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

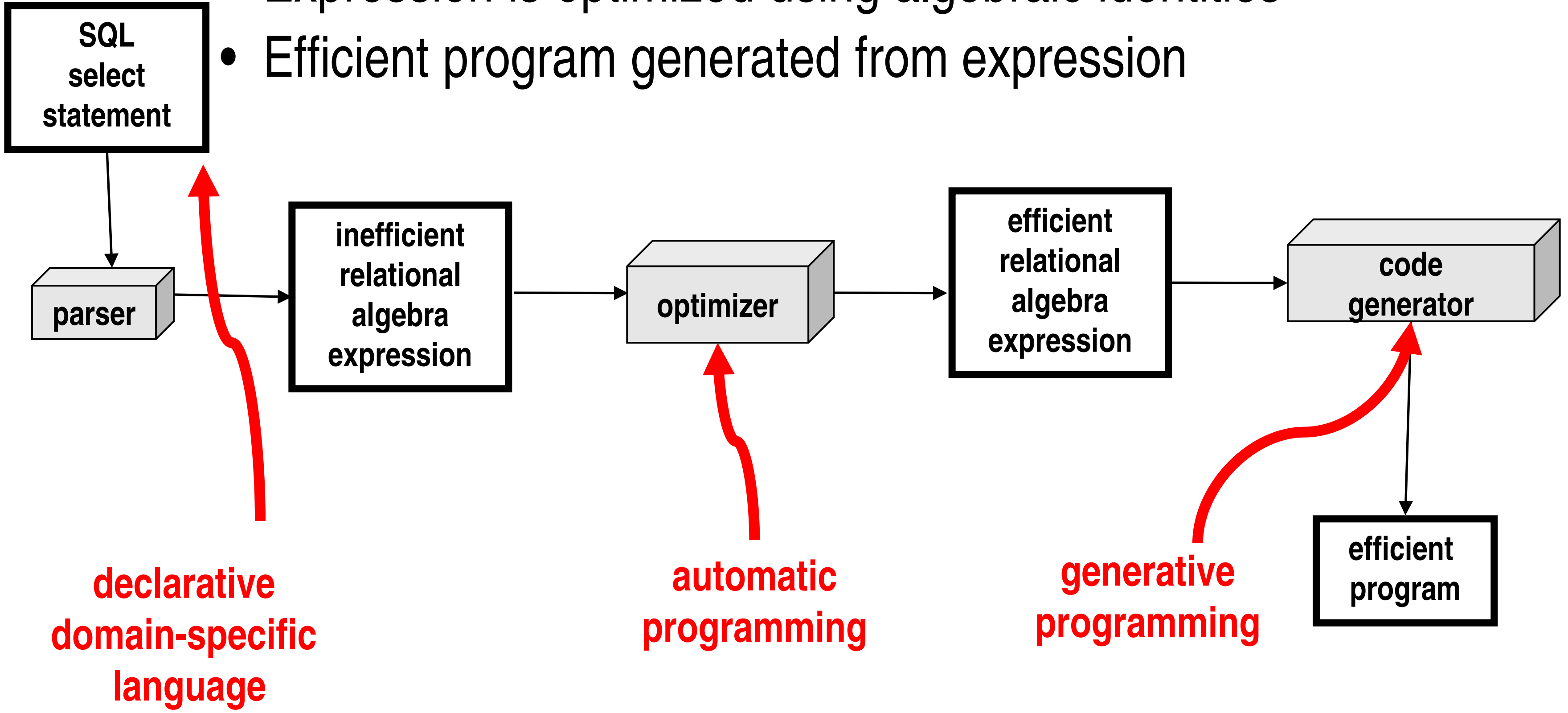
- My work focused on **software product lines (SPLs)**
  - set of related programs that are differentiated by “features”
  - **feature** is an “increment in program functionality”
  - different compositions of features yield different programs

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


SQL select statement
parser
inefficient relational algebra expression
optimizer
efficient relational algebra expression
code generator
efficient program
declarative domain-specific language
automatic programming
generative programming

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

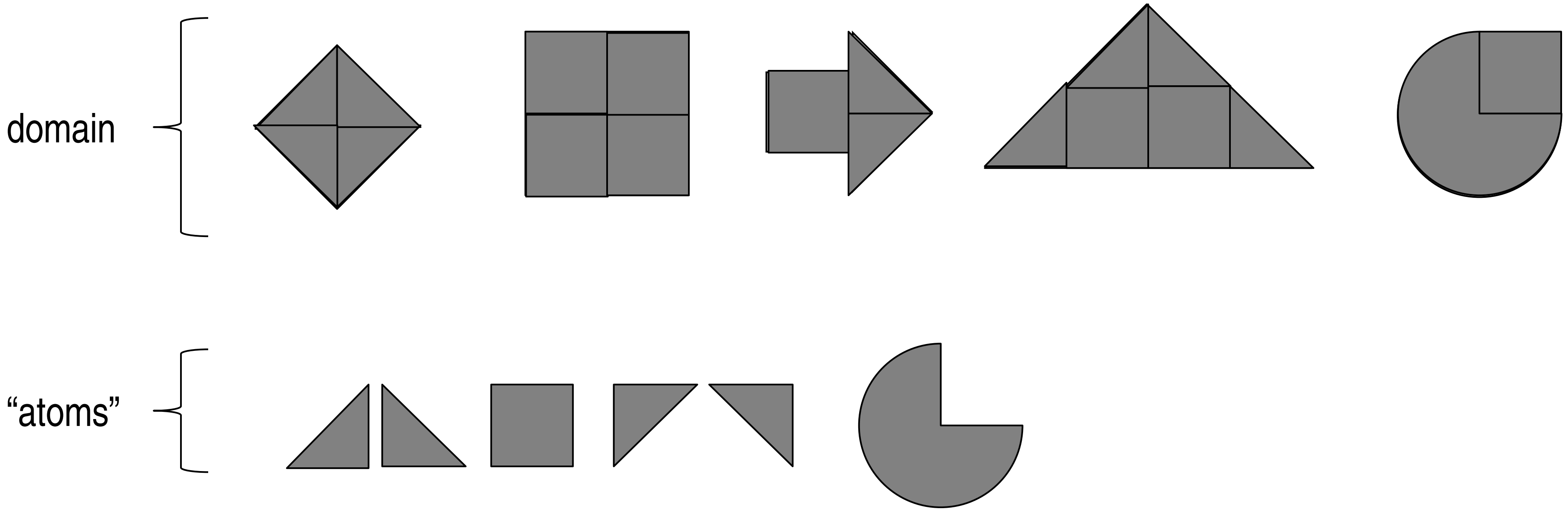
- Explain how RQO paradigm generalizes to SPLs
- Also show how proofs scale from a single program to families of programs – big win
- Within an algebraic framework of automated program generation and SPLs – general approach

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

$$\begin{tikzcd}
\square \arrow[r, "f"] \arrow[ddr, "g"'] & P_0 \arrow[r, "h"] \arrow[dr, "i"] & P_1 \arrow[r, "k"] & P_2 \\
& & P_2 \arrow[r, "m"] \arrow[dr, "n"] & P_4 \\
& P_6 \arrow[r, "h"] \arrow[dr, "i"] \arrow[ddr, "j"] & P_7 & P_5 \\
& & P_8 \arrow[r, "m"] & P_9 \\
& & P_A &
\end{tikzcd}$$

$P_2 = k\square h\square f\square\square$

actually categories, but this is not relevant to this talk

Example: a 4-Program SPL

- Elementary product line of Java calculators

$$\begin{tikzcd}
\square \arrow[r, "\mathtt{base}"] & \mathbf{P}_0 \arrow[r, "\mathtt{sub}"] \arrow[dr, "\mathtt{format}"] & \mathbf{P}_1 \arrow[r, "\mathtt{format}"] & \mathbf{P}_2 \\
 & & \mathbf{P}_3 &
\end{tikzcd}$$

$$\begin{tikzcd}
\square \arrow[r, "\text{base}"] & P_0 \arrow[r, "\text{sub}"] \arrow[dr, "\text{format}"'] & P_1 \arrow[r, "\text{format}"] & P_2 \\
 & & P_3 &
\end{tikzcd}$$

```
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

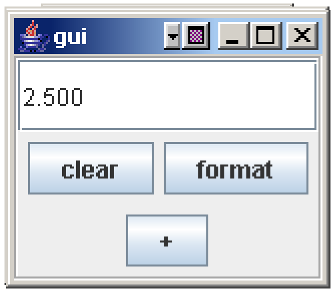
new fields

new fields

extend existing methods

extend existing methods

new methods



$\text{format} \square \ \text{sub} \square \ \text{base} \square\square = P_3$

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

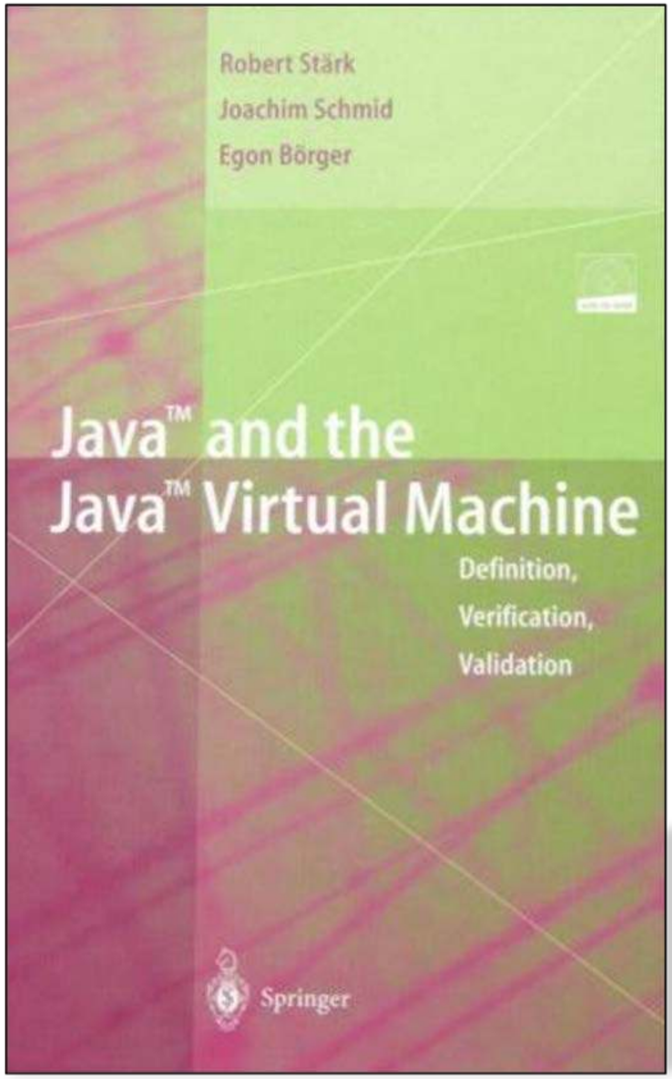
- Using features has right look and feel
    - standard idea in software product lines
    - features as transformations is key to a modern approach

    - 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

JBOOK

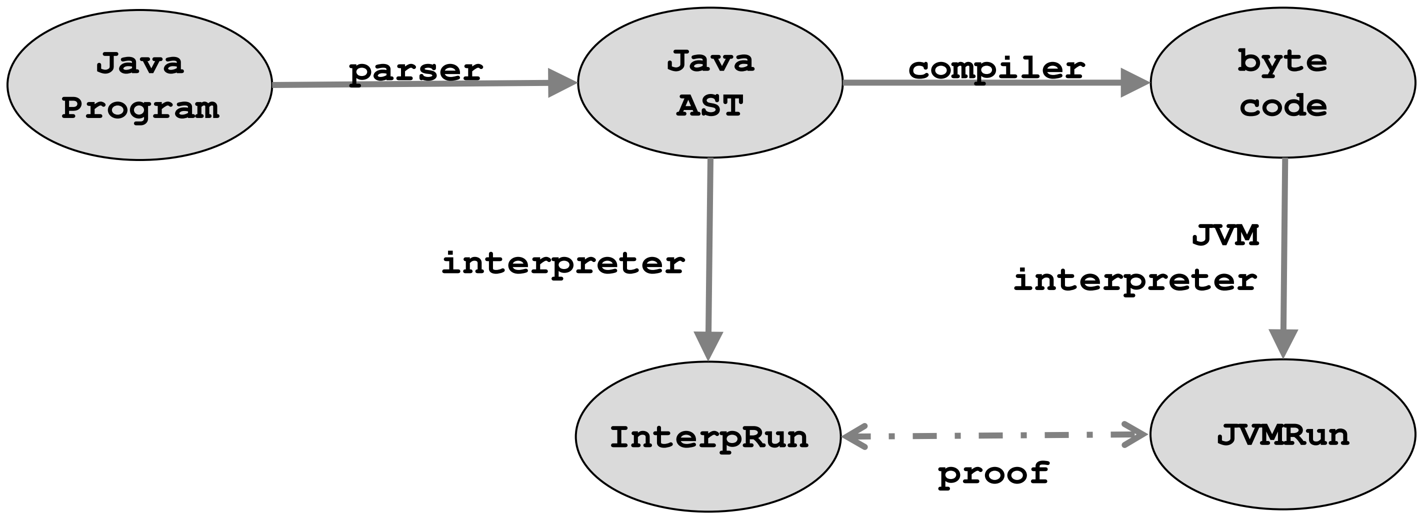

Robert Stärk
Joachim Schmid
Egon Börger

Java™ and the Java™ Virtual Machine

Definition, Verification, Validation

Springer

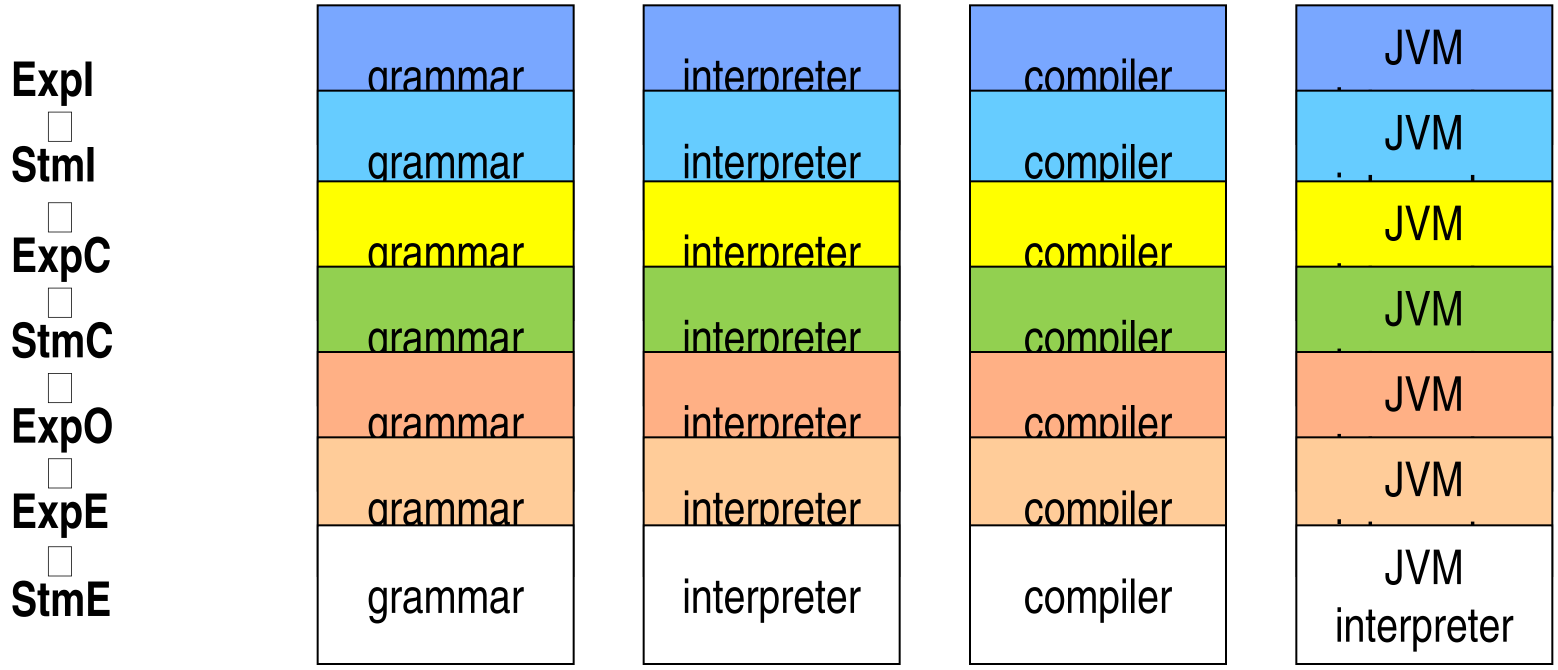
Structure of JBook



- 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...

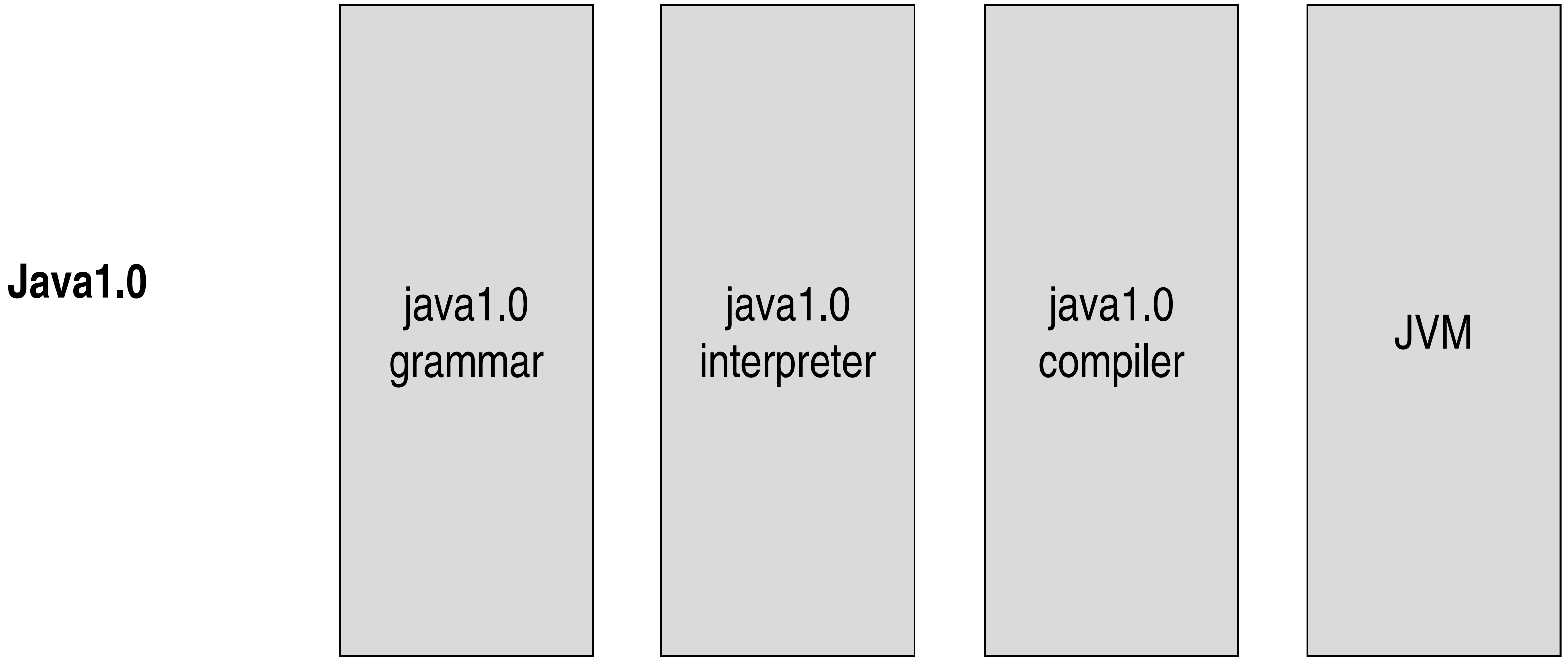
Overview of JBook

- 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



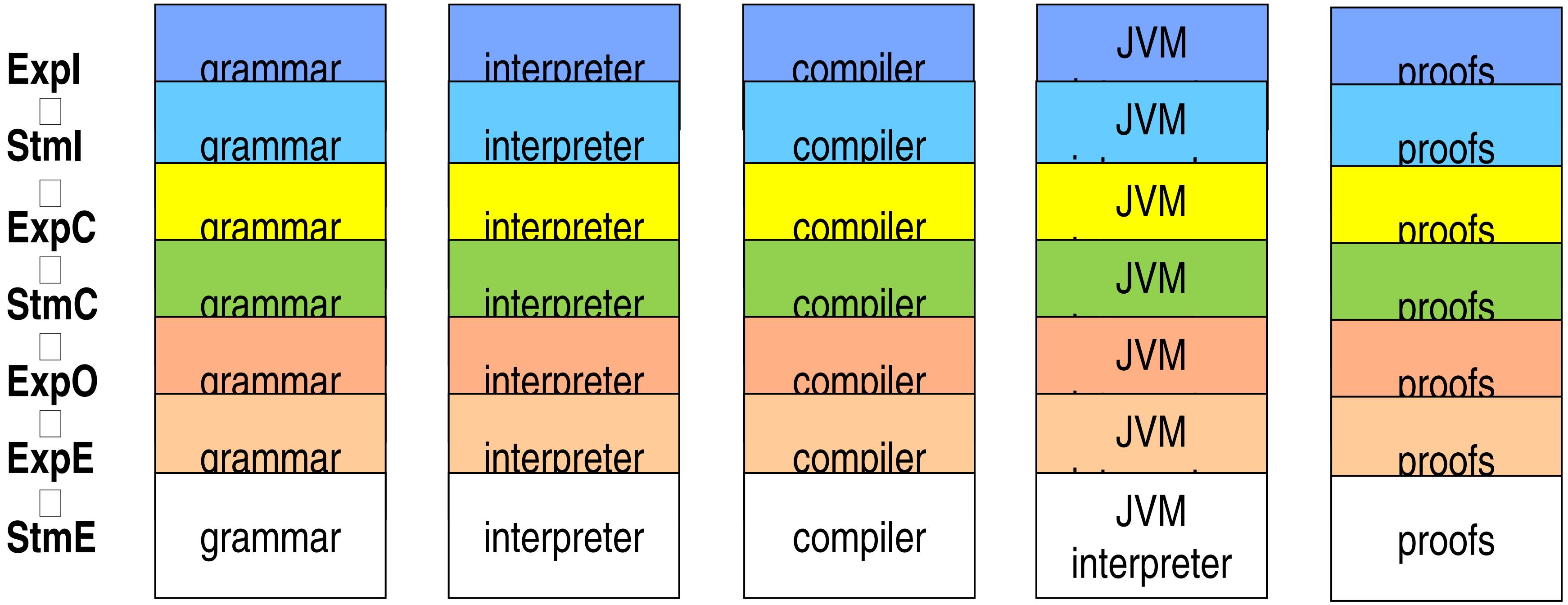
Overview of JBook

- 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
- <u>Only when the Java 1.0 definitions were complete were the proofs constructed</u>



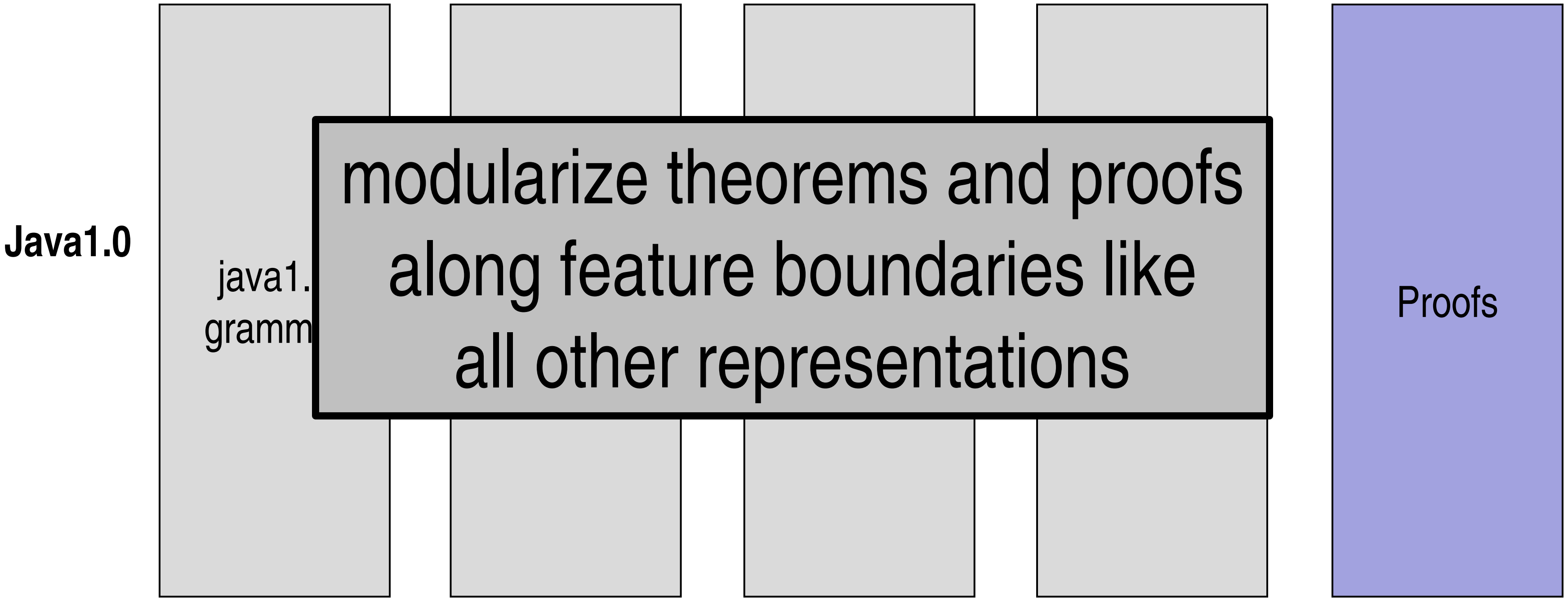
Features Update All Representations of a Program Lock-Step

- 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...

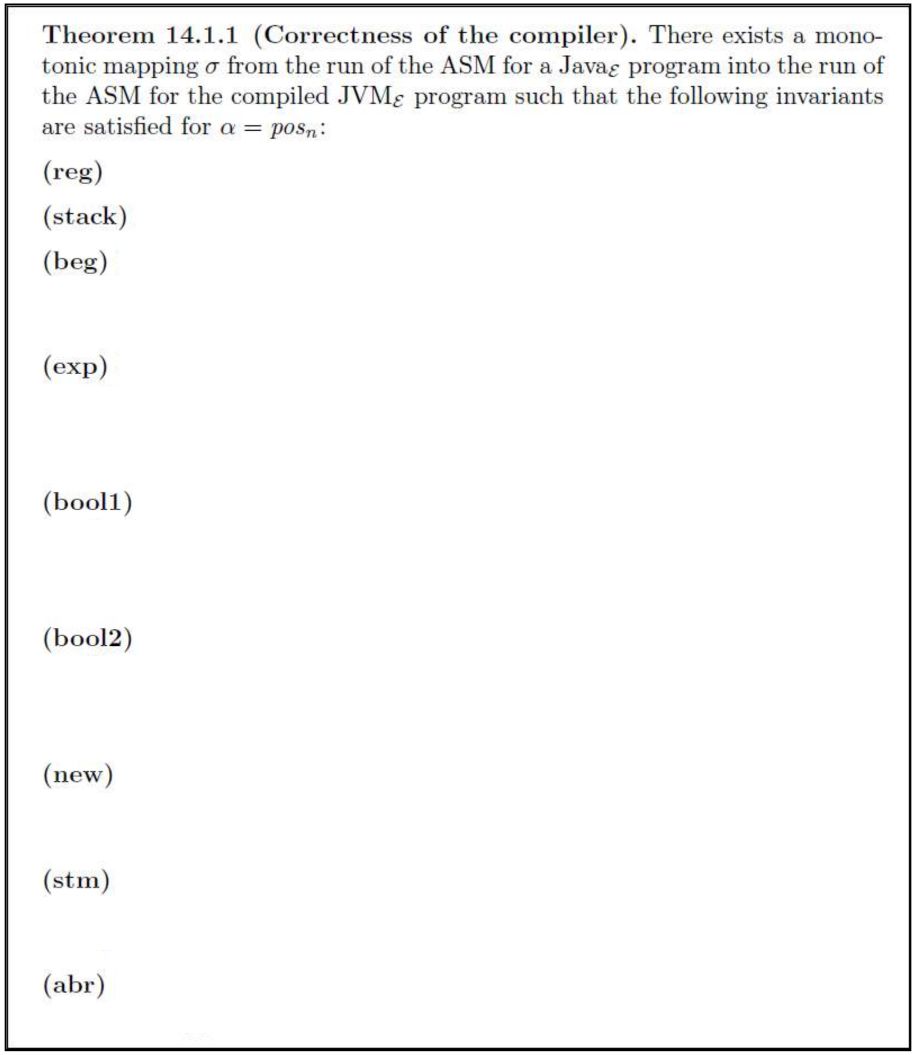
- Theorems and proofs could be developed incrementally from features as well, lock-step like other representations – structurally treat them no differently



modularize theorems and proofs along feature boundaries like all other representations

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 σ from the run of the ASM for a $\text{Java}_{\mathcal{E}}$ program into the run of the ASM for the compiled $\text{JVM}_{\mathcal{E}}$ program such that the following invariants are satisfied for $\alpha = pos_n$:

(reg)

(stack)

(beg)

(exp)

(bool1)

(bool2)

(new)

(stm)

(abr)

Statement of Correctness

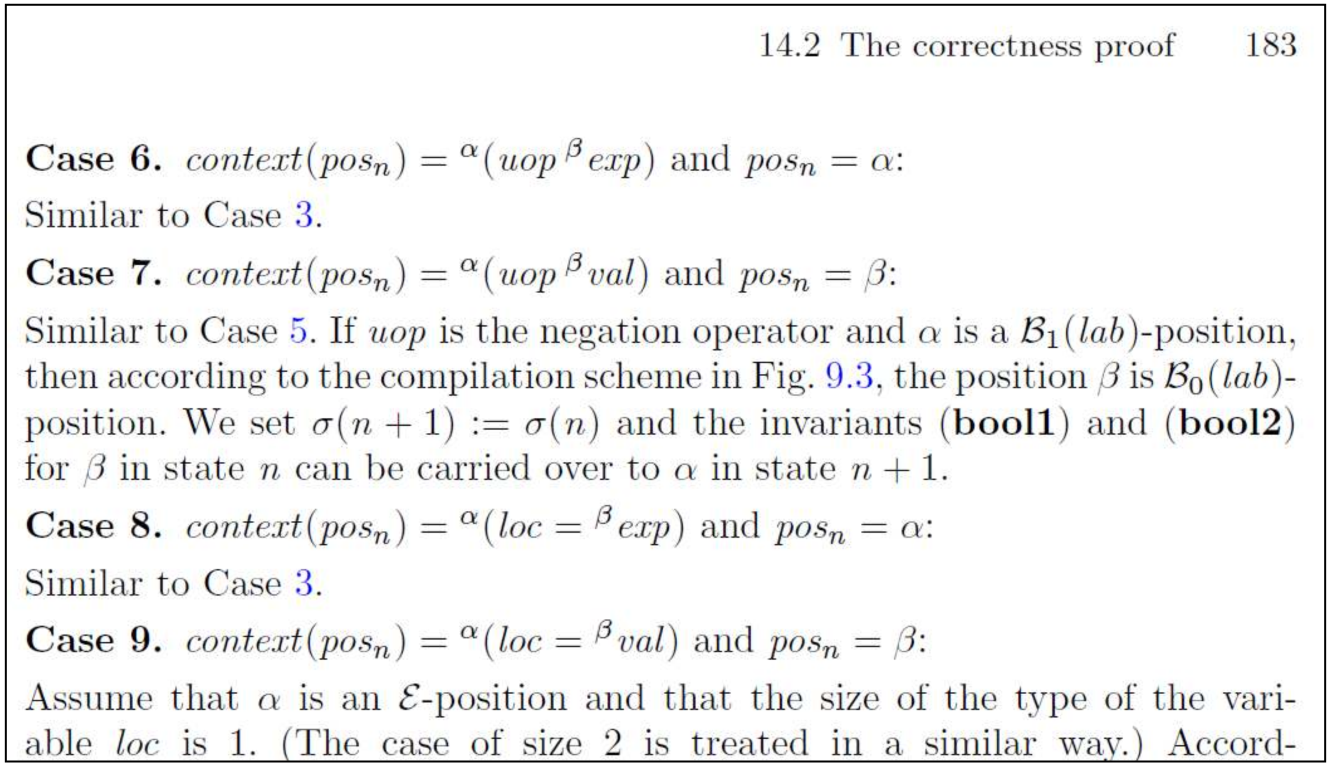
Theorem 14.1.1 (Correctness of the Compiler). There exists a monotonic mapping □ from the run of the ASM for a Java program into the run of the ASM for the compiled JVM program such that the following invariants are satisfied:

(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 α is a $\mathcal{B}_1(lab)$-position, then according to the compilation scheme in Fig. 9.3, the position β is $\mathcal{B}_0(lab)$-position. We set $\sigma(n+1) := \sigma(n)$ and the invariants (**bool1**) and (**bool2**) for β in state n can be carried over to α 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 α 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.
As features are composed, the theorem is elaborated with
new invariants, the proof
is extended with new cases and
elaborations of existing cases.

[illegible]	T
[illegible]	[illegible]
Java1.0	83

Reaction…

- JBook proofs were manually created
- Need to be mechanically verified

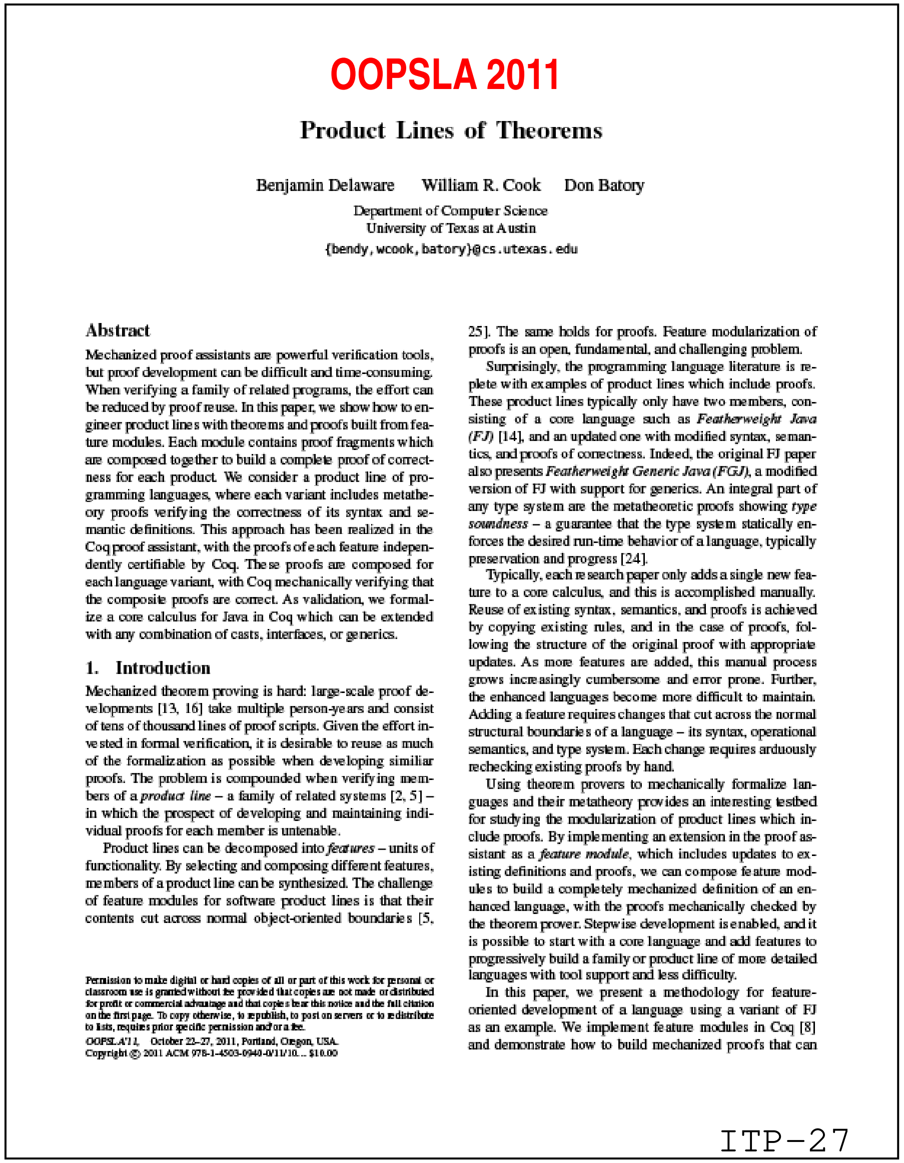
- Our conjecture was that theorems + proofs could be generated just like other representations of programs in an SPL
- Show how our conjecture held with modern tools and approaches
- Starting point for this work

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 similiar 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.

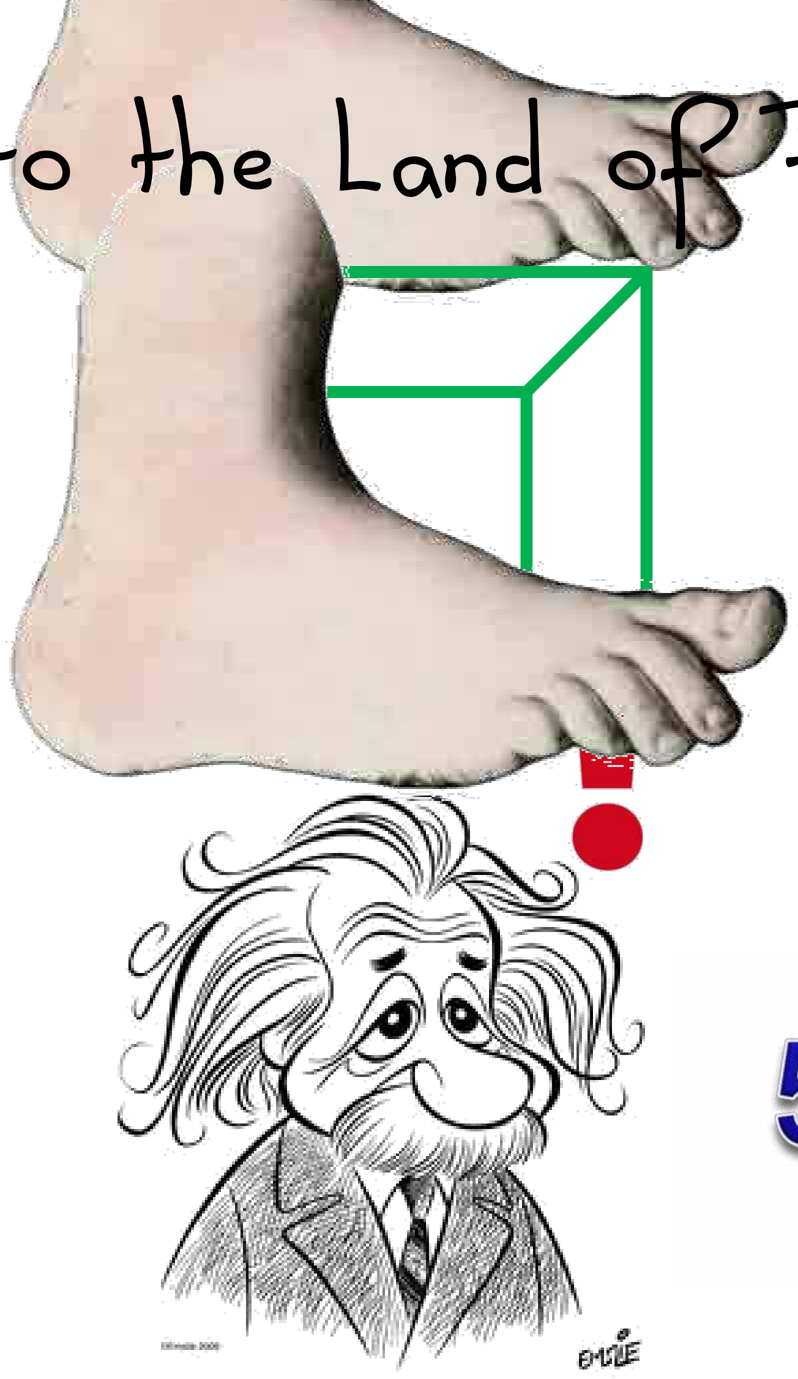
Surprisingly, the programming language literature is replete with examples of product lines which include proofs. These product lines typically only have two members, consisting of a core language such as *Featherweight Java (FJ)* [14], and an updated one with modified syntax, semantics, and proofs of 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

Welcome to the Land of Features!



5 ideas

#1: Features and Domains

- Given a domain `D` of programs to generate, identify the core features that underlie the domain via domain analysis. Domain `D` has the set of features:

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

instead of starting from □

- 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

- Small product line of 4 features:

base

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 □ j □ 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:

$$P = [\, gram_P, src_P, man_P \,]$$

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

$$F = [\, \Delta gram_F, \Delta src_F, \Delta man_F \,]$$

Feature Composition

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

$$FP = F \square P$$

$$= [\ \Delta gram_F,\ \Delta src_F,\ \Delta man_F\ ] \square [\ gram_P,\ src_P,\ man_P\ ]$$

$$= [\ \underbrace{\Delta gram_F \square gram_P}_{\text{grammar of FP}},\ \underbrace{\Delta src_F \square src_P}_{\text{source of FP}},\ \underbrace{\Delta man_F \square man_P}_{\text{manual of FP}}\ ]$$

Our Example

- Base language (cFJ) has multiple representations

base representation	specification
syntax	s_{cFJ}
operational semantics	o_{cFJ}
type system	t_{cFJ}
meta-theory proofs	p_{cFJ}

preservation and progress proofs

- Base language is a 4-tuple:

$$cFJ = [\, s_{cFJ}, o_{cFJ}, t_{cFJ}, p_{cFJ} \,]$$

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 `FJ` is:

$$
\begin{aligned}
&\mathtt{FJ}\\
&= \mathtt{Cast} \,\square\, \mathtt{cFJ}\\
&= [\, \Delta s_{cast}, \Delta o_{cast}, \Delta t_{cast}, \Delta p_{cast} \,] \,\square\, [\, s_{cFJ}, o_{cFJ}, t_{cFJ}, p_{cFJ} \,]\\
&= [\, \underbrace{\Delta s_{cast} \square s_{cFJ}}_{\text{syntax of FJ}}, \underbrace{\Delta o_{cast} \square o_{cFJ}}_{\text{semantics of FJ}}, \underbrace{\Delta t_{cast} \square t_{cFJ}}_{\text{type system of FJ}}, \underbrace{\Delta p_{cast} \square p_{cFJ}}_{\text{theorems and proofs FJ}} \,]
\end{aligned}
$$

one more piece…

#4: FEATURE INTERACTIONS

Feature Interactions

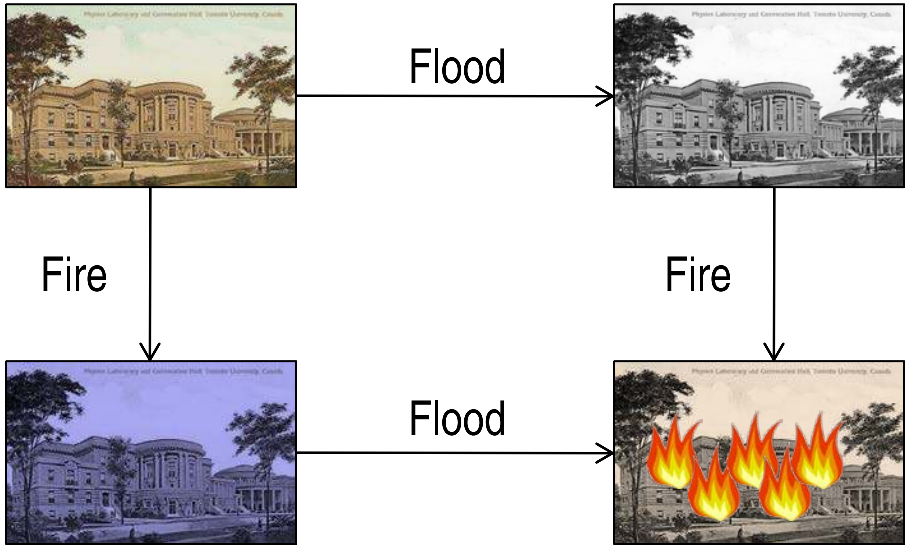
- **Feature interaction (FI)** occurs when two features behave incorrectly together

- **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

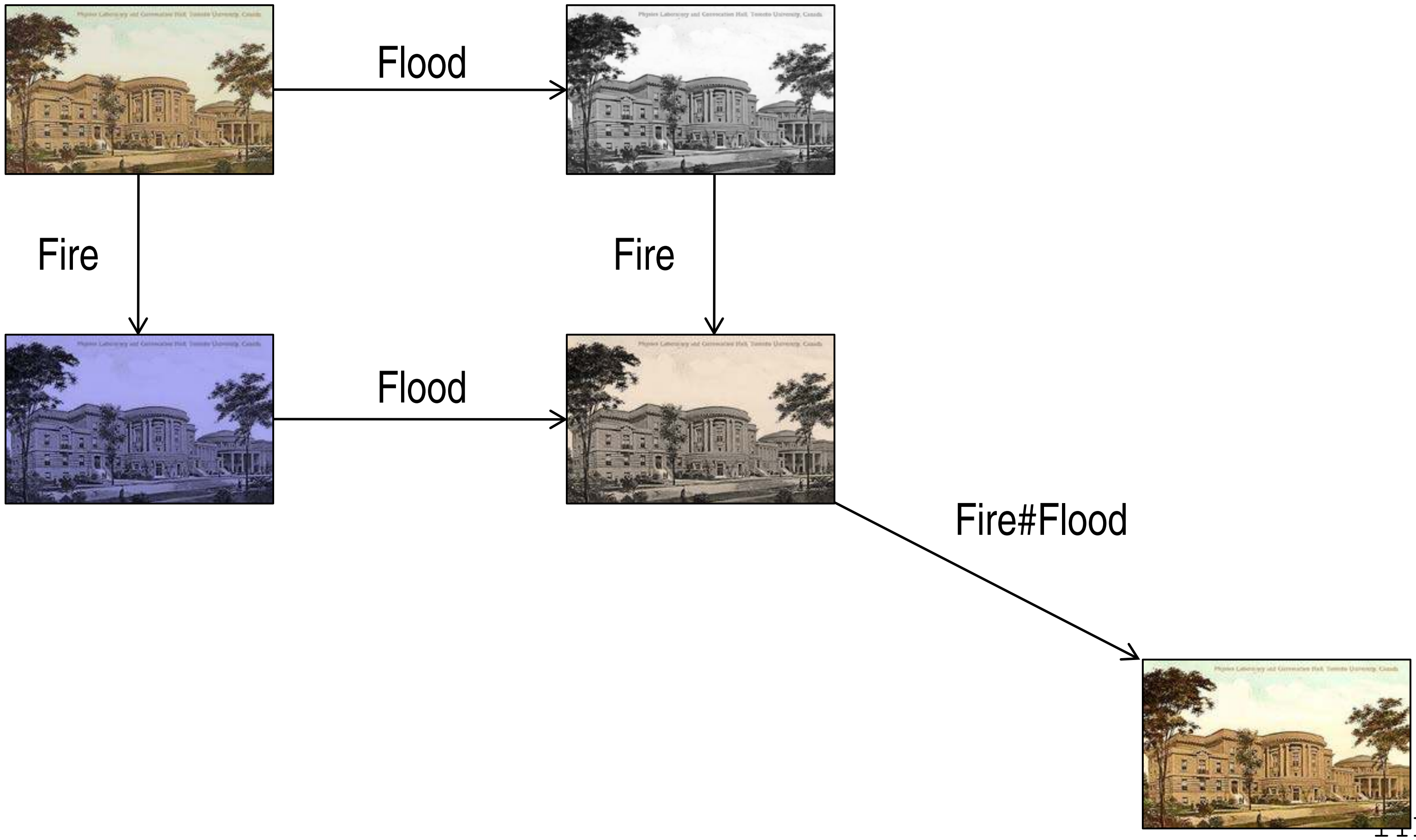
- Flood control – Fire control problem (Kang 2003)
  - isomorphic to feature interaction problems in telephony



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 (□) 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 ‘`k j b`’ is mapped to a expression by a cross-product (not by a dot-product) of its terms

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

- 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 × Interface × cFJ

= Generic # Interface # cFJ □ Generic # Interface □
  Generic # cFJ □ Generic □ Interface # cFJ □ Interface □ cFJ

= 1 □ Generic # Interface □ 1 □ Generic □ 1 □ Interface □ cFJ

= Generic # Interface □ Generic □ Interface □ cFJ
```

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

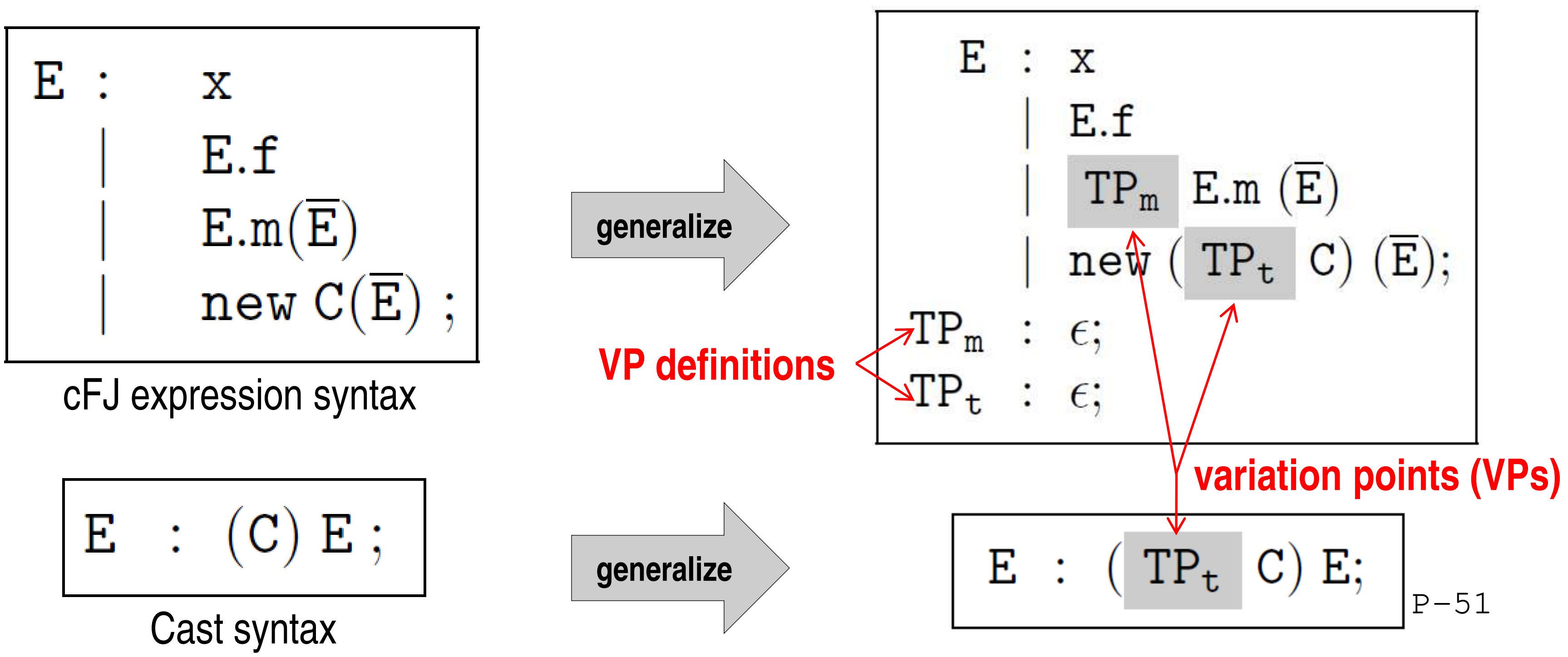
Adding Syntax

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

$$\boxed{E \;:\; (C)\,E\,;} \quad □ \quad \boxed{\begin{array}{lll} E & : & x \\ & | & E.f \\ & | & E.m(\overline{E}) \\ & | & \texttt{new}\ C(\overline{E})\ ; \end{array}} \quad = \quad \boxed{\begin{array}{lll} E & : & x \\ & | & E.f \\ & | & E.m(\overline{E}) \\ & | & \texttt{new}\ C(\overline{E}) \\ & | & (C)\,E\,; \end{array}}$$

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



$$
\begin{aligned}
\texttt{E} \;:\;& \texttt{x} \\
\mid\;& \texttt{E.f} \\
\mid\;& \texttt{E.m}(\overline{\texttt{E}}) \\
\mid\;& \texttt{new C}(\overline{\texttt{E}})\;;
\end{aligned}
$$

cFJ expression syntax

$$
\begin{aligned}
\texttt{E} \;:\;& \texttt{x} \\
\mid\;& \texttt{E.f} \\
\mid\;& \texttt{TP}_{\texttt{m}}\ \texttt{E.m}\ (\overline{\texttt{E}}) \\
\mid\;& \texttt{new}\ (\ \texttt{TP}_{\texttt{t}}\ \texttt{C})\ (\overline{\texttt{E}}); \\
\texttt{TP}_{\texttt{m}} \;:\;& \epsilon; \\
\texttt{TP}_{\texttt{t}} \;:\;& \epsilon;
\end{aligned}
$$

$$\texttt{E} \;:\; (\texttt{C})\ \texttt{E}\;;$$

Cast syntax

$$\texttt{E} \;:\; (\ \texttt{TP}_{\texttt{t}}\ \texttt{C})\ \texttt{E};$$

Composition

- Syntax for original `FJ = Cast□cFJ`
- Syntax for `Generics`
- Syntax for `Generics□FJ`
- Exception (mentioned earlier) – replace default VP definition

$$
\boxed{\begin{array}{lcl}
\mathrm{TP}_{\mathrm{m}} & : & \langle \overline{\mathrm{T}} \rangle; \\
\mathrm{TP}_{\mathrm{t}} & : & \langle \overline{\mathrm{T}} \rangle;
\end{array}}
\quad □ \quad
\boxed{\begin{array}{lcl}
\mathrm{E} & : & \mathrm{x} \\
 & | & \mathrm{E.f} \\
 & | & \mathrm{TP}_{\mathrm{m}}\ \mathrm{E.m}\ (\overline{\mathrm{E}}) \\
 & | & \mathrm{new}\ (\mathrm{TP}_{\mathrm{t}}\ \mathrm{C})\ (\overline{\mathrm{E}}) \\
 & | & (\mathrm{TP}_{\mathrm{t}}\ \mathrm{C})\ \mathrm{E}; \\
\mathrm{TP}_{\mathrm{m}} & : & \epsilon; \\
\mathrm{TP}_{\mathrm{t}} & : & \epsilon;
\end{array}}
\quad = \quad
\boxed{\begin{array}{lcl}
\mathrm{E} & : & \mathrm{x} \\
 & | & \mathrm{E.f} \\
 & | & \mathrm{TP}_{\mathrm{m}}\ \mathrm{E.m}\ (\overline{\mathrm{E}}) \\
 & | & \mathrm{new}\ (\mathrm{TP}_{\mathrm{t}}\ \mathrm{C})\ (\overline{\mathrm{E}}) \\
 & | & (\mathrm{TP}_{\mathrm{t}}\ \mathrm{C})\ \mathrm{E}; \\
\mathrm{TP}_{\mathrm{m}} & : & \langle \overline{\mathrm{T}} \rangle; \\
\mathrm{TP}_{\mathrm{t}} & : & \langle \overline{\mathrm{T}} \rangle;
\end{array}}
$$

`Generics` `FJ` `Generics□FJ`

Inlining

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

$$
\begin{array}{rcl}
\mathtt{E} & : & \mathtt{x} \\
 & | & \mathtt{E.f} \\
 & | & \mathtt{TP_m}\ \mathtt{E.m}\ (\overline{\mathtt{E}}) \\
 & | & \mathtt{new}\ (\ \mathtt{TP_t}\ \mathtt{C})\ (\overline{\mathtt{E}}) \\
 & | & (\ \mathtt{TP_t}\ \mathtt{C})\ \mathtt{E}; \\
\mathtt{TP_m} & : & \langle\overline{\mathtt{T}}\rangle; \\
\mathtt{TP_t} & : & \langle\overline{\mathtt{T}}\rangle;
\end{array}
\quad \xrightarrow{\text{inline}} \quad
\begin{array}{rcl}
\mathtt{E} & : & \mathtt{x} \\
 & | & \mathtt{E.f} \\
 & | & \langle\overline{\mathtt{T}}\rangle\ \mathtt{E.m}\ (\overline{\mathtt{E}}) \\
 & | & \mathtt{new}\ (\langle\overline{\mathtt{T}}\rangle\ \mathtt{C})\ \ (\overline{\mathtt{E}}) \\
 & | & (\langle\overline{\mathtt{T}}\rangle\ \mathtt{C})\ \mathtt{E}\ ;
\end{array}
$$

- 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 `Cast`
- Composition `Cast□cFJ` is the union of these rules

$$\boxed{\dfrac{\Gamma \vdash \mathtt{e}_0 : \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}}\ \text{(T-UCAST)}}$$

`Cast`

□

$$\boxed{\begin{array}{c}\dfrac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{V}}\ \overline{\mathtt{f}} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \overline{\mathtt{U}} <: \overline{\mathtt{V}}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}}\ \text{(T-NEW)} \\ \vdots\end{array}}$$

`cFJ`

=

$$\boxed{\begin{array}{c}\dfrac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{V}}\ \overline{\mathtt{f}} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \overline{\mathtt{U}} <: \overline{\mathtt{V}}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}}\ \text{(T-NEW)} \\ \vdots \\ \dfrac{\Gamma \vdash \mathtt{e}_0 : \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}}\ \text{(T-UCAST)}\end{array}}$$

`Cast□cFJ`

Modifying Rules

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

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

$$\vdots$$

generalize

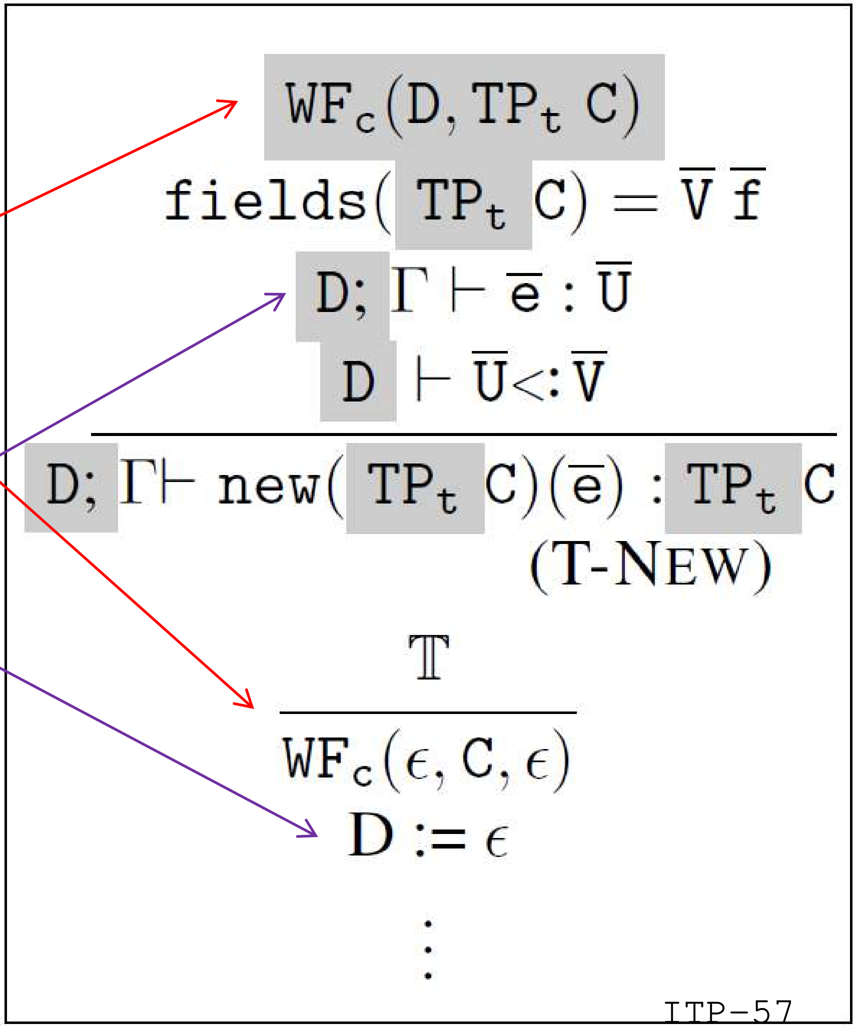
$$\frac{\begin{array}{c}\mathtt{WF}_{\mathtt{c}}(\mathrm{D}, \mathtt{TP}_{\mathtt{t}}\ \mathtt{C}) \\ \mathtt{fields}(\mathtt{TP}_{\mathtt{t}}\ \mathtt{C}) = \overline{\mathtt{V}}\ \overline{\mathtt{f}} \\ \mathrm{D}; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \\ \mathrm{D} \vdash \overline{\mathtt{U}} <: \overline{\mathtt{V}}\end{array}}{\mathrm{D}; \Gamma \vdash \mathtt{new}(\mathtt{TP}_{\mathtt{t}}\ \mathtt{C})(\overline{\mathtt{e}}) : \mathtt{TP}_{\mathtt{t}}\ \mathtt{C}} \quad \text{(T-NEW)}$$

$$\frac{\mathbb{T}}{\mathtt{WF}_{\mathtt{c}}(\epsilon, \mathrm{C}, \epsilon)} \quad \mathrm{D} := \epsilon$$

$$\vdots$$

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)



$$\frac{\begin{array}{c}\mathtt{WF_c(D, TP_t\ C)} \\ \mathtt{fields(\ TP_t\ C) = \overline{V}\ \overline{f}} \\ \mathtt{D;\ \Gamma \vdash \overline{e} : \overline{U}} \\ \mathtt{D\ \vdash \overline{U} <: \overline{V}}\end{array}}{\mathtt{D;\ \Gamma \vdash new(\ TP_t\ C)(\overline{e})\ :\ TP_t\ C}}\ \text{(T-NEW)}$$

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

$$\mathrm{D} := \epsilon$$

$$\vdots$$

ITP-57

Composition (as Before)

- Typing rules for `cFJ`
- Typing rules for `Generics` (replaces default declarations for $\mathtt{WF}_{\mathtt{C}}$ and `D`)
- Typing rules for `Generics`□`cFJ`

$$\frac{\Delta \vdash \langle \overline{\mathtt{T}} \rangle \mathtt{C}\ \mathtt{ok}}{\mathtt{WF}_{\mathtt{c}}(\Delta, \langle \overline{\mathtt{T}} \rangle\ \mathtt{C})}$$

$$\mathrm{D} := \Delta$$

□

$$\frac{\begin{array}{c} \mathtt{WF}_{\mathtt{c}}(\mathtt{D}, \mathtt{TP}_{\mathtt{t}}\ \mathtt{C}) \\ \mathtt{fields}(\ \mathtt{TP}_{\mathtt{t}}\ \mathtt{C}) = \overline{\mathtt{V}}\ \overline{\mathtt{f}} \\ \mathtt{D};\ \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \\ \mathtt{D}\ \vdash \overline{\mathtt{U}} <: \overline{\mathtt{V}} \end{array}}{\mathtt{D};\ \Gamma \vdash \mathtt{new}(\ \mathtt{TP}_{\mathtt{t}}\ \mathtt{C})(\overline{\mathtt{e}}) : \mathtt{TP}_{\mathtt{t}}\ \mathtt{C}} \quad \text{(T-NEW)}$$

$$\frac{\mathbb{T}}{\mathtt{WF}_{\mathtt{c}}(\epsilon, \mathtt{C}, \epsilon)}$$

$$\mathrm{D} := \epsilon$$

=

$$\frac{\begin{array}{c} \mathtt{WF}_{\mathtt{c}}(\mathtt{D}, \mathtt{TP}_{\mathtt{t}}\ \mathtt{C}) \\ \mathtt{fields}(\ \mathtt{TP}_{\mathtt{t}}\ \mathtt{C}) = \overline{\mathtt{V}}\ \overline{\mathtt{f}} \\ \mathtt{D};\ \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \\ \mathtt{D}\ \vdash \overline{\mathtt{U}} <: \overline{\mathtt{V}} \end{array}}{\mathtt{D};\ \Gamma \vdash \mathtt{new}(\ \mathtt{TP}_{\mathtt{t}}\ \mathtt{C})(\overline{\mathtt{e}}) : \mathtt{TP}_{\mathtt{t}}\ \mathtt{C}} \quad \text{(T-NEW)}$$

$$\frac{\Delta \vdash \langle \overline{\mathtt{T}} \rangle \mathtt{C}\ \mathtt{ok}}{\mathtt{WF}_{\mathtt{c}}(\Delta, \langle \overline{\mathtt{T}} \rangle\ \mathtt{C})}$$

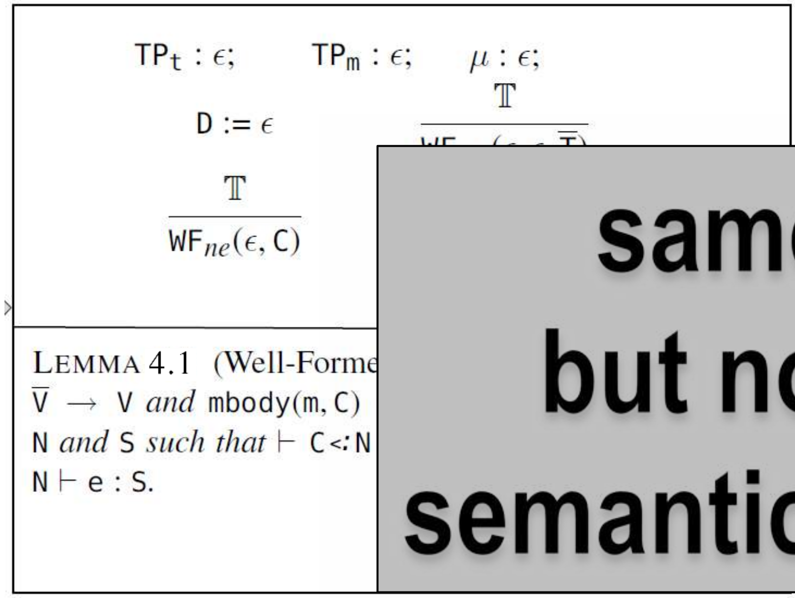
$$\mathrm{D} := \Delta$$

finally!

THEOREMS AND PROOFS

Theorems

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



$\mathsf{TP_t} : \epsilon;$ $\mathsf{TP_m} : \epsilon;$ $\mu : \epsilon;$

$\mathsf{D} := \epsilon$

$$\frac{\mathbb{T}}{\mathsf{WF}_{ne}(\epsilon, \mathsf{C})}$$

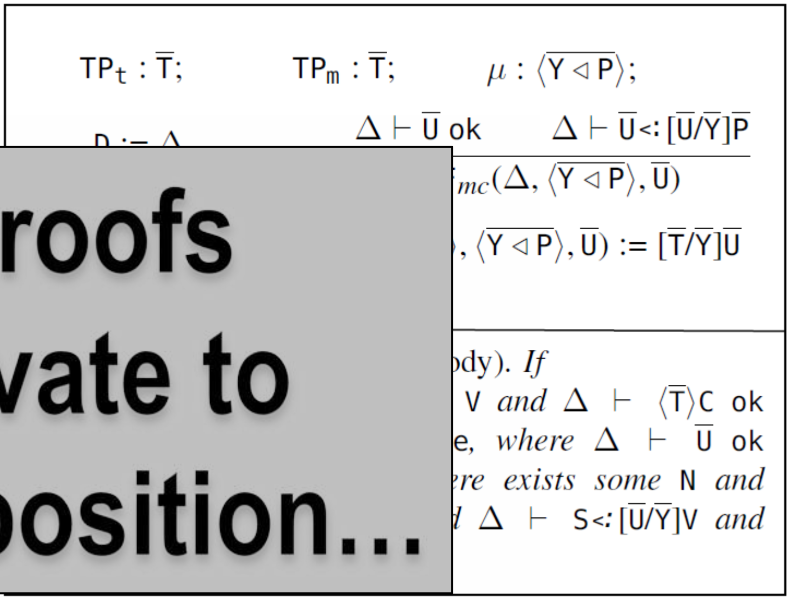
$\mathbb{T}$

LEMMA 4.1 (Well-Forme

$\overline{\mathsf{V}} \rightarrow \mathsf{V}$ *and* mbody(m, C)

N *and* S *such that* $\vdash$ C<:N

N $\vdash$ e : S.



$\mathsf{TP_t} : \overline{\mathsf{T}};$ $\mathsf{TP_m} : \overline{\mathsf{T}};$ $\mu : \langle \overline{\mathsf{Y} \triangleleft \mathsf{P}} \rangle;$

$\Delta \vdash \overline{\mathsf{U}}$ ok $\quad \Delta \vdash \overline{\mathsf{U}}$<: $[\overline{\mathsf{U}}/\overline{\mathsf{Y}}]\overline{\mathsf{P}}$

$mc(\Delta, \langle \overline{\mathsf{Y} \triangleleft \mathsf{P}} \rangle, \overline{\mathsf{U}})$

$, \langle \overline{\mathsf{Y} \triangleleft \mathsf{P}} \rangle, \overline{\mathsf{U}}) := [\overline{\mathsf{T}}/\overline{\mathsf{Y}}]\overline{\mathsf{U}}$

dy). *If*

V *and* $\Delta \vdash \langle \overline{\mathsf{T}} \rangle$C ok

e, *where* $\Delta \vdash \overline{\mathsf{U}}$ ok

re exists some N *and*

$\Delta \vdash$ S<: $[\overline{\mathsf{U}}/\overline{\mathsf{Y}}]$V *and*

same for proofs
but now elevate to
semantic composition…

cFJ

Generic□cFJ

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)

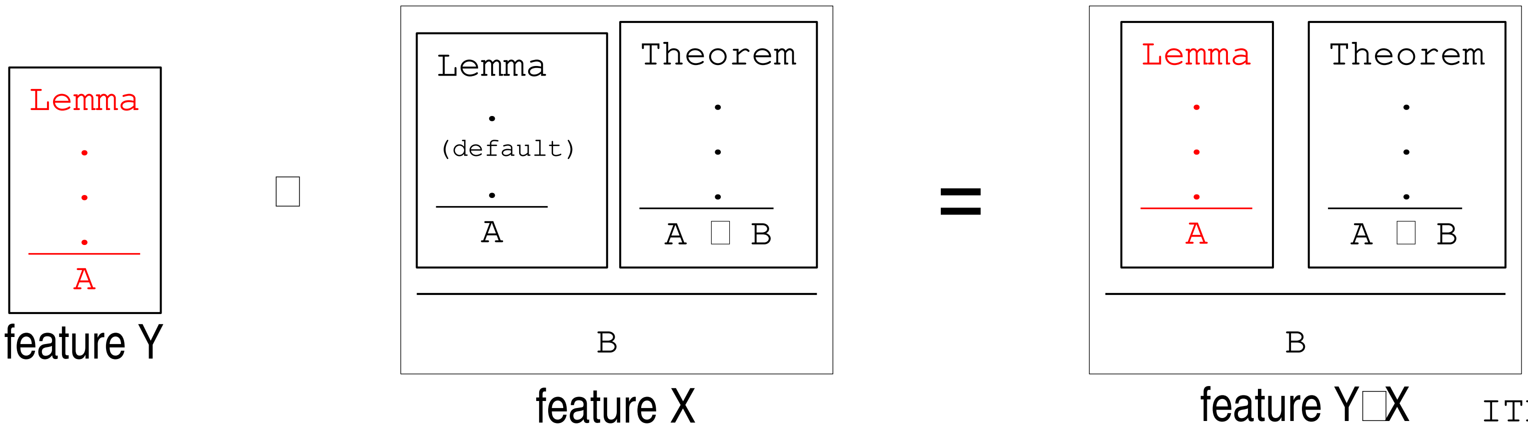
- Allows a general theorem to be proven, independent of features that might "plug-in" specific definitions for its VPs
    - *in effect, the proof assumes a general behavior for all possible VP instantiations*

- Obligation: any feature that "plugs-in" a VP definition <u>must supply a proof</u> 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

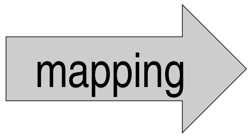
ENCODING FEATURE MODULES IN COQ

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:

$$\begin{array}{lcl} \mathrm{E} & : & \mathrm{x} \\ & | & \mathrm{E.f} \\ & | & \mathrm{TP_m}\ \mathrm{E.m}\,(\overline{\mathrm{E}}) \\ & | & \mathtt{new}\ (\mathrm{TP_t}\ \mathrm{C})\ (\overline{\mathrm{E}}) \\ & | & (\mathrm{TP_t}\ \mathrm{C})\ \mathrm{E}; \\ \mathrm{TP_m} & : & \epsilon; \\ \mathrm{TP_t} & : & \epsilon; \end{array}$$

Syntax Notation

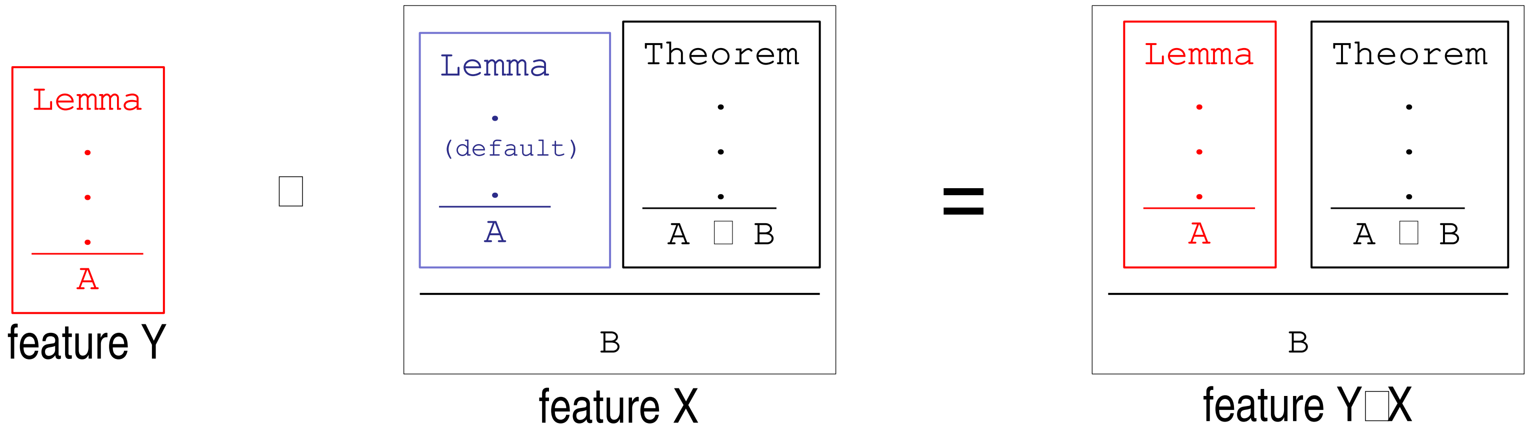


```
Definition TP_m := unit.
Definition TP_t := unit.

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_m_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

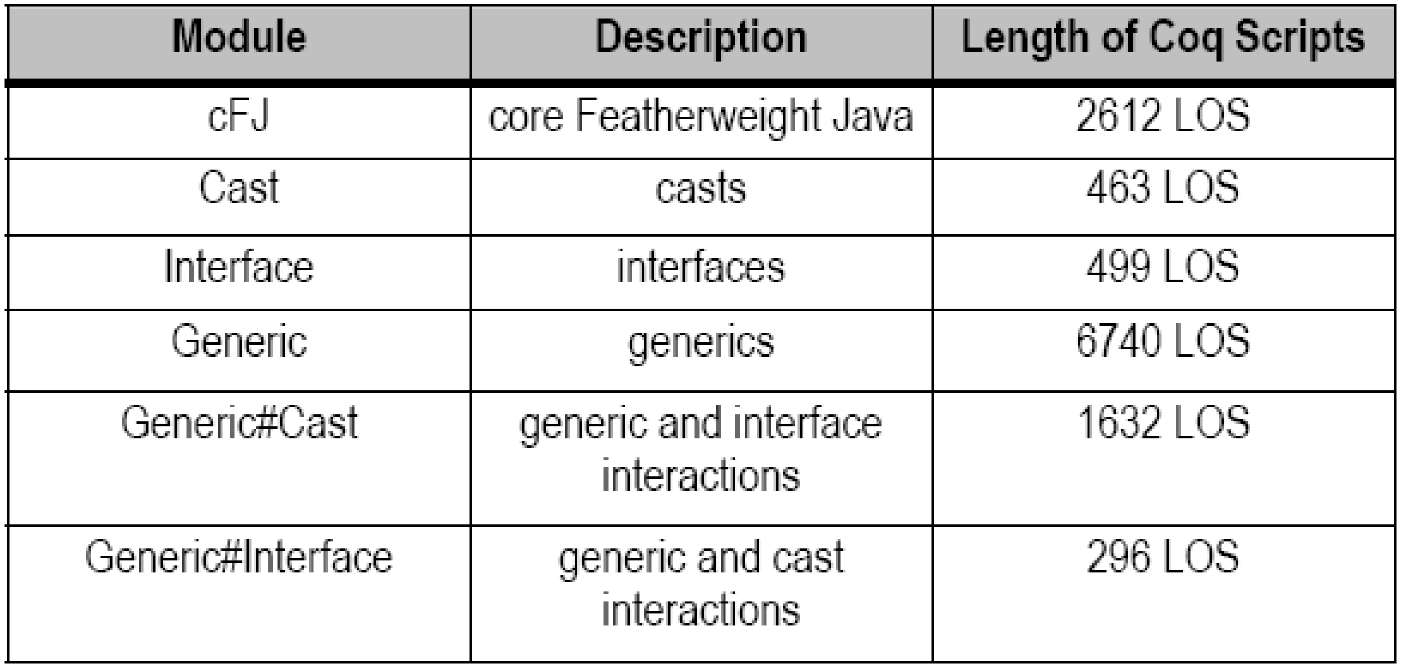
```
Variable TLookup_app : forall gamma delta X ty,
   TLookup gamma X ty →
   TLookup (app_context gamma delta) X ty.

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

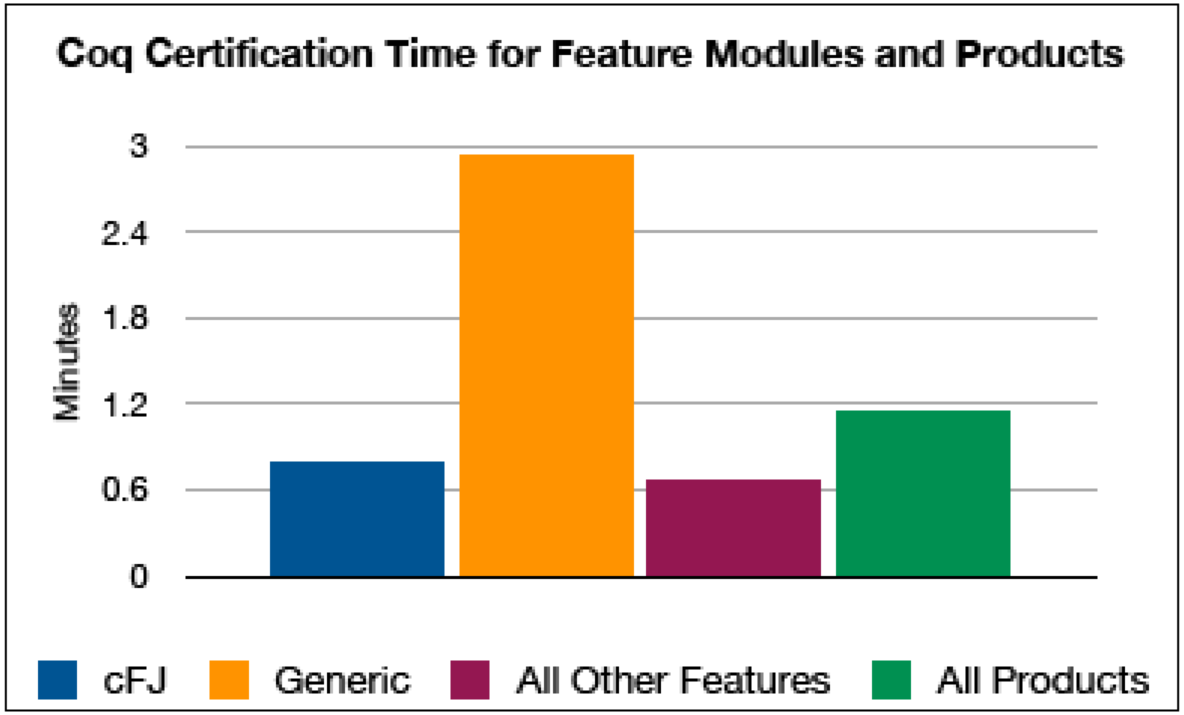
- One Coq file per module that encapsulates all representations



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

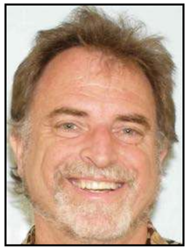
- A feature-module-level composition operator builds the complete set of definitions and proofs from a product specification automatically

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!