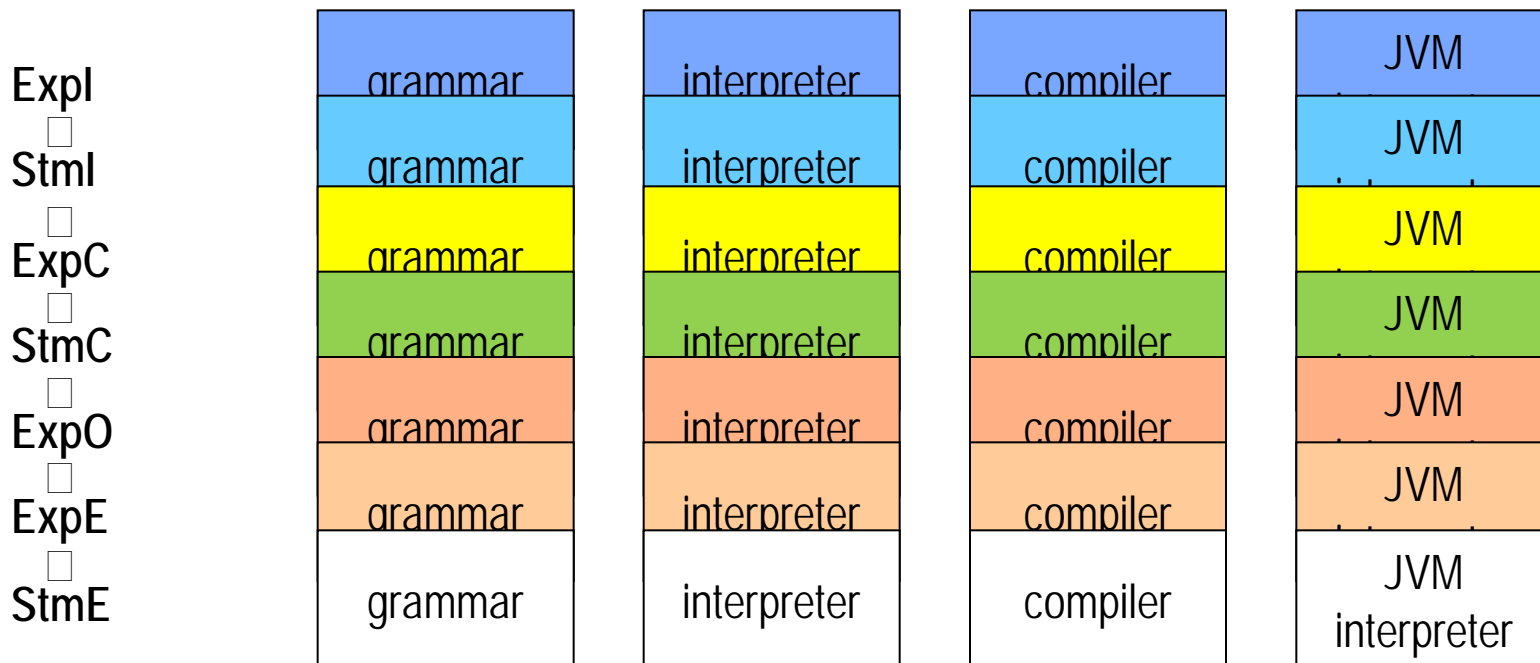
# Structure of JBook



- At this point, various correctness issues are considered
  - ex: equivalence of interpreter execution of program and the JVM execution of compiled program
- JBook not written with product lines in mind
  - definition, correctness of **single** interpreter, compiler of Java1.0
- But the tools (parser, interpreter,...) were developed by features...

# Overview of JBook

- JBook presents structured way to incrementally develop a Java 1.0 grammar, and ASM definitions of an interpreter, compiler, and bytecode (JVM) interpreter
- Start with the sublanguage of imperative expressions and incrementally extend it

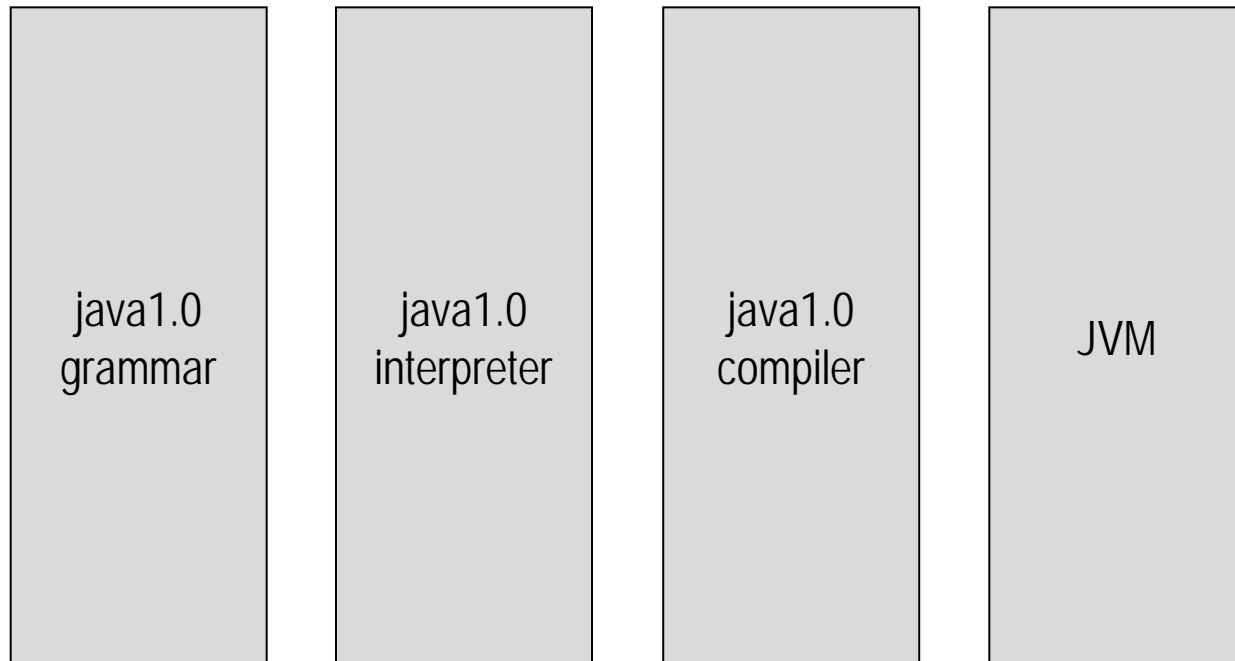




# Overview of JBook

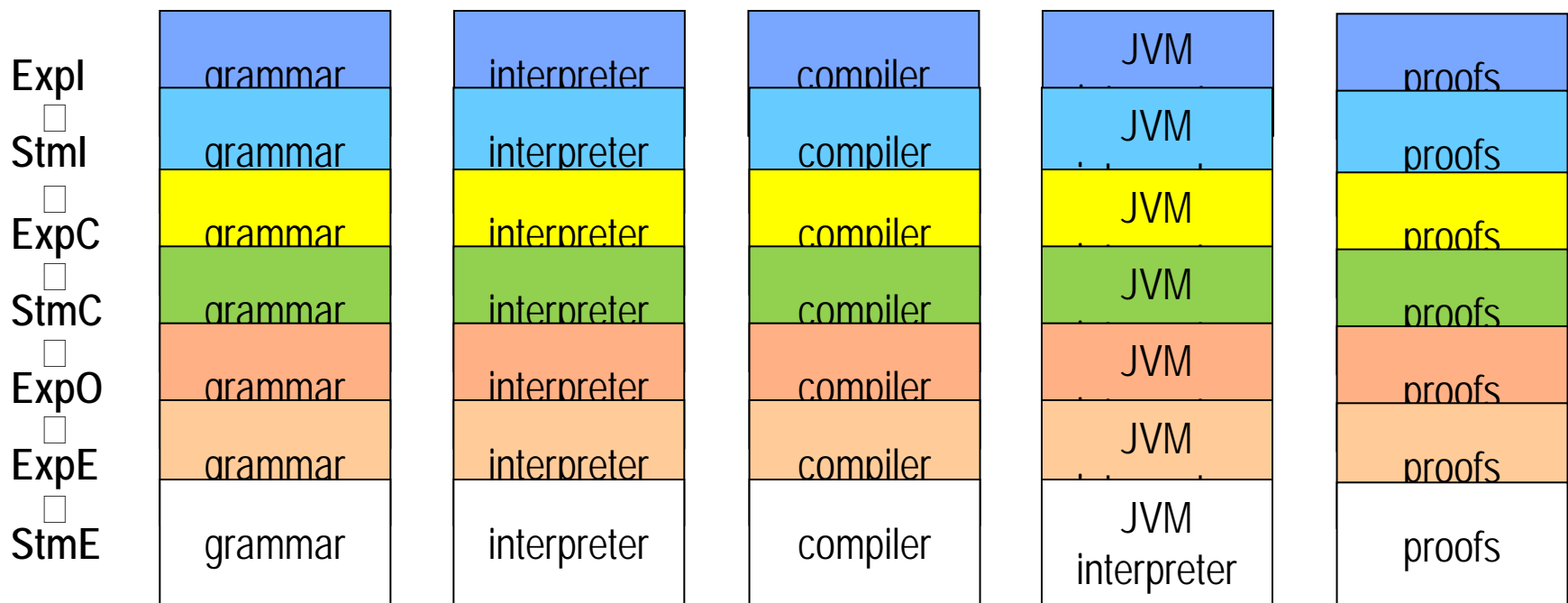
- JBook presents structured way to incrementally develop a Java 1.0 grammar, and ASM definitions of an interpreter, compiler, and bytecode (JVM) interpreter
- Start with the sublanguage of imperative expressions and incrementally extend it
- Only when the Java 1.0 definitions were complete were the proofs constructed

Java1.0



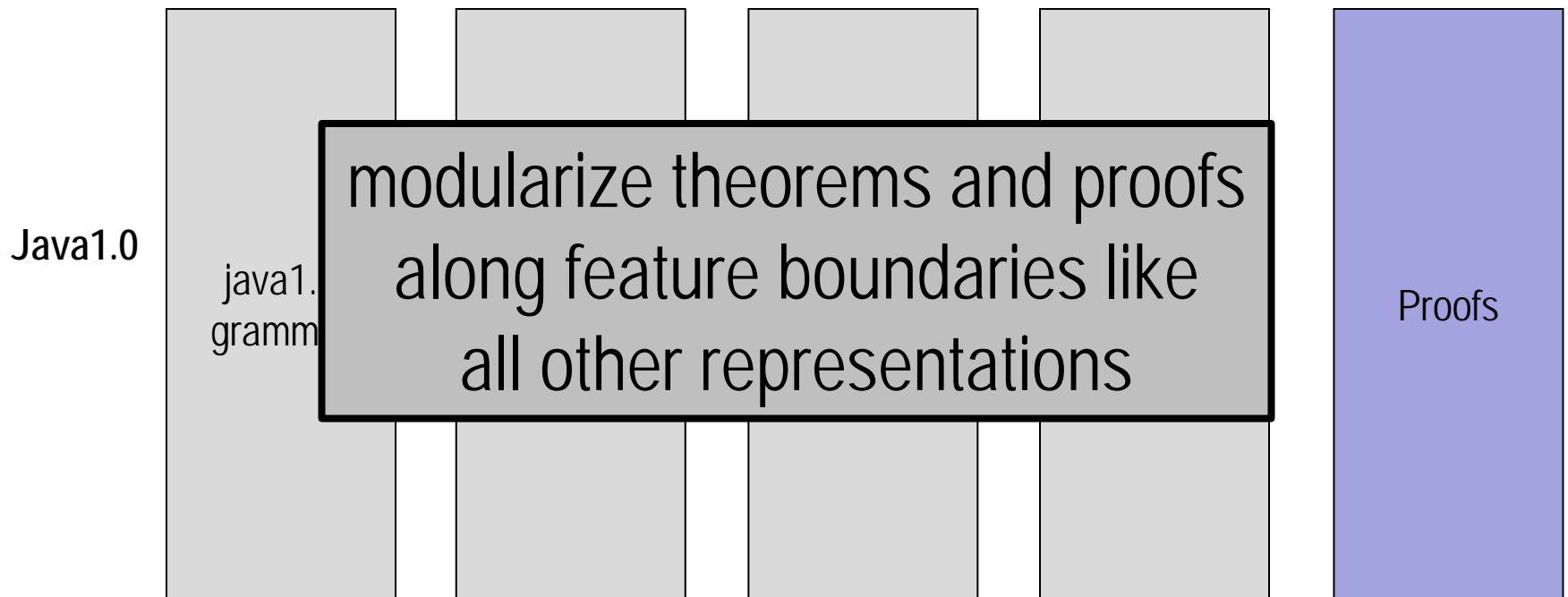
# Features Update All Representations of a Program Lock-Step

- Can develop theorems and proofs incrementally from features as well, lock-step like all other representations – structurally treat them no differently



# Found What We Expected...

- Theorems and proofs could be developed incrementally from features as well, lock-step like other representations – structurally treat them no differently



# Correctness of Compiler

- Statement of theorem is a list of invariants
- 14 invariants in all
- Don't need to know the specifics of the invariants to understand the effects of features

**Theorem 14.1.1 (Correctness of the compiler).** There exists a monotonic mapping  $\sigma$  from the run of the ASM for a  $\text{Java}_\varepsilon$  program into the run of the ASM for the compiled  $\text{JVM}_\varepsilon$  program such that the following invariants are satisfied for  $\alpha = pos_n$ :

(reg)

(stack)

(beg)

(exp)

(bool1)

(bool2)

(new)

(stm)

(abr)

# Statement of Correctness

**Theorem 14.1.1 (Correctness of the Compiler).** There exists a monotonic mapping  $\square$  from the run of the ASM for a Java program into the run of the ASM for the compiled JVM program such that the following invariants are satisfied:

**(reg)**

**(bool1)**

**(exc)**

**(stack)**

**(bool2)**

**(exc-clinit)**

**(begS)**

**(new)**

**(clinit)**

**(begE)**

**(stm)**

**(fin)**

**(exp)**

**(abr)**

---

**Java1.0 = StmE · ExpE · ExpO · StmC · ExpC · StmI · ExpI**

# Proof of Correctness

- Proof is a case analysis using structural induction to show correctness of compiling each kind of expression
  - Proof is a **list of 83 cases** that show invariants holds

14.2 The correctness proof 183

**Case 6.**  $context(pos_n) = \alpha(uop^\beta exp)$  and  $pos_n = \alpha$ :

Similar to Case 3.

**Case 7.**  $context(pos_n) = \alpha(uop^\beta val)$  and  $pos_n = \beta$ :

Similar to Case 5. If  $uop$  is the negation operator and  $\alpha$  is a  $\mathcal{B}_1(lab)$ -position, then according to the compilation scheme in Fig. 9.3, the position  $\beta$  is  $\mathcal{B}_0(lab)$ -position. We set  $\sigma(n+1) := \sigma(n)$  and the invariants (**bool1**) and (**bool2**) for  $\beta$  in state  $n$  can be carried over to  $\alpha$  in state  $n+1$ .

**Case 8.**  $context(pos_n) = \alpha(loc =^\beta exp)$  and  $pos_n = \alpha$ :

Similar to Case 3.

**Case 9.**  $context(pos_n) = \alpha(loc =^\beta val)$  and  $pos_n = \beta$ :

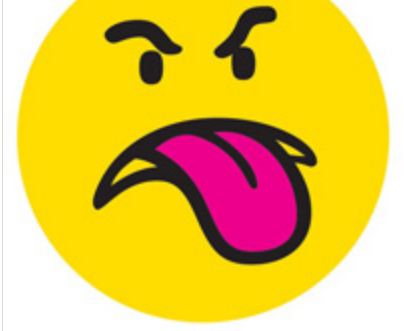
Assume that  $\alpha$  is an  $\mathcal{E}$ -position and that the size of the type of the variable  $loc$  is 1. (The case of size 2 is treated in a similar way.) Accord-

# Adding Cases

- Same pattern repeats
- Invariant refinement: original proof cases remains unchanged

|         |   |    |   |
|---------|---|----|---|
|         | Each program in the JBook product line had a Proof of Correctness.  |    | T |
|         | As features are composed, the theorem is elaborated with new invariants, the proof is extended with new cases and elaborations of existing cases. |    |   |
| Java1.0 |   | 83 |   |

# Reaction...

- JBook proofs were manually created
  - Need to be mechanically verified
- 
- Our conjecture was that theorems + proofs could be generated just like other representations of programs in an SPL
  - Show how our conjecture held with modern tools and approaches
  - Starting point for this work



our current work:

PRODUCT LINE OF THEOREMS

# A Step Forward

- Showed how to build syntax & semantic definitions of a SPL of languages, proofs in features and their compositions are independently certified by Coq proof assistant
- Next slides I'll review algebraic structure that features impose on software development
- Ben will present details on how he accomplished this in Coq
- Future work...

OOPSLA 2011

## Product Lines of Theorems

Benjamin Delaware William R. Cook Don Batory

Department of Computer Science  
University of Texas at Austin  
(bendy, wcook, batory)@cs.utexas.edu

### Abstract

Mechanized proof assistants are powerful verification tools, but proof development can be difficult and time-consuming. When verifying a family of related programs, the effort can be reduced by proof reuse. In this paper, we show how to engineer product lines with theorems and proofs built from feature modules. Each module contains proof fragments which are composed together to build a complete proof of correctness for each product. We consider a product line of programming languages, where each variant includes metatheory proofs verifying the correctness of its syntax and semantic definitions. This approach has been realized in the Coq proof assistant, with the proofs of each feature independently certifiable by Coq. These proofs are composed for each language variant, with Coq mechanically verifying that the composite proofs are correct. As validation, we formalize a core calculus for Java in Coq which can be extended with any combination of casts, interfaces, or generics.

### 1. Introduction

Mechanized theorem proving is hard: large-scale proof developments [13, 16] take multiple person-years and consist of tens of thousand lines of proof scripts. Given the effort invested in formal verification, it is desirable to reuse as much of the formalization as possible when developing similar proofs. The problem is compounded when verifying members of a *product line* – a family of related systems [2, 5] – in which the prospect of developing and maintaining individual proofs for each member is untenable.

Product lines can be decomposed into *features* – units of functionality. By selecting and composing different features, members of a product line can be synthesized. The challenge of feature modules for software product lines is that their contents cut across normal object-oriented boundaries [5,

25]. The same holds for proofs. Feature modularization of proofs is an open, fundamental, and challenging problem.

Surprisingly, the programming language literature is replete with examples of product lines which include proofs. These product lines typically only have two members, consisting of a core language such as *Featherweight Java (FJ)* [14], and an updated one with modified syntax, semantics, and proofs of correctness. Indeed, the original FJ paper also presents *Featherweight Generic Java (FGJ)*, a modified version of FJ with support for generics. An integral part of any type system are the metatheoretic proofs showing *type soundness* – a guarantee that the type system statically enforces the desired run-time behavior of a language, typically preservation and progress [24].

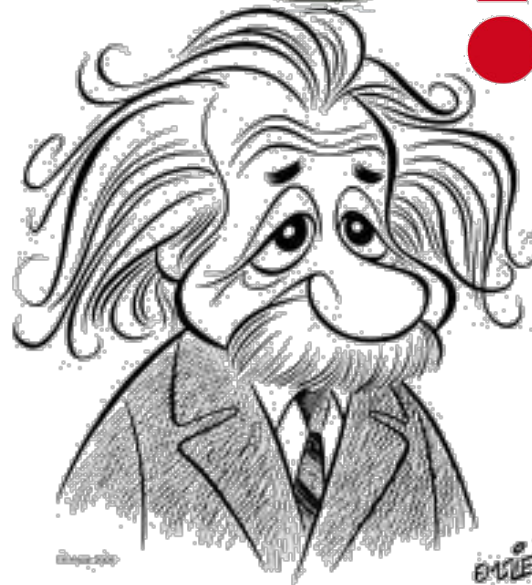
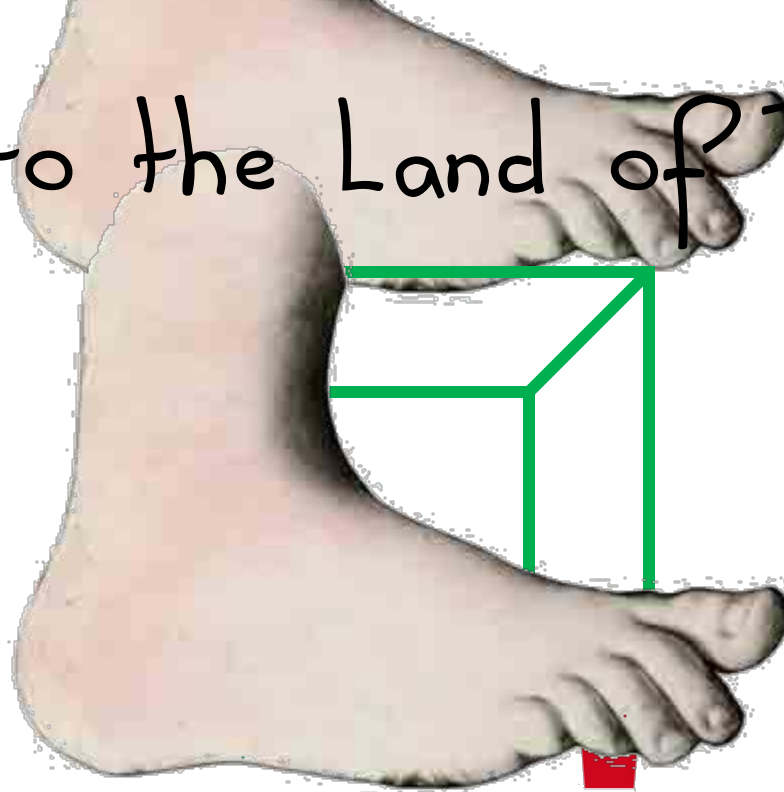
Typically, each research paper only adds a single new feature to a core calculus, and this is accomplished manually. Reuse of existing syntax, semantics, and proofs is achieved by copying existing rules, and in the case of proofs, following the structure of the original proof with appropriate updates. As more features are added, this manual process grows increasingly cumbersome and error prone. Further, the enhanced languages become more difficult to maintain. Adding a feature requires changes that cut across the normal structural boundaries of a language – its syntax, operational semantics, and type system. Each change requires arduously rechecking existing proofs by hand.

Using theorem provers to mechanically formalize languages and their metatheory provides an interesting testbed for studying the modularization of product lines which include proofs. By implementing an extension in the proof assistant as a *feature module*, which includes updates to existing definitions and proofs, we can compose feature modules to build a completely mechanized definition of an enhanced language, with the proofs mechanically checked by the theorem prover. Stepwise development is enabled, and it is possible to start with a core language and add features to progressively build a family or product line of more detailed languages with tool support and less difficulty.

In this paper, we present a methodology for feature-oriented development of a language using a variant of FJ as an example. We implement feature modules in Coq [8] and demonstrate how to build mechanized proofs that can

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.  
OOPSLA '11, October 22–27, 2011, Portland, Oregon, USA.  
Copyright © 2011 ACM 978-1-4503-0940-9/11/10...\$10.00

# Welcome to the Land of Features!



## 5 ideas

# #1: Features and Domains

- Given a domain  $\mathcal{D}$  of programs to generate, identify the core features that underlie the domain via **domain analysis**. Domain  $\mathcal{D}$  has the set of features:

$$\mathcal{D} = \left\{ \begin{array}{ll} B_1 & // \text{ base program 1} \\ B_2 & // \text{ base program 2} \\ F_1 & // \text{ optional feature 1} \\ F_2 & // \text{ optional feature 2} \\ \dots & \\ F_n & // \text{ optional feature n} \end{array} \right.$$

instead of  
starting from  $\square$

- Program in this domain is a composition of features:

$$P_1 = F_n \square F_3 \square F_1 \square B_1$$

$$P_2 = F_4 \square F_1 \square B_2$$

# Our Example

base

- Small product line of 4 features:

|           |                           |
|-----------|---------------------------|
| cFJ       | core Featherweight Java   |
| Cast      | adds casts to expressions |
| Interface | adds interfaces           |
| Generic   | adds type parameters      |

- Different compositions yields different languages:

|                           |                                    |
|---------------------------|------------------------------------|
| cFJ                       | // Core FJ                         |
| Cast · cFJ                | // Original FJ [13]                |
| Interface · cFJ           | // Core FJ with Interfaces         |
| Interface · Cast · cFJ    | // Original FJ with Interfaces     |
| Generic · cFJ             | // Core Featherweight Generic Java |
| Generic · Cast · cFJ      | // Original FGJ                    |
| Generic · Interface · cFJ | // core Generic FJ with            |
|                           | // Generic Interfaces              |
| Generic · Interface       | // FGJ with                        |
| · Cast · cFJ              | // Generic Interfaces              |

## #2: FEATURE MODELS

# Feature Models

- Not all combinations of features are meaningful
- Some features require/preclude other features
- **Feature model** defines the legal combinations
- Is a context sensitive grammar
  - context free grammar whose language include all legal combinations
  - constraints that eliminate nonsensical sentences

```
D : [k] [i] [j] b ;      // context free grammar
    k □ j □ i ;          // additional constraints
    k □ j ;
```

- *Assuming no feature interactions*, sentence of a feature model ('kjb') is mapped to an expression by a dot-product of its terms

$$k \square j \square b$$

# Our Example

**L : [Generic] [Interface] [Cast] cFJ;**

- Is just a context free grammar
- Its language (sentences):

|                           |                                    |
|---------------------------|------------------------------------|
| cFJ                       | // Core FJ                         |
| Cast · cFJ                | // Original FJ [13]                |
| Interface · cFJ           | // Core FJ with Interfaces         |
| Interface · Cast · cFJ    | // Original FJ with Interfaces     |
| Generic · cFJ             | // Core Featherweight Generic Java |
| Generic · Cast · cFJ      | // Original FGJ                    |
| Generic · Interface · cFJ | // core Generic FJ with            |
|                           | // Generic Interfaces              |
| Generic · Interface       | // FGJ with                        |
| · Cast · cFJ              | // Generic Interfaces              |



# #3: LOCK-STEP UPDATE OF REPRESENTATIONS

# Feature Modules

- Every program has multiple consistent representations
  - ex: a parser  $P$  has: grammar, source code, manual
- Base program is a tuple:

$$P = [ \text{gram}_P, \text{src}_P, \text{man}_P ]$$

- Optional feature ( $F$ ) modifies any or all representations

$$F = [ \Delta\text{gram}_F, \Delta\text{src}_F, \Delta\text{man}_F ]$$

# Feature Composition

- Is tuple composition – tuples are composed element-wise
- Extended parser (FP):

$$FP = F \square P$$

$$= [ \Delta\text{gram}_F, \Delta\text{src}_F, \Delta\text{man}_F ] \square [ \text{gram}_P, \text{src}_P, \text{man}_P ]$$

$$= [ \Delta\text{gram}_F \square \text{gram}_P, \Delta\text{src}_F \square \text{src}_P, \Delta\text{man}_F \square \text{man}_P ]$$


grammar of FP

source of FP

manual of FP

# Our Example

- Base language ( $c_{FJ}$ ) has multiple representations



| base representation   | specification |
|-----------------------|---------------|
| syntax                | $s_{c_{FJ}}$  |
| operational semantics | $o_{c_{FJ}}$  |
| type system           | $t_{c_{FJ}}$  |
| meta-theory proofs    | $p_{c_{FJ}}$  |

- Base language is a 4-tuple:

$$c_{FJ} = [ s_{c_{FJ}}, o_{c_{FJ}}, t_{c_{FJ}}, p_{c_{FJ}} ]$$

# Our Example

- An optional feature  $j$  extends each representation:

| representation change | specification |
|-----------------------|---------------|
| syntax                | $\square s_j$ |
| operational semantics | $\square o_j$ |
| type system           | $\square t_j$ |
| meta-theory proofs    | $\square p_j$ |

- Feature  $j$  is a 4-tuple of changes (functions) that update each representation

$$j = [ \Delta s_j, \Delta o_j, \Delta t_j, \Delta p_j ]$$

# Our Example

- Tuple for Featherweight Java  $\mathbb{FJ}$  is:

$\mathbb{FJ}$

$= \text{Cast} \sqsubseteq \text{cFJ}$

$= [ \Delta s_{\text{cast}}, \Delta o_{\text{cast}}, \Delta t_{\text{cast}}, \Delta p_{\text{cast}} ] \sqsubseteq [ s_{\text{cFJ}}, o_{\text{cFJ}}, t_{\text{cFJ}}, p_{\text{cFJ}} ]$

$= [ \Delta s_{\text{cast}} \sqsubseteq s_{\text{cFJ}}, \Delta o_{\text{cast}} \sqsubseteq o_{\text{cFJ}}, \Delta t_{\text{cast}} \sqsubseteq t_{\text{cFJ}}, \Delta p_{\text{cast}} \sqsubseteq p_{\text{cFJ}} ]$

syntax of FJ

semantics of FJ

type system of FJ

theorems and proofs FJ

one more piece...

## #4: FEATURE INTERACTIONS

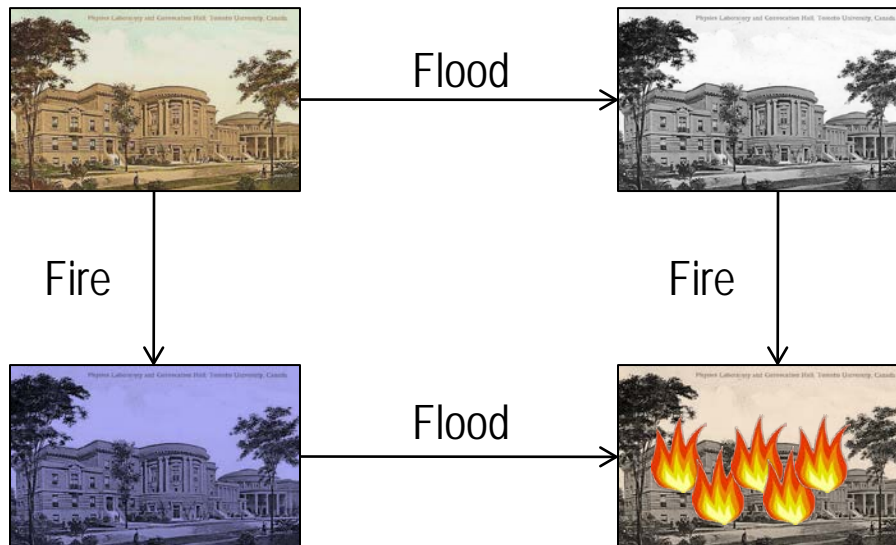
# Feature Interactions

- **Feature interaction (FI)** occurs when two features behave incorrectly together
- **Resolution** of a feature interaction is an additional module/transformation that “patches” features so that they correctly work together
- Illustrate with a classical example



# Feature Interactions

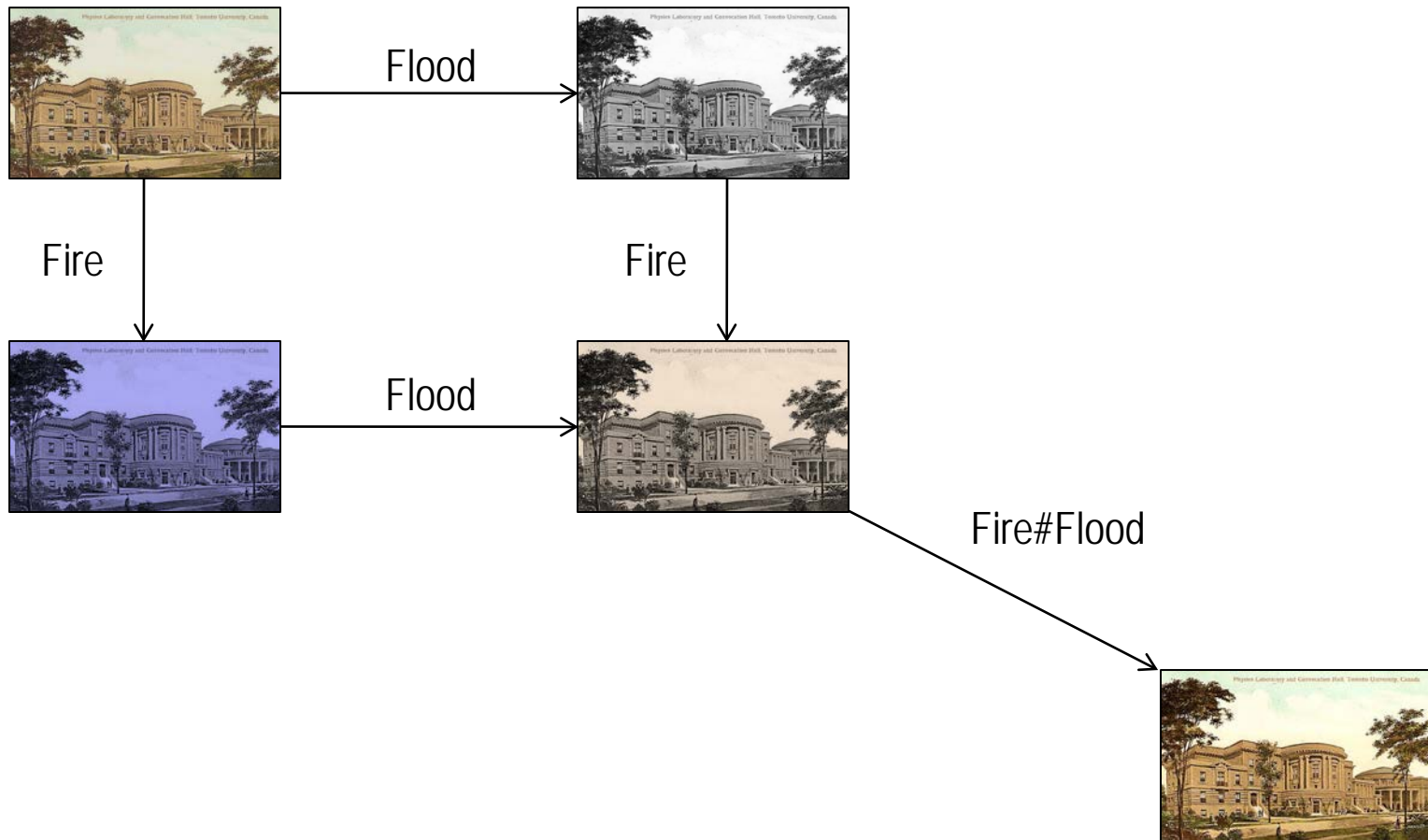
- Flood control – Fire control problem (Kang 2003)
  - isomorphic to feature interaction problems in telephony



fire detected @ i  
sprinklers on @ i+1  
standing water @ i+2  
water turned off @ i+3  
building burns down

# Feature Interactions

- Flood control – Fire control problem (Kang 2003)
  - isomorphic to feature interaction problems in telephony



# New Operations on Features

- Cross-product ( $\times$ ) says we want the integration of two features so that they work together correctly

$$f \times g = (f \# g) \cdot f \cdot g$$

- $\#$  distributes over dot and  $\#$  takes precedence over dot:

$$f \# (g \cdot h) = (f \# g) \cdot (f \# h)$$

interaction of a feature with a dot-product = the dot-product of their interactions

# Connection to Prior Discussions

- *To account for feature interactions*, a sentence of a feature model 'kjb' is mapped to an expression by a cross-product (not by a dot-product) of its terms

$$\begin{aligned}
 p &= k \times j \times b && // \text{ def of } p \\
 &= k \times (j \# b \sqcap j \sqcap b) && // \text{ def of } \times \\
 &= k \# (j \# b \sqcap j \sqcap b) \sqcap k \sqcap (j \# b \sqcap j \sqcap b) && // \text{ def of } \times \\
 &= \boxed{k \# j \# b} \sqcap k \# j \sqcap \boxed{k \# b} \sqcap \boxed{k} \sqcap j \# b \sqcap \boxed{j} \sqcap \boxed{b} && // \# \text{ dist over } \sqcap
 \end{aligned}$$

- So not only do we compose features (k, j, b), we also consider all possible 2-way and 3-way (in general n-way) interactions of these features

# In Our Case Study

| Module            | Description                        |
|-------------------|------------------------------------|
| cFJ               | core Featherweight Java            |
| Cast              | cast                               |
| Interface         | interfaces                         |
| Generic           | generics                           |
| Generic#Interface | generic and interface interactions |
| Generic#Cast      | generic and cast interactions      |

Generic  $\times$  Interface  $\times$  cFJ

$$\begin{aligned}
 &= \text{Generic} \# \text{Interface} \# \text{cFJ} \sqcap \text{Generic} \# \text{Interface} \sqcap \\
 &\quad \text{Generic} \# \text{cFJ} \sqcap \text{Generic} \sqcap \text{Interface} \# \text{cFJ} \sqcap \text{Interface} \sqcap \text{cFJ} \\
 &= 1 \sqcap \text{Generic} \# \text{Interface} \sqcap 1 \sqcap \text{Generic} \sqcap 1 \sqcap \text{Interface} \sqcap \text{cFJ} \\
 &= \text{Generic} \# \text{Interface} \sqcap \text{Generic} \sqcap \text{Interface} \sqcap \text{cFJ}
 \end{aligned}$$

given this super-structure, here's the next key step

## #5: IMPLEMENTING MODULES

# How We Implement Modules

- Design features to be monotonic: what was true before a feature is added is true afterwards – although scope of validity may be qualified
  - standard design technique
- Features are allowed to make 2 kinds of changes
  - add new definitions
  - modify existing definitions
- Single **syntactic** approach for all representations

how we define and modify

# SYNTAX RULES



# Adding Syntax

- Syntax for expressions in  $cFJ$
- Syntax for expressions in  $Cast$
- Composition  $Cast \sqcup cFJ$  is the union of rules
- Easy – only one exception to be considered shortly

$$\boxed{E : (C) E ;} \sqcup \boxed{
 \begin{array}{l}
 E : \quad x \\
 | \quad E.f \\
 | \quad E.m(\overline{E}) \\
 | \quad new\ C(\overline{E}) ;
 \end{array}
 } = \boxed{
 \begin{array}{l}
 E : \quad x \\
 | \quad E.f \\
 | \quad E.m(\overline{E}) \\
 | \quad new\ C(\overline{E}) \\
 | \quad (C) E ;
 \end{array}
 }$$

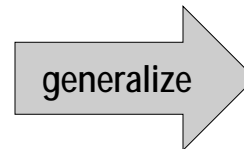
# Modifying Syntax

- Requires foresight to know how productions may be changed by other features
  - engineering result from domain analysis
  - no different than OO refactorings that prepare source code for extensions
  - visitor, framework, strategy patterns

```

E :   x
    |   E.f
    |   E.m( $\bar{E}$ )
    |   new C( $\bar{E}$ ) ;
    
```

cFJ expression syntax



VP definitions

```

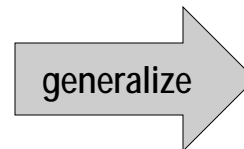
E :   x
    |   E.f
    |   TPm E.m( $\bar{E}$ )
    |   new (TPt C) ( $\bar{E}$ ) ;
TPm :  $\epsilon$  ;
TPt :  $\epsilon$  ;
    
```

variation points (VPs)

```

E : (C) E ;
    
```

Cast syntax

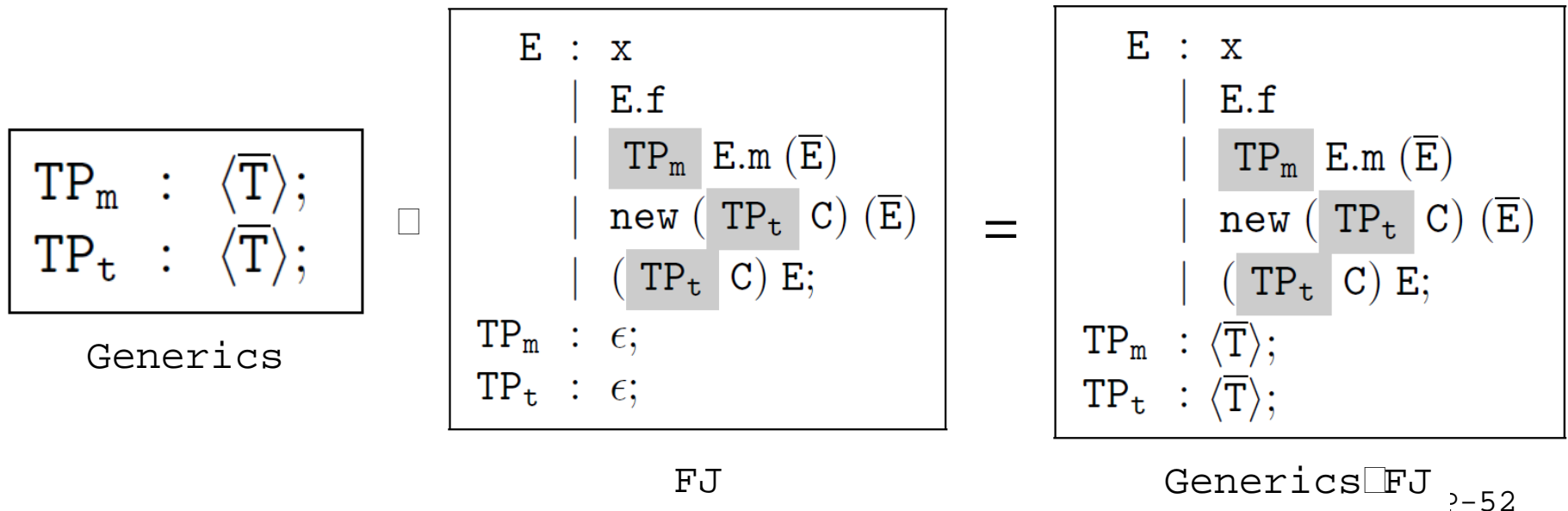


```

E : (TPt C) E ;
    
```

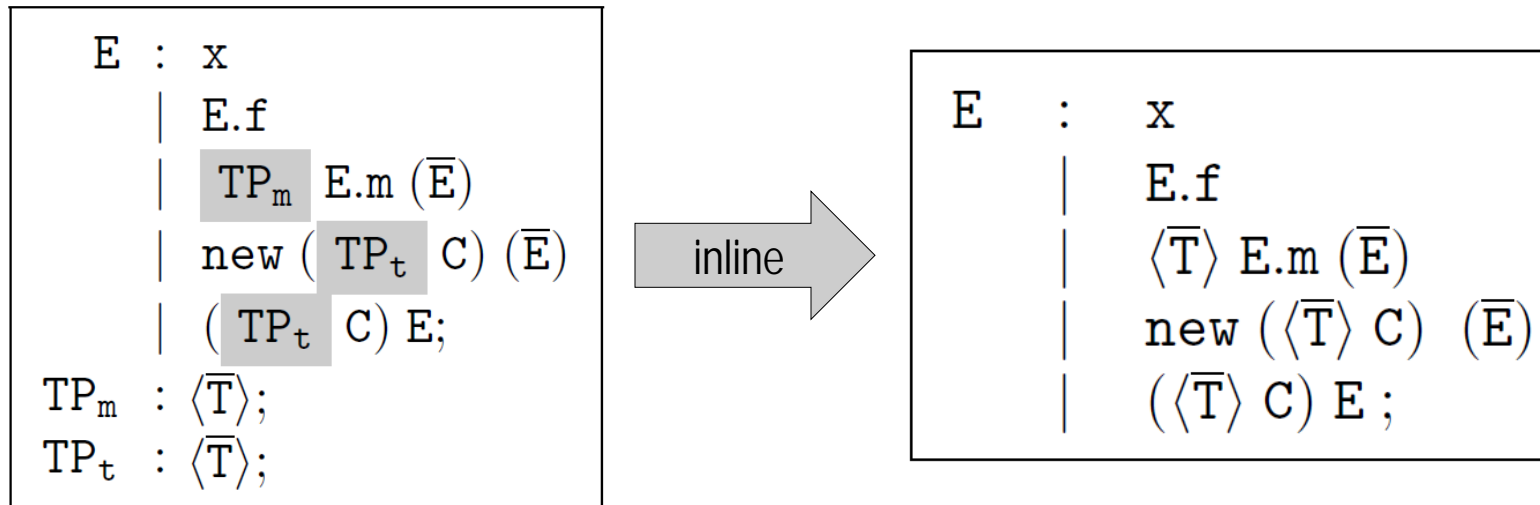
# Composition

- Syntax for original FJ = Cast  $\square$  cFJ
- Syntax for Generics
- Syntax for Generics  $\square$  FJ
- Exception (mentioned earlier) – replace default VP definition



# Inlining

- At the end of a composition process, VP definitions can be inlined to simplify result



- Typically, inlining yields what you would have written by hand
- This is one way how we check if feature compositions are “correct”

other representations are handled no differently – such as:

# REDUCTION AND TYPING RULES

# Adding Rules

- Typing rules for  $cFJ$  expressions
- Typing rule added by  $\text{Cast}$
- Composition  $\text{Cast} \sqcap cFJ$  is the union of these rules

$$\boxed{\frac{\Gamma \vdash e_0 : D \quad D <: C}{\Gamma \vdash e_0 : C} \text{ (T-UCAST)}} \sqcap \boxed{\begin{array}{c} \frac{\text{fields}(C) = \bar{V} \bar{f} \quad \Gamma \vdash \bar{e} : \bar{U} \quad \bar{U} <: \bar{V}}{\Gamma \vdash \text{new } C(\bar{e}) : C} \text{ (T-NEW)} \\ \vdots \end{array}} = \boxed{\begin{array}{c} \frac{\text{fields}(C) = \bar{V} \bar{f} \quad \Gamma \vdash \bar{e} : \bar{U} \quad \bar{U} <: \bar{V}}{\Gamma \vdash \text{new } C(\bar{e}) : C} \text{ (T-NEW)} \\ \vdots \\ \frac{\Gamma \vdash e_0 : D \quad D <: C}{\Gamma \vdash e_0 : C} \text{ (T-UCAST)} \end{array}}$$

$\text{Cast}$

$cFJ$

$\text{Cast} \sqcap cFJ$  -55

# Modifying Rules

- Requires VPs to be defined
- Typing rules for  $cFJ$  expressions
- Generalize by adding VPs
- VPs have more sophisticated meaning

$$\frac{\text{fields}(C) = \bar{V} \bar{f} \quad \Gamma \vdash \bar{e} : \bar{U} \quad \bar{U} <: \bar{V}}{\Gamma \vdash \text{new } C(\bar{e}) : C} \quad (\text{T-NEW})$$

⋮

generalize

$$\frac{\text{WF}_c(D, \text{TP}_t C) \quad \text{fields}(\text{TP}_t C) = \bar{V} \bar{f} \quad D; \Gamma \vdash \bar{e} : \bar{U} \quad D \vdash \bar{U} <: \bar{V}}{D; \Gamma \vdash \text{new}(\text{TP}_t C)(\bar{e}) : \text{TP}_t C} \quad (\text{T-NEW})$$

$$\frac{\mathbb{T}}{\text{WF}_c(\epsilon, C, \epsilon)} \quad D := \epsilon$$

⋮

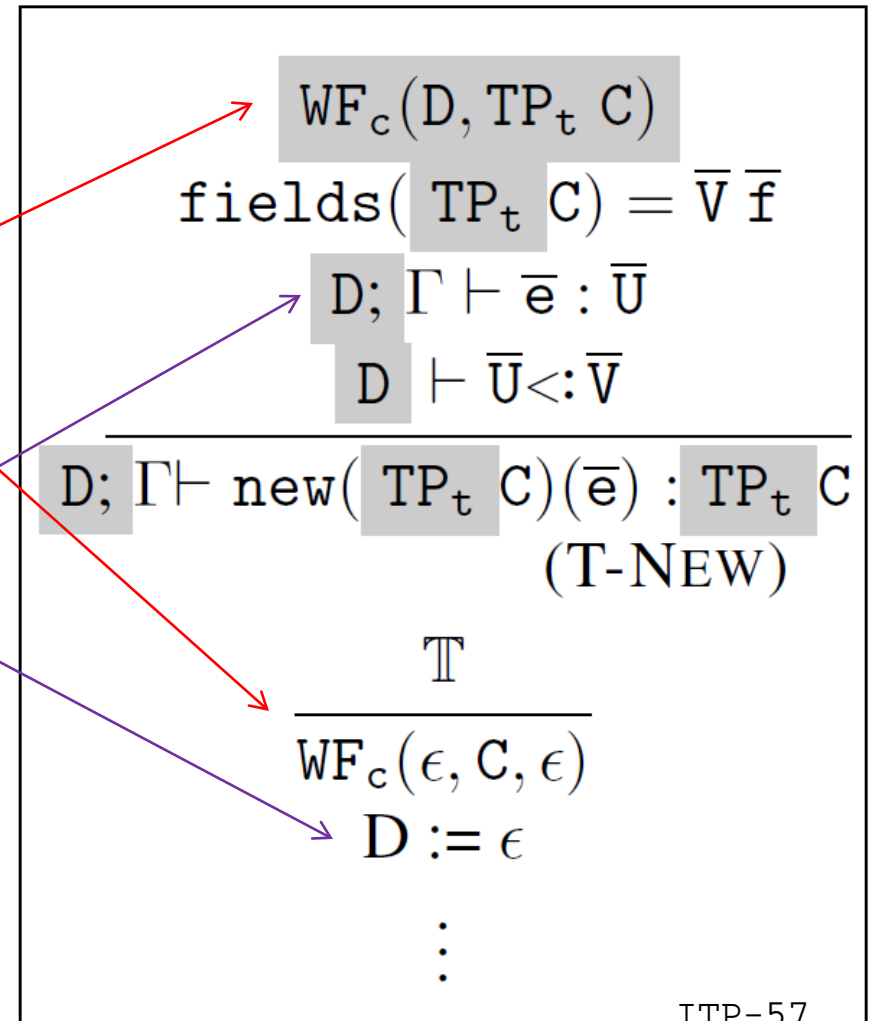
# Semantics of VPs

- Three kinds of VPs:

- predicates that extend the premise of a rule  
(true by default)

- relational holes which extend a judgment's signature (empty by default)

- functions that transform existing premises and conclusions  
(identity function by default)





# Composition (as Before)

- Typing rules for  $cFJ$
- Typing rules for `Generics` (replaces default declarations for  $WF_c$  and  $D$ )
- Typing rules for `Generics`  $\square$   $cFJ$

$$\boxed{
 \frac{\Delta \vdash \langle \bar{T} \rangle C \text{ ok}}{WF_c(\Delta, \langle \bar{T} \rangle C)}
 }
 \quad \square \quad
 \boxed{
 \frac{
 \begin{array}{c}
 WF_c(D, TP_t C) \\
 \text{fields}(TP_t C) = \bar{V} \bar{f} \\
 D; \Gamma \vdash \bar{e} : \bar{U} \\
 D \vdash \bar{U} <: \bar{V}
 \end{array}
 }{
 D; \Gamma \vdash \text{new}(TP_t C)(\bar{e}) : TP_t C
 }
 }
 \quad = \quad
 \boxed{
 \frac{
 \begin{array}{c}
 WF_c(D, TP_t C) \\
 \text{fields}(TP_t C) = \bar{V} \bar{f} \\
 D; \Gamma \vdash \bar{e} : \bar{U} \\
 D \vdash \bar{U} <: \bar{V}
 \end{array}
 }{
 D; \Gamma \vdash \text{new}(TP_t C)(\bar{e}) : TP_t C
 }
 }
 }$$

$$D := \Delta$$

$$\frac{\mathbb{T}}{WF_c(\epsilon, C, \epsilon)}$$

$$D := \epsilon$$

$$\frac{\Delta \vdash \langle \bar{T} \rangle C \text{ ok}}{WF_c(\Delta, \langle \bar{T} \rangle C)}$$

$$D := \Delta$$

ITP-58

finally!

# THEOREMS AND PROOFS

# Theorems

- A “general” theorem in cFJ with VPs and default definitions
- Theorem “adapts” to VP instantiations of Generic

$$\frac{\begin{array}{c} TP_t : \epsilon; \quad TP_m : \epsilon; \quad \mu : \epsilon; \\ D := \epsilon \\ \hline T \\ \hline WF_{ne}(\epsilon, C) \end{array}}{T}$$

LEMMA 4.1 (Well-Formedness). If  $\bar{V} \rightarrow V$  and  $mbody(m, C)$  then there exist  $N$  and  $S$  such that  $\vdash C \leq N$  and  $N \vdash e : S$ .

cFJ

$$\frac{\begin{array}{c} TP_t : \bar{T}; \quad TP_m : \bar{T}; \quad \mu : \langle \bar{Y} \triangleleft \bar{P} \rangle; \\ D := \Delta \\ \hline \Delta \vdash \bar{U} \text{ ok} \quad \Delta \vdash \bar{U} \leq [\bar{U}/\bar{Y}] \bar{P} \\ \hline mc(\Delta, \langle \bar{Y} \triangleleft \bar{P} \rangle, \bar{U}) \\ \hline \Delta \vdash S \leq [\bar{U}/\bar{Y}] \bar{U} \end{array}}{T}$$

body). If  $\bar{V} \rightarrow V$  and  $\Delta \vdash \langle \bar{T} \rangle C \text{ ok}$  then there exist  $N$  and  $S$  such that  $\vdash C \leq N$  and  $N \vdash e : S$ , where  $\Delta \vdash \bar{U} \text{ ok}$  and  $\Delta \vdash S \leq [\bar{U}/\bar{Y}] V$  and

Generic cFJ

same for proofs  
but now elevate to  
semantic composition...

# Semantic Composition

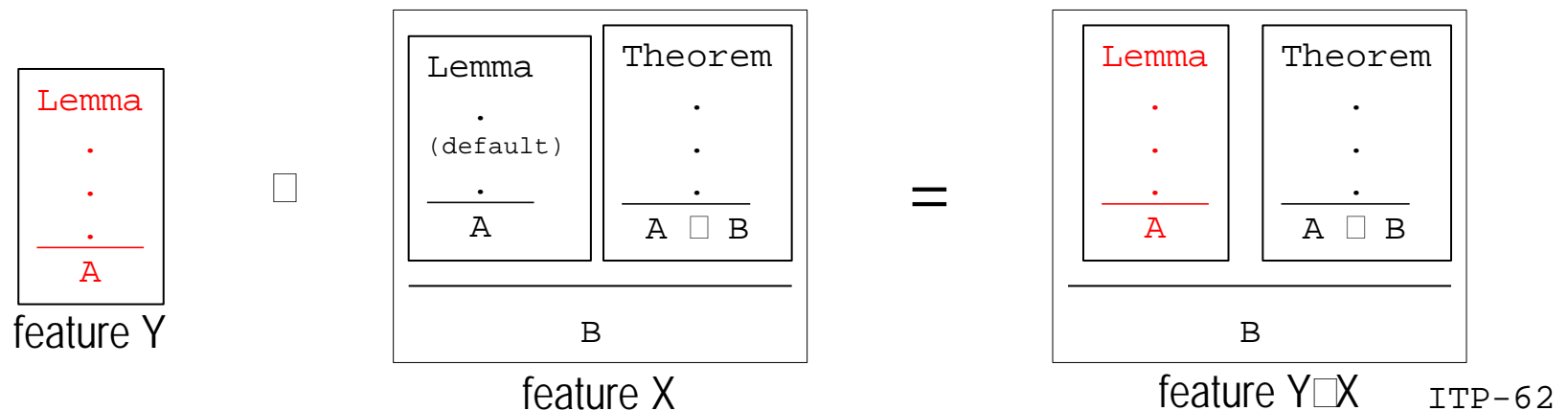
that guarantees the correctness of proofs

- When VPs are used in theorems and proofs, we define properties that must be satisfied by any VP plug-in
  - stated as additional assumptions with default lemma(s)
- Allows a general theorem to be proven, independent of features that might “plug-in” specific definitions for its VPs
  - *in effect, the proof assumes a general behavior for all possible VP instantiations*
- Obligation: any feature that “plugs-in” a VP definition must supply a proof that the properties assumed by the general theorem are satisfied

# Semantic Composition

that guarantees the correctness of proofs

- In effect, the assumptions of a general theorem form an explicit interface against which a proof is written.
- General theorem does not have to be recertified, reuse as is
- Plug-in theorems do not need to be recertified, reused as is
- Must certify that general assumptions hold for plug-ins



VP for Ben



# ENCODING FEATURE MODULES IN COG

# Coq Encodings

- Syntax, operational semantics, and typing rules are written as standard inductive data types in Coq. Proofs are then written over these encodings
- Encoding of syntax:

```
E : x
    | E.f
    |  $\text{TP}_m$  E.m ( $\bar{E}$ )
    | new ( $\text{TP}_t$  C) ( $\bar{E}$ )
    | ( $\text{TP}_t$  C) E;
TPm :  $\epsilon$ ;
TPt :  $\epsilon$ ;
```

Syntax Notation

mapping

```
Definition TP_m := unit.
Definition TP_t := unit.

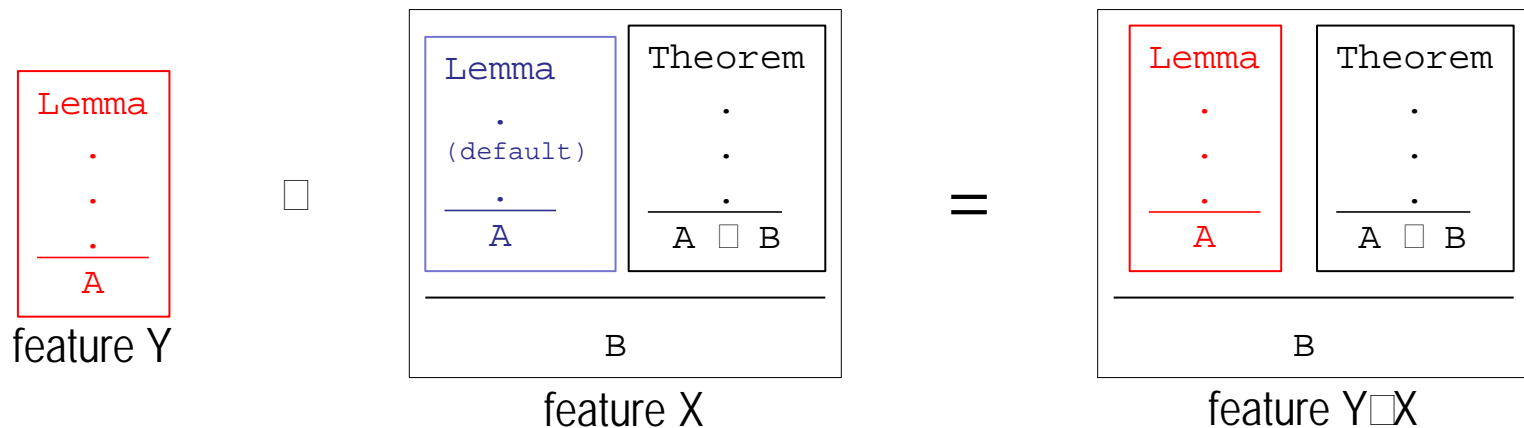
Inductive C : Set :=
| ty : TP_t → Name → E.

Inductive E : Set :=
| e_var : Var → E
| fd_access : E → F → E
| m_call : TP_m → E → M → List E → E
| new : C → List E → E.
```

Coq Encoding

# Semantic (not Syntactic!) Composition

- So far, we defined composition **syntactically**
- Fine for definitions, but how does this work with proofs?
- Could do syntactic updates on proof terms



- Specifying VPs on large proof trees is difficult
- Have to recheck resulting term for each variant
- Need a more semantic notion of composition!



# Semantic (not Syntactic) Composition

- Use abstraction mechanisms built into Coq
- Definitions are parameterized on **variation points**
- Modules provide **instantiations**
- Composition is simply instantiation

```
Definition TP_m := unit.  
Definition TP_t := unit.  
  
Inductive C : Set :=  
| ty : TP_t → Name → E.  
  
Inductive E : Set :=  
| e_var : Var → E  
| fd_access : E → F → E  
| m_call : TP_m → E → M → List E → E  
| new : C → List E → E.
```

```
Variable TP_m : Set.  
Definition TP_m_def := unit.  
Variable TP_t : Set.  
Definition TP_t_def := unit.  
  
Inductive C : Set :=  
| ty : TP_t → Name → E.  
  
Variable E : Set.  
Inductive E_def : Set :=  
| e_var : Var → E_def  
| fd_access : E → F → E_def  
| m_call : TP_m → E → M → List E → E_def  
| new : C → List E → E_def.
```

# Semantic (not Syntactic) Composition

- Parameterized definitions enable variation points in proofs
- VPs are opaque to Coq
  - need to make assumptions about their behavior to complete proofs
  - assumptions are the **proof variation points**
  - proof composition is again instantiation
  - allows each module to be checked independently

```
Variable TLookup_app : forall gamma delta X ty,  
  TLookup gamma X ty →  
  TLookup (app_context gamma delta) X ty.  
  
Lemma GJ_Weaken_Subtype_app : forall gamma S T  
  (sub_S_T : GJ_subtype gamma S T),  
  Weaken_Subtype_app_P _ _ _ sub_S_T.  
  cbv beta delta; intros; apply GJ_subtype_Wrap.  
  inversion sub_S_T; subst.  
  econstructor; eapply TLookup_app; eauto.  
Qed.
```

# Feature Modules in Coq

- One Coq file per feature, which encapsulates all pieces of that feature
- Each file is independently certified by Coq
  - To compose modules, a new file is created
  - Definitions and proofs are composed one at a time by instantiating variation points in definitions from features
- Coq simply checks that each proof's assumptions are satisfied
  - Effectively an interface check
  - No need to recheck proof terms from the modules

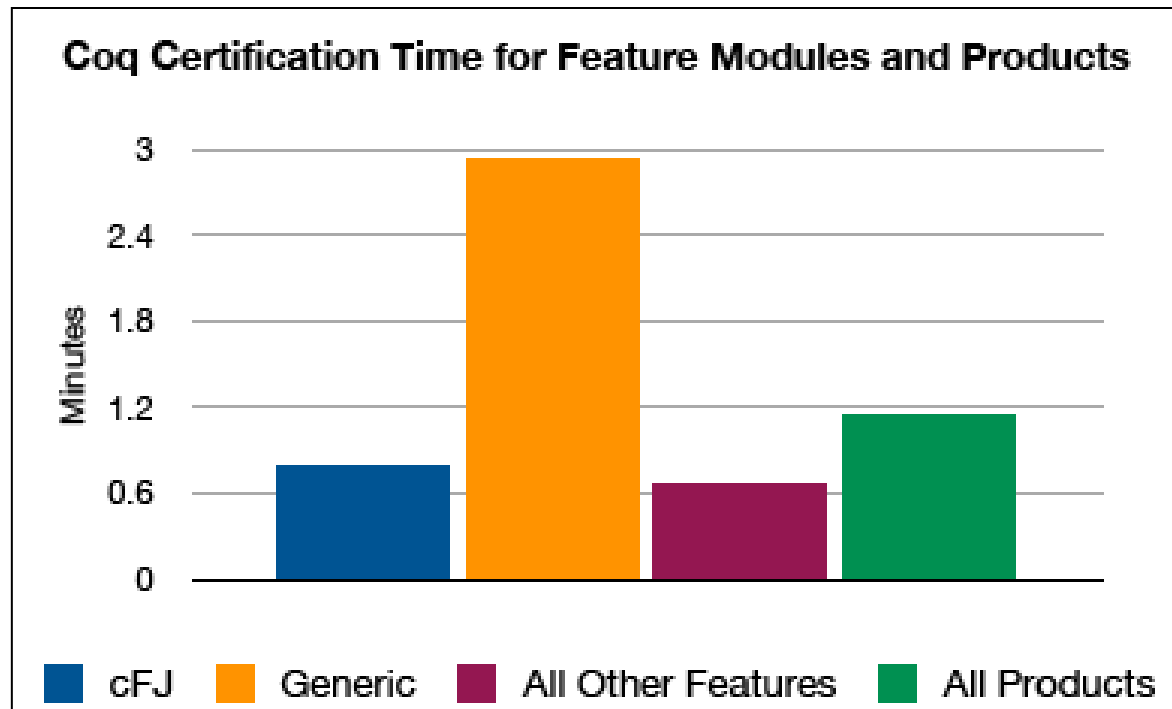
# Feature Module Statistics

- One Coq file per module that encapsulates all representations

| Module            | Description                        | Length of Coq Scripts |
|-------------------|------------------------------------|-----------------------|
| cFJ               | core Featherweight Java            | 2612 LOS              |
| Cast              | casts                              | 463 LOS               |
| Interface         | interfaces                         | 499 LOS               |
| Generic           | generics                           | 6740 LOS              |
| Generic#Cast      | generic and interface interactions | 1632 LOS              |
| Generic#Interface | generic and cast interactions      | 296 LOS               |

# Performance

- Once proofs in each feature module have been certified, they do not need to be rechecked for a target language
- Practical effect: certification time for feature modules is non-trivial
- Certifying **all products** in our SPL approx. same time as required by **cFJ** module



FUTURE WORK

# Enhanced Support in Coq

- Relying on parameterization for feature composition has clear benefits:
  - Everything works “out of the box”: same level of assurance as anything in Coq
  - Separate verification of feature modules means we don’t have to recheck proofs for each product
- But there are drawbacks:
  - Composition scripts are tediously built piece-by-piece
  - Adding a new feature requires modifying existing features to allow for extension:
    - Recursion needs to be opened and VPs added to inductive data types
    - Every proof over an extended type has to be reengineered

# Enhanced Support in Coq

- We are looking at extending Coq to better support feature composition
- Ideally, a feature module can be designed without extension in mind
- Subsequent feature modules can extend its definitions with new cases or variations
- Given an extension and an existing proof, a feature module provides the necessary pieces to build a new proof
  - typing rules of CIC indicate where the proof extensions need to occur
- A feature-module-level composition operator builds the complete set of definitions and proofs from a product specification automatically



# Holy Grail



- Safe Composition
- A general structural analysis certifies that *all programs of an SPL are type correct*
  - uses a SAT solver and feature model to examine all legal combinations of features to verify type safety properties of all programs in an SPL
  - **much** faster than building and verifying each product separately
- Believe a similar analysis can be done to certify correctness of all Coq products in an SPL
  - won't have to generate and then certify theorems for each product
  - know ahead of time that the process is correct

VP for Don



# CONCLUSIONS

# Conclusions

- Mechanically verifying artifacts using theorem provers is hard work
- Compounded when verifying all members of a product line
- Features are a natural way to decompose a family of programs
- Decomposing proofs along feature boundaries enables a natural reuse of proofs
  - same for other representations as well
- Follows a typical way in which language definitions (syntax, semantics, type system, proofs) evolve over time
- We use simple design and implementation techniques to structure a product line of theorems and their proofs, requiring:
  - engineering features so that they “fit together”
  - mathematical foundation of feature structures

# Proof of Concept

- Applied ideas to an SPL of Featherweight Java, using standard facilities in Coq to mechanically check proofs of progress and preservation for composed languages
- A feature-based approach supports a structured evolution of languages from a simple core to a fully-featured language
- Doing so transforms a mechanized formalization of a language from a rigorous check of correctness into an important way to reuse definitions and proofs across a family of related languages
- We conjecture that our success can be replicated in other domains, and herein lies future work. We welcome your thoughts and suggestions.

# Thank You!