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

**Diagram:**

1. **SQL select statement**
2. **Parser**
3. **Inefficient relational algebra expression**
4. **Optimizer**
5. **Efficient relational algebra expression**
6. **Code generator**

**Keywords:**
- Declarative domain-specific language
- Automatic programming
- Generative programming
- Efficient program
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
• **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

```
P0  P1  P2
  f  h  k
P6  P7  P8
  g  i  j
```

Actually categories, but this is not relevant to this talk
Example: a 4-Program SPL

- Elementary product line of Java calculators
```java
class calculator {
    float result;
    void clear() { result=0; }
    void add( float x ) { result=+x; }
    void sub( float x ) { result=-x; }
}
class gui {
    JButton add = new JButton("+")
    JButton sub = new JButton("-");
    JButton format = new JButton("format");
    void initGui() {
        ContentPane.add( add );
        ContentPane.add( sub );
        ContentPane.add( format );
    }
    void initListeners() {
        add.addActionListener(...);
        sub.addActionListener(...);
    }
    void formatResultString() {...}
}
```
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

Java™ and the Java™ Virtual Machine
Definition, Verification, Validation

Springer
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
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
Statement of Correctness

Theorem 14.1.1 (Correctness of the Compiler). There exists a monotonic mapping \( \pi \) 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:

\[
(reg) \quad (bool1) \quad (exc) \\
(stack) \quad (bool2) \quad (exc-clinit) \\
(begS) \quad (new) \quad (clinit) \\
(begE) \quad (stm) \quad (fin) \\
(exp) \quad (abr) \\
\]

\[
\text{Java1.0 = StmE \cdot ExpE \cdot ExpO \cdot StmC \cdot ExpC \cdot StmI \cdot ExpI}
\]
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.

**Case 6.** $\text{context}(\text{pos}_n) = \alpha(\text{uop}\beta\text{exp})$ and $\text{pos}_n = \alpha$:
Similar to Case 3.

**Case 7.** $\text{context}(\text{pos}_n) = \alpha(\text{uop}\beta\text{val})$ and $\text{pos}_n = \beta$:
Similar to Case 5. If $\text{uop}$ is the negation operator and $\alpha$ is a $B_1(\text{lab})$-position, then according to the compilation scheme in Fig. 9.3, the position $\beta$ is $B_0(\text{lab})$-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.** $\text{context}(\text{pos}_n) = \alpha(\text{loc} = \beta\text{exp})$ and $\text{pos}_n = \alpha$:
Similar to Case 3.

**Case 9.** $\text{context}(\text{pos}_n) = \alpha(\text{loc} = \beta\text{val})$ and $\text{pos}_n = \beta$:
Assume that $\alpha$ is an $E$-position and that the size of the type of the variable $\text{loc}$ 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.
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

ITP-27

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…
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:

  $$D = \left\{ \begin{array}{l}
  B_1 \quad // \text{base program 1} \\
  B_2 \quad // \text{base program 2} \\
  F_1 \quad // \text{optional feature 1} \\
  F_2 \quad // \text{optional feature 2} \\
  \ldots \\
  F_n \quad // \text{optional feature n}
  \end{array} \right\}$$

- 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:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cFJ</td>
<td>core Featherweight Java</td>
</tr>
<tr>
<td>Cast</td>
<td>adds casts to expressions</td>
</tr>
<tr>
<td>Interface</td>
<td>adds interfaces</td>
</tr>
<tr>
<td>Generic</td>
<td>adds type parameters</td>
</tr>
</tbody>
</table>

- 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 · Cast · cFJ // FGJ with
// 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

\[
D : \; [k] \; [i] \; [j] \; b \; ; \quad \text{// context free grammar}
\]
\[
k \; \sqcup \; j \; \sqcup \; i ; \quad \text{// additional constraints}
\]
\[
k \; \sqcup \; j ;
\]

- **Assuming no feature interactions**, sentence of a feature model (‘kj b’) is mapped to an expression by a dot-product of its terms

\[
k \; \cdot \; j \; \cdot \; b
\]
Our Example

\[ L : \text{[Generic]} \text{[Interface]} \text{[Cast]} \text{cFJ}; \]

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cFJ</code></td>
<td>Core FJ</td>
</tr>
<tr>
<td><code>Cast\cdot cFJ</code></td>
<td>Original FJ [13]</td>
</tr>
<tr>
<td><code>Interface\cdot cFJ</code></td>
<td>Core FJ with Interfaces</td>
</tr>
<tr>
<td><code>Interface\cdot Cast\cdot cFJ</code></td>
<td>Original FJ with Interfaces</td>
</tr>
<tr>
<td><code>Generic\cdot cFJ</code></td>
<td>Core Featherweight Generic Java</td>
</tr>
<tr>
<td><code>Generic\cdot Cast\cdot cFJ</code></td>
<td>Original FGJ</td>
</tr>
<tr>
<td><code>Generic\cdot Interface\cdot cFJ</code></td>
<td>core Generic FJ with</td>
</tr>
<tr>
<td></td>
<td>[ Generic Interfaces ]</td>
</tr>
<tr>
<td><code>Generic\cdot Interface\cdot Cast\cdot cFJ</code></td>
<td>FGJ with</td>
</tr>
<tr>
<td></td>
<td>[ Generic Interfaces ]</td>
</tr>
</tbody>
</table>
#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 ($FP$):

$$FP = F \circ P$$

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

- grammar of FP
- source of FP
- manual of FP
Our Example

• Base language ($cFJ$) has multiple representations

<table>
<thead>
<tr>
<th>base representation</th>
<th>specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>syntax</td>
<td>$S_{cFJ}$</td>
</tr>
<tr>
<td>operational semantics</td>
<td>$O_{cFJ}$</td>
</tr>
<tr>
<td>type system</td>
<td>$T_{cFJ}$</td>
</tr>
<tr>
<td>meta-theory proofs</td>
<td>$P_{cFJ}$</td>
</tr>
</tbody>
</table>

• Base language is a 4-tuple:

$$cFJ = [ S_{cFJ}, O_{cFJ}, T_{cFJ}, P_{cFJ} ]$$
**Our Example**

- An optional feature \( j \) extends each representation:

<table>
<thead>
<tr>
<th>representation change</th>
<th>specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>syntax</td>
<td>( \Delta s_j )</td>
</tr>
<tr>
<td>operational semantics</td>
<td>( \Delta o_j )</td>
</tr>
<tr>
<td>type system</td>
<td>( \Delta t_j )</td>
</tr>
<tr>
<td>meta-theory proofs</td>
<td>( \Delta p_j )</td>
</tr>
</tbody>
</table>

- 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:

$$FJ = \text{Cast} \triangleq cFJ$$

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

= $\begin{bmatrix}
\Delta s_{\text{cast}} \oplus s_{\text{cFJ}} & \Delta o_{\text{cast}} \oplus o_{\text{cFJ}} & \Delta t_{\text{cast}} \oplus t_{\text{cFJ}} & \Delta p_{\text{cast}} \oplus p_{\text{cFJ}}
\end{bmatrix}$

- 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

ITP-42
Feature Interactions

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

- Cross-product ($\square$) 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

\[ p = k \times j \times b \]  
\[ = k \times (j \# b \# j \# b) \]  
\[ = k \# (j \# b \# j \# b) \times k \# (j \# b \# j \# b) \]  
\[ = k \# j \# b \# k \# j \# b \# k \# j \# b \# j \# b \]  

// # dist over \[ \]  

• 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

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cFJ</td>
<td>core Featherweight Java</td>
</tr>
<tr>
<td>Cast</td>
<td>cast</td>
</tr>
<tr>
<td>Interface</td>
<td>interfaces</td>
</tr>
<tr>
<td>Generic</td>
<td>generics</td>
</tr>
<tr>
<td>Generic#Interface</td>
<td>generic and interface interactions</td>
</tr>
<tr>
<td>Generic#Cast</td>
<td>generic and cast interactions</td>
</tr>
</tbody>
</table>

\[
\text{Generic} \times \text{Interface} \times \text{cFJ} = \text{Generic} \# \text{Interface} \# \text{cFJ} \\
\text{Generic} \# \text{cFJ} \# \text{Generic} \# \text{Interface} \# \text{cFJ} \# \text{Interface} \# \text{cFJ} \\
= 1 \# \text{Generic} \# \text{Interface} \# 1 \# \text{Generic} \# 1 \# \text{Interface} \# \text{cFJ} \\
= \text{Generic} \# \text{Interface} \# \text{Generic} \# \text{Interface} \# \text{cFJ} \]
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 \( Cast \)\( \downarrow \)\( cFJ \) is the union of rules
- Easy – only one exception to be considered shortly

\[
\begin{align*}
E : \ (C) \ E ; & \quad \square \\
\text{|} & \\
\text{|} & E : x \\
\text{|} & E.f \\
\text{|} & E.m(\overline{E}) \\
\text{|} & \text{new} \ C(\overline{E}) ; \\
\end{align*}
\]

\[
\text{=} \\
\begin{align*}
E : \ x & \\
\text{|} & E.f \\
\text{|} & E.m(\overline{E}) \\
\text{|} & \text{new} \ C(\overline{E}) \\
\text{|} & (C) \ E ; \\
\end{align*}
\]
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

\[
E : x
\]
\[
E.f
\]
\[
E.m(E)
\]
\[
\text{new } C(E)
\]

**cFJ expression syntax**

\[
E : (C) E;
\]

**Cast syntax**

\[
E : x
\]
\[
E.f
\]
\[
E.m(E)
\]
\[
\text{new } (TP_t C)(E)
\]

**generalize**

**VP definitions**

**variation points (VPs)**

- \[ TP_m : \epsilon; \]
- \[ TP_t : \epsilon; \]
Composition

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

\[
\begin{align*}
\text{TP}_m : & \left\langle \overline{T} \right\rangle; \\
\text{TP}_t : & \left\langle \overline{T} \right\rangle;
\end{align*}
\]

Generics

\[
\begin{align*}
E : & x \\
| & E.f \\
| & \text{TP}_m \text{E.m} (\overline{E}) \\
| & \text{new} (\text{TP}_t \text{C}) (\overline{E}) \\
| & (\text{TP}_t \text{C}) E;
\end{align*}
\]

Generics $\square FJ$

\[
\begin{align*}
\text{TP}_m : & \langle T \rangle; \\
\text{TP}_t : & \langle T \rangle;
\end{align*}
\]
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”.

\[
\begin{align*}
E &: x \\
& | \hspace{0.5cm} E.f \\
& | \hspace{0.5cm} TP_m \ E.m (E) \\
& | \hspace{0.5cm} \text{new} \ (TP_t \ C) \ (E) \\
& | \hspace{0.5cm} (TP_t \ C) \ E ; \\
TP_m &: \langle T \rangle ; \\
TP_t &: \langle T \rangle ;
\end{align*}
\]
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 \( \text{Cast} \sqcup cFJ \) is the union of these rules

\[
\begin{align*}
\Gamma \vdash e_0 : D & \quad D <: C \\
\Gamma \vdash e_0 : C & \\
\hline
\Gamma \vdash e_0 : C & \\
\text{(T-CAST)}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \overline{e} : \overline{U} & \quad \overline{U} <: \overline{V} \\
\Gamma \vdash \text{new } C(\overline{e}) : C & \\
\hline
\Gamma \vdash \text{new } C(\overline{e}) : C & \\
\text{(T-NEW)}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_0 : D & \quad D <: C \\
\hline
\Gamma \vdash e_0 : C & \\
\text{(T-CAST)}
\end{align*}
\]

\[
\begin{align*}
\text{fields}(C) = \overline{V} \overline{f} \\
\Gamma \vdash \overline{e} : \overline{U} & \quad \overline{U} <: \overline{V} \\
\Gamma \vdash \text{new } C(\overline{e}) : C & \\
\hline
\Gamma \vdash \text{new } C(\overline{e}) : C & \\
\text{(T-NEW)}
\end{align*}
\]
Modifying Rules

- Requires VPs to be defined
- Typing rules for \( cF\) expressions
- Generalize by adding VPs
- VPs have more sophisticated meaning

\[
\begin{align*}
\text{fields}(C) &= \overline{V} \overline{f} \\
\Gamma &\vdash \overline{e} : \overline{U} \quad \overline{U} <: \overline{V} \\
\Gamma &\vdash \text{new } C(\overline{e}) : C \\
&\text{(T-NEW)}
\end{align*}
\]

\[
\begin{align*}
\text{WF}_c(D, \text{TP}_t C) \\
\text{fields}(\text{TP}_t C) &= \overline{V} \overline{f} \\
D; \Gamma &\vdash \overline{e} : \overline{U} \\
D &\vdash \overline{U} <: \overline{V} \\
D; \Gamma &\vdash \text{new } (\text{TP}_t C)(\overline{e}) : \text{TP}_t C \\
&\text{(T-NEW)}
\end{align*}
\]

\[
\begin{align*}
T \\
\text{WF}_c(\epsilon, C, \epsilon) \\
D := \epsilon \\
\vdots
\end{align*}
\]
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 $cFJ$

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

\[
\frac{}{D := \Delta}
\]

\[
\frac{WF_c(D, TP_t C) \quad \text{fields}(TP_t C) = \bar{V} \bar{f} \quad D; \Gamma \vdash e : \bar{U} \quad D \vdash \bar{U} < : \bar{V}}{WF_c(D, \langle T \rangle C) \text{ ok}}
\]

\[
\frac{D; \Gamma \vdash \text{new}(TP_t C)(e) : TP_t C \quad (T-NEW)}{WF_c(D, \langle T \rangle C) \text{ ok} \quad \text{fields}(TP_t C) = \bar{V} \bar{f}}
\]

\[
\frac{D; \Gamma \vdash e : \bar{U} \quad D \vdash \bar{U} < : \bar{V}}{WF_c(\Delta, \langle T \rangle C) \text{ ok}}
\]

\[
\frac{T}{WF_c(\epsilon, C, \epsilon)}
\]

\[
\frac{}{D := \epsilon}
\]

\[
\frac{D := \Delta \quad \text{ITP-58}}{WF_c(D, \langle T \rangle C)}
\]
finally!

THEOREMS AND PROOFS
Theorems

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

---

**Lemma 4.1** (Well-Formed Body) If \( \bar{V} \rightarrow \bar{V} \) and \( \text{mbody}(m, c) \) are such that \( \vdash C \leftarrow N \) and \( N \vdash e : S \).
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
ENCODING FEATURE MODULES IN COQ

VP for Ben
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:

Syntax Notation

```
E : x
  | E.f
  | TP_m E.m (E)
  | new (TP_t C) (E)
  | (TP_t C) E;
TP_m : ε;
TP_t : ε;
```

Coq Encoding

```
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.
```
**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

![Diagram of semantic composition](image)

- 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

```coq
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
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

```plaintext
Variable TLookup_app : forall gamma delta X ty, 
  TLookup gamma X ty \rightarrow 
  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.
```

ITP-67
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

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>Length of Coq Scripts</th>
</tr>
</thead>
<tbody>
<tr>
<td>cFJ</td>
<td>core Featherweight Java</td>
<td>2612 LOS</td>
</tr>
<tr>
<td>Cast</td>
<td>casts</td>
<td>463 LOS</td>
</tr>
<tr>
<td>Interface</td>
<td>interfaces</td>
<td>499 LOS</td>
</tr>
<tr>
<td>Generic</td>
<td>generics</td>
<td>6740 LOS</td>
</tr>
<tr>
<td>Generic#Cast</td>
<td>generic and interface interactions</td>
<td>1632 LOS</td>
</tr>
<tr>
<td>Generic#Interface</td>
<td>generic and cast interactions</td>
<td>296 LOS</td>
</tr>
</tbody>
</table>
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!