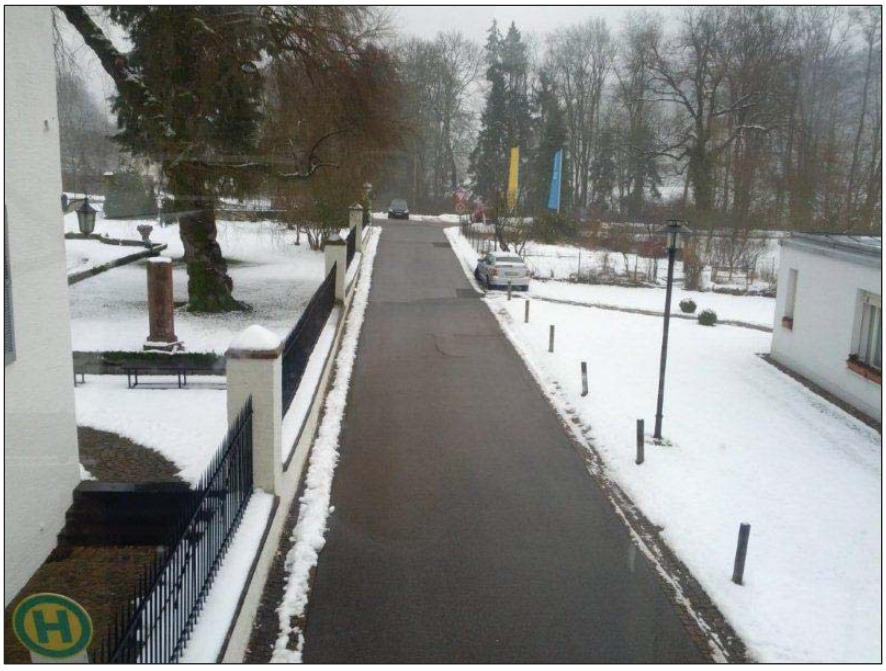
The Challenges and Science of Variability

Don Batory
Department of Computer Science
University of Texas at Austin
Austin, Texas 78712

Variability is Everywhere!

- Except the weather at Dagstuhl…



This Talk is a Perspective

- On the looming storms that face our community

in this talk, I'll explain why
our community will
play a vital role in its answer

Ivar Jacobson

workshop at this year at ICSE on the title

WHERE IS THE THEORY FOR SOFTWARE ENGINEERING?

Definition of Science

- From `dictionary.com`

sci·ence [sahy-*uh* ns] Show IPA

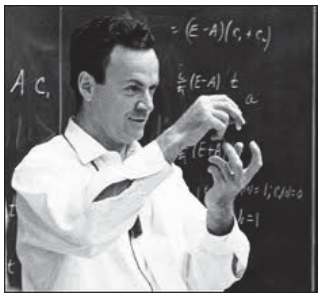
noun

1. a branch of knowledge or study dealing with a body of facts or truths systematically arranged and showing the operation of general laws: *the mathematical sciences.*
2. systematic knowledge of the physical or material world gained through observation and experimentation.
3. any of the branches of natural or physical science.
4. systematized knowledge in general.

- Dominant paradigm in SE insists on a rigorous hypothesis evaluation. A set of tests are conducted by an author and a careful analysis of one or more hypotheses must be presented. This is the "Scientific Method"
- This matches Definition 2 and the intended use of empirical methods in SE
- We are missing the most important part of science

And the Important Part?

- My answer is an analogy from physics…
- In physics, there are lots of poorly related phenomena – they vary some how
- A theoretical physicist would select a set and seek a mathematical theory that unifies them as manifestations of the same underlying concepts
  - broader the initial set
  - fewer the concepts
  - more general and significant the theory might be

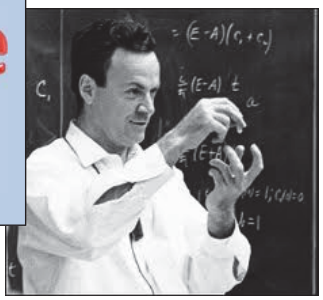


- Initial test of a theory is a check that it does precisely what it claims
  - reproduce, explain phenomena of the initial set
  - explain, predict other phenomena as well

And the Important Part?

- My answer is an analogy from physics…
- In physics, there are lots of poorly related phenomena – they vary some how
- A theoretical phy
  mathematical the
  of the same unde
  - broader
  - fewer th
  - more general and significant the theory might be

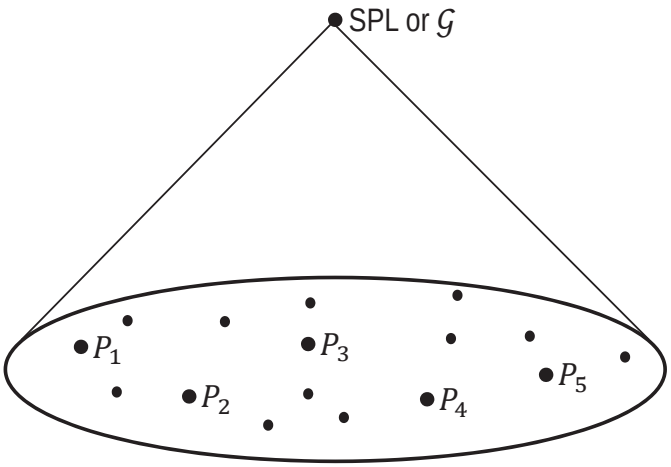
This is Science Definition #1



- Initial test of a theory is a check that it does precisely what it claims
  - reproduce, explain phenomena of the initial set
  - explain, predict other phenomena as well

Phenomena of Software Engineering

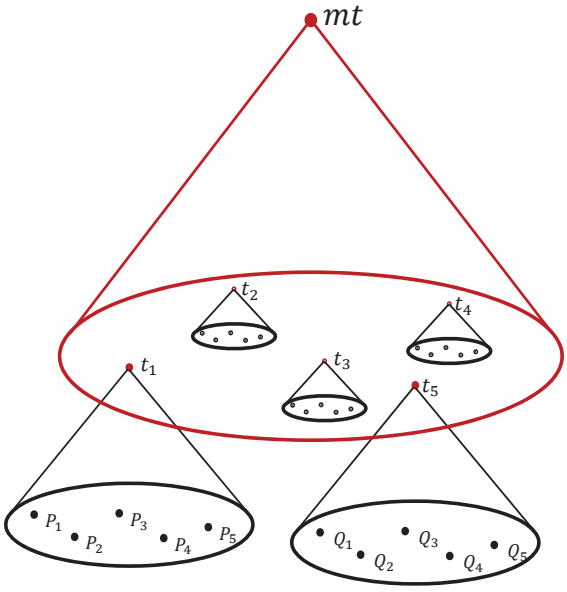
- Are programs with certain properties
- A software product line (SPL) or generator ($\mathcal{G}$), is a concrete embodiment of an "implicit" SE theory of how to automatically build programs in this domain with lower cost and higher quality



SPL or $\mathcal{G}$ not only explains and reproduces initial programs, but predicts and explains the existence of other programs as well

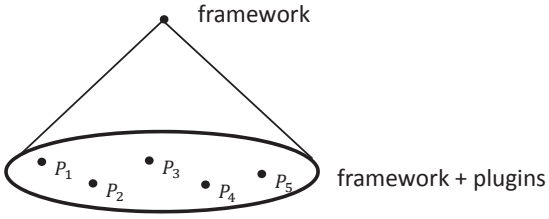
History and Experience Tells Us

- Such SE "theories" must be domain-specific (DS) to have any chance of success
- DS knowledge is rich and deep, with few specifics transferable to other domains
  - irony: DS theories ($t_1 \ldots t_n$) are not very interesting to the general SE community
- Meta-theories (mt) are more valued
  - domain-independent concepts
  - instances are DS theories
  - teach ideas to students; they will produce instances of their own



Familiar SE Meta-Theories

- Just not very "automatic" or mathematical
- OO frameworks are common in today's libraries
  - framework designers understand that a set of similar programs will be built
  - their OO framework codes the common objects and activities in this domain to minimize what others have to write
  - concepts of frameworks, abstract classes, plugins are meta-theory
  - we teach (meta-theory) to our students
  - our students instantiate concepts to create frameworks, plugins of their own



Another Example

- UML asserts than an OO design can be documented in the languages of class diagrams, state machines, etc. (the meta-theory part)
- We teach UML (meta-theory) to our students; they instantiate to design their own OO programs

sci·ence [sahy-*uh* ns] Show IPA

noun

1. a branch of knowledge or study dealing with a body of facts or truths systematically arranged and showing the operation of general laws: *the mathematical sciences.*
2. systematic knowledge of the physical or material world gained through observation and experimentation.
3. any of the branches of natural or physical science.
4. systematized knowledge in general.

- Not Definition 1, maybe Definition 4

What Brings Us Together Today

- The study of variability and its manifestations in SE
  - understand how program families can be built and analyzed automatically
  - our engineering efforts (SPLs, $\mathcal{G}$) are concrete demonstrations that our "theories" work

sci·ence [sahy-*uh* ns] Show IPA

noun

1. a branch of knowledge or study dealing with a body of facts or truths systematically arranged and showing the operation of general laws: *the mathematical sciences.*
2. systematic knowledge of the physical or material world gained through observation and experimentation.
3. any of the branches of natural or physical science.
4. systematized knowledge in general.

- For most, not Definition 1, maybe Definition 4

We need a MT!

- How tools should work – gives a precise definition of what “composition” means
    - are you aware of the volume of work where “composition” makes no sense mathematically?

- In mature communities, MT provides a standard way to describe problems and how to formulate solutions
    - type systems for programming languages
    - relational algebra and sets for classical databases
    - conceptual & technical glue that holds communities together

- MTs bring organization to what would otherwise be intellectual chaos

Your Work is Important!

Understanding Variability is the key to Scientific
Theories of Automated Software Design and Analysis

Your Thoughts on MT

Martin Luecker: “(we need) a well-understood theory with tool support”

Janet Siegmund: “(we need) proper tool support for developers working with variability”

Tiziana Margaria: “(we have) a lot of formalisms with unclear relation to each other”

Alessandro Fantechi: “there is no standard description language (to express) variability, only ad-hoc solutions are available”

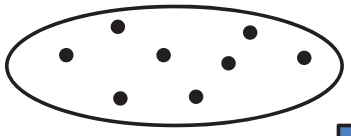
Andrzey Wasowski: “In my experience, SPLs are so complex and idiosyncratic, that providing generic tools for them is almost impossible (the build systems are for example all very different and project specific)”

First Step on the Road to MT Maturity

- Separate Concerns: distinguish abstractions from their implementations

- Agree on the core abstractions – implementations will sort themselves out in time

Towards Maturity

- For SPLs: Agree on basic MT abstractions
  - domain $\mathcal{D} =$ 
  - semantic modules $\vec{\mathcal{D}} = \{ F_1, F_2, \ldots F_n \}$
  - how modules compose $P \in \mathcal{D}: \; P = F_7 + F_4 + F_3$

assume no feature replication, static configs

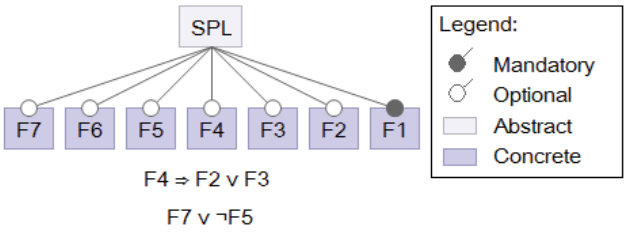
- Notation is by Pamela Zave circa 1999 akpublic.research.att.com/~pamela/faq.html

What is a feature?

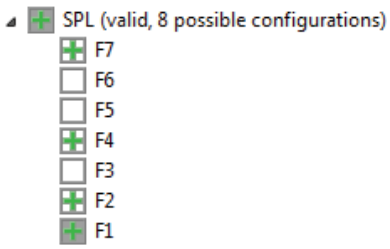
In a software system, a feature is an increment of functionality, usually with a coherent purpose. If a system description is organized by features, then it probably takes the form B + F1 + F2 + F3 . . ., where B is a base description, each Fi is a feature module, and + denotes some feature-composition operation.

We Know from Experience

- From feature modeling:



- And feature selection:



- $F_7 + F_4 + F_2 + F_1 = F_1 + F_2 + F_4 + F_7$ commutative
- $((F_7 + F_4) + F_2) + F_1 = (F_7 + (F_4 + (F_2 + F_1)))$ associative

But why?

Honestly…

- I was one of the last people to come to this conclusion about +
- Took years for me to understand feature interactions and other "stuff" to believe it
- So what I've just said is not obvious…

Your Thoughts on Implementations

Andreas Classen: "(we need) FOSD language support in mainstream programming languages (C#) and their IDEs (Visual Studio)"

Shriram Krishnamurthi: "True programming language modularity for supporting true variability modeling; we get one or the other, but not both".

Martin Erwig: "(we need) understand the fundamental difference between 'projectional' variation (SYSGEN) and 'alternative-based' variation (modular) "

Initial Work on ~~Semantic~~ Feature Modules

- Showed how scaling of ideas on mixins and was useful to distinguish code modules from semantic modules. It is a Language Problem!

Classes and Mixins

Matthew Flatt Shriram Krishnamurthi Matthias Felleisen

Department of Computer Science*
Rice University
Houston, Texas 77005-1892

POPL98

Implementing Layered Designs with Mixin Layers[1]

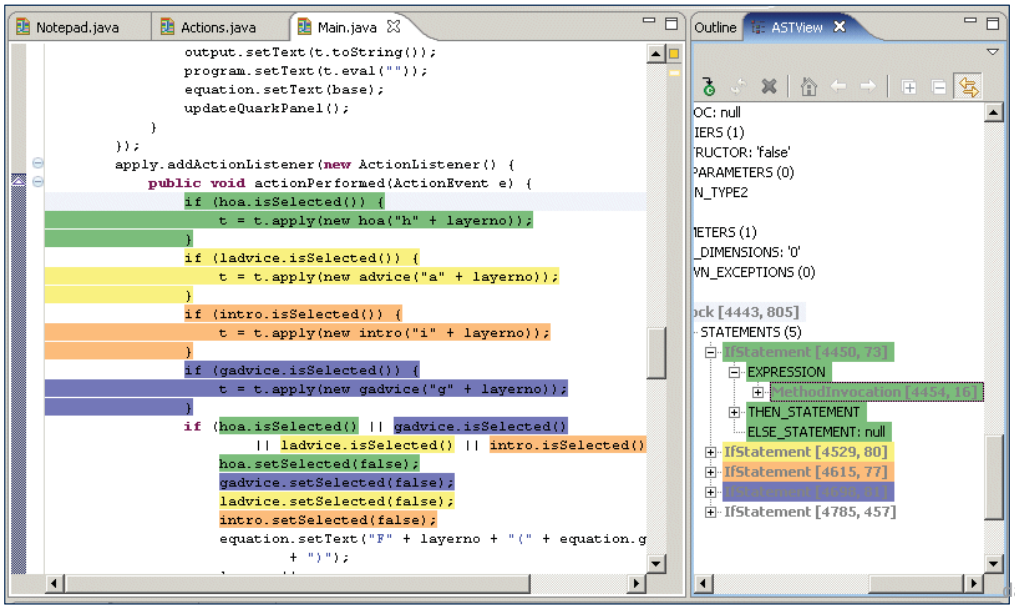
Yannis Smaragdakis
Department of Com
The University of T
Austin, Texas
{smaragd, dsb}@c

ECOOP98

- Can be traced to earlier work (Beta language, van Hilst)
- These ideas have since evolved…

More Recently: Coloring

- Equates semantic modularity with virtual modularity
- Allows micro changes to programs. It is a Tool Problem!



My Take…

- Very different implementations of the same ideas (abstractions)
  - classical modularity vs. virtual modularity
  - still thinking abstractly in terms of features and +
- We should expect that there will be different implementations of abstractions
  - multiple ways to implement feature (modules) and the + operation
  - different implementations, languages are used for different problems, purposes, analyses
  - one implementation does not fit all…
- Challenge: classic modules or virtual modules – YOUR work will decide when to use which and we all win
  - remember: manifestations of the same ideas

Your Thoughts on Variability

Kim Lauenroth: "(how to) specify variability across requirement artifacts"

Stefan Sobernig: "look beyond source code artifacts (e.g., documentation, building/deployment); how are variation points manifested in a product line?"

Stefania Gnesi: "how to combine feature models with behavior models?"

Sven Apel: "(how to reason with feature modules)"

Norbert Siegmund: "(how to) ensure software-quality (non-functional) properties of customizable programs"

What Meta-Theory Says

- We *know* programs have many concrete representations: source code (σ), performance estimate (π), requirements (ρ), makefile (μ), …
  - want to construct each representation by composing modules
- Suppose: $P = F_4 + F_3 + F_1$
- The source code of P, namely $\sigma(P)$, is constructed by code-composing ($+_\sigma$) the code modules of each of P's features

$$\begin{aligned} \sigma(P) &= \sigma(F_4 + F_3 + F_1) \\ &= \sigma(F_4) +_\sigma \sigma(F_3) +_\sigma \sigma(F_1) \end{aligned}$$

- Last step is a key behind modular construction AND modular reasoning

Meta-Theory

- Translating an expression in one algebra to an expression in another is called a

Homomorphism

- Fundamental concept in mathematics, just like addition (+)

Did you know that homomorphisms play a fundamental role in well-known and recent results on SPLs?

Scalable Prediction of Non-functional Properties in Software Product Lines

SPLC'11

- Problem: we have a feature expression of a program $P = A + B$
- For any $P \in \mathcal{D}$, want an estimate its efficiency w.r.t. a fixed workload

- Invented procedures to estimate the "change" in performance that each feature contributes to a program. Assuming that performance estimates are arithmetically added, work relied on the homomorphism:

$$\pi(A + B) = \pi(A) + \pi(B)$$

"efficiency estimate of a program is the arithmetic-sum of the estimates for each of its features"

- Surprisingly accurate predictions were reported with this simple approach

FeatureHouse: Language-Independent, Automated Software Composition

ICSE 09

Sven Apel
Department of Informatics and Mathematics
University of Passau
apel@uni-passau.de

Christian Kästner
School of Computer Science
University of Magdeburg
ckaestne@ovgu.de

Christian Lengauer
Department of Informatics and Mathematics
University of Passau
lengauer@uni-passau.de

- Shows how a representation of a program can be encoded as syntax trees and feature-composition is syntax-tree composition

- Give me the grammar of language $\mathcal{L}$ and specialized rules for composing $\mathcal{L}$-syntax trees, and FH produces a tool that implements the following homomorphism:

$$\mathcal{L}(A + B) = \mathcal{L}(A) +_{\mathcal{L}} \mathcal{L}(B)$$

"$\mathcal{L}$-source of a program is the $\mathcal{L}$-sum of the $\mathcal{L}$-source of each of its features"

- FH is a generator of language-specific tools that implement homomorphisms

Your Thoughts on Verification

Alessandro Fantechi: “(how to) factorize formal verification activities among products of a family”

Jean-Vivien Millo: “(we want) design verification in the presence of variability”

Holger Schlingloff: “(we want) reuse of verification artifacts within an SPL”

Ina Schaefer: “(we want) compositional verification of features and products”

Dave Clark: “need an adequate formal model of ‘feature’ that is amenable to formal verification in a modular way”

Proofs for SPL Programs

Thomas Thum: “how can we efficiently verify SPLs using theorem proving”

Proof Composition for Deductive Verification of Software Product Lines

Thomas Thüm*, Ina Schaefer†, Martin Kuhlemann*, and Sven Apel‡
*University of Magdeburg, Germany
†University of Braunschweig, Germany
‡University of Passau, Germany

ICSTW11

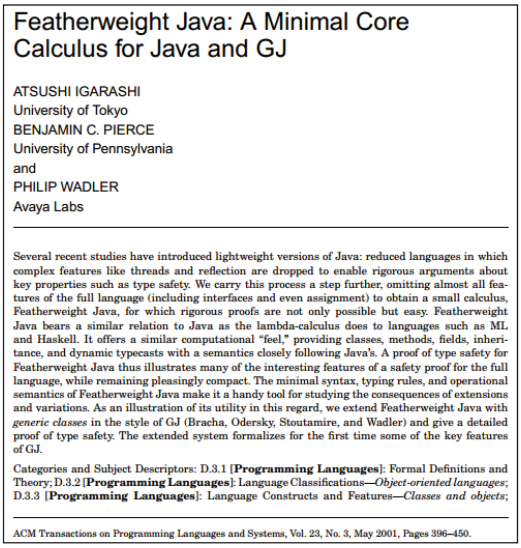
Product Lines of Theorems

Delaware William R. Cook

Department of Computer Science
University of Texas at Austin
{bendy,wcook,batory}@cs.utexas.e

OOPSLA 11

- Programming language literature is replete with examples of (tiny) product lines that include proofs
- Typically have only 2 members:
  - core Featherweight Java (FJ)
  - and a feature-extended version of FJ
- Original FJ paper also presented Featherweight Generic Java (FGJ) a modified version of FJ with support for generics



Featherweight Java: A Minimal Core Calculus for Java and GJ

ATSUSHI IGARASHI
University of Tokyo
BENJAMIN C. PIERCE
University of Pennsylvania
and
PHILIP WADLER
Avaya Labs

Several recent studies have introduced lightweight versions of Java: reduced languages in which complex features like threads and reflection are dropped to enable rigorous arguments about key properties such as type safety. We carry this process a step further, omitting almost all features of the full language (including interfaces and even assignment) to obtain a small calculus, Featherweight Java, for which rigorous proofs are not only possible but easy. Featherweight Java bears a similar relation to Java as the lambda-calculus does to languages such as ML and Haskell. It offers a similar computational "feel," providing classes, methods, fields, inheritance, and dynamic typecasts with a semantics closely following Java's. A proof of type safety for Featherweight Java thus illustrates many of the interesting features of a safety proof for the full language, while remaining pleasingly compact. The minimal syntax, typing rules, and operational semantics of Featherweight Java make it a handy tool for studying the consequences of extensions and variations. As an illustration of its utility in this regard, we extend Featherweight Java with *generic classes* in the style of GJ (Bracha, Odersky, Stoutamire, and Wadler) and give a detailed proof of type safety. The extended system formalizes for the first time some of the key features of GJ.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*;

ACM Transactions on Programming Languages and Systems, Vol. 23, No. 3, May 2001, Pages 396–450.

Type Soundness

- Integral part of any type system are the meta-theoretic proofs showing type soundness – the guarantee that the type system statically enforces the desired run-time behavior of a language, typically preservation and progress
  - preservation – if expression e has type T and e evaluates to a value v then v also has type T
  - progress – there are no expressions in the language that can't be evaluated
- To write these proofs, you need at 4 different representations of a language
  - syntax, typing rules for preservation, operational semantics rules for progress, meta-theory proofs

Formalization Includes

- Specification of language syntax
- Typing rules for preservation
- Rules for operational semantics (not shown) for progress
- All in their own notations

FJ Expression Syntax

$$
\begin{array}{rl}
e ::= & x \\
\mid & e.f \\
\mid & e.m\ (\overline{e}) \\
\mid & \texttt{new}\ C(\overline{e}) \\
\mid & (C)\ e
\end{array}
$$

FJ Subtyping $T <: T$

$$\frac{S <: T \qquad T <: V}{S <: V}\ (\text{S-TRANS})$$

$$\frac{}{T <: T}\ (\text{S-REFL})$$

$$\frac{\texttt{class C extends D \{...\}}}{C <: D}\ (\text{S-DIR})$$

FJ New Typing $\Gamma \vdash e : T$

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

Same for Proofs

- Proofs of preservation and progress
- Here's a fragment of the proof for field inheritance which proceeds by induction on the derivation of the subtyping judgment
- Has its own notation

FJ Fields of a Supertype Lemma

Lemma 2.1. *If* $S <: T$ *and* $\texttt{fields}(T) = \overline{T}\ f$, *then* $\texttt{fields}(S) = \overline{S}\ \overline{g}$ *and* $S_i = T_i$ *and* $g_i = f_i$ *for all* $i \leq \#(f)$.

Proof. By induction on the derivation of $S <: T$

Case S-REFL $S = T$.
Follows immediately.

Case S-TRANS $S <: V$ and $V <: T$.
By the inductive hypothesis, $\texttt{fields}(V) = \overline{V}\ \overline{h}$ and $V_i = T_i$ and $h_i = f_i$ for all $i \leq \#(f)$. Again applying the inductive hypothesis, $\texttt{fields}(S) = \overline{S}\ \overline{g}$ and $S_i = V_i$ and $g_i = h_i$ for all $i \leq \#(h)$. Since $\#(f) \leq \#(h)$, the conclusion is immediate.

Case S-DIR $S = C$, $T = D$,

$\texttt{class C extends D}\ \{\overline{S}\ \overline{g}; \ldots\}$.

By the rule F-CLASS, $\texttt{fields}(C) = \overline{U}\ \overline{f}; \overline{S}\ \overline{g}$, where $\overline{U}\ \overline{f} = \texttt{fields}(D)$, from which the conclusion is immediate.

Variability adds Complexity

FJ Expression Syntax		FGJ Expression Syntax
e ::= x | e.f | e.m (ē) | new C(ē) | (C) e	⟾	e ::= x | e.f | e.m ⟨T̄⟩$^{\beta}$ (ē) | new C ⟨T̄⟩$^{\beta}$ (ē) | (C ⟨T̄⟩$^{\beta}$) e

add new syntax, modify syntax

FJ Subtyping	T <: T	FGJ Subtyping	Δ$^{\delta}$ ⊢ T <: T

yes, this is coloring

$$\frac{S<:T \quad T<:V}{S<:V}\ (\text{S-TRANS}) \qquad T<:T\ (\text{S-REFL}) \qquad \frac{\texttt{class C extends D \{...\}}}{C<:D}\ (\text{S-DIR})$$

$$\Delta \vdash X<:\Delta(X)\ (\text{GS-VAR})^{\alpha} \qquad \frac{\Delta^{\delta}\vdash S<:T \quad \Delta^{\delta}\vdash T<:V}{\Delta^{\delta}\vdash S<:V}\ (\text{GS-TRANS}) \qquad \Delta^{\delta}\vdash T<:T\ (\text{GS-REFL})$$

$$\frac{\texttt{class C}\ \langle X \lhd N\rangle^{\beta}\ \texttt{extends D}\ \langle V\rangle^{\beta}\ \{...\}}{\Delta^{\delta}\vdash C\langle T\rangle^{\beta} <: [T/X]^{\eta} D\langle V\rangle^{\beta}}\ (\text{GS-DIR})$$

add new rules, modify rules

FJ New Typing	Γ ⊢	FGJ New Typing	Δ; $^{\delta}$Γ ⊢ e : T

$$\frac{\texttt{fields(C)} = \overline{V}\ \overline{f} \qquad \Gamma\vdash \overline{e} : \overline{U} \qquad \overline{U}<:\overline{V}}{\Gamma\vdash \texttt{new C}(\overline{e}) : C}\ (\text{T-NEW}) \qquad \frac{\Delta\vdash C\langle\overline{T}\rangle^{\gamma} \qquad \texttt{fields}(C\langle\overline{T}\rangle^{\beta}) = \overline{V}\ \overline{f} \qquad \Delta;^{\delta}\Gamma\vdash\overline{e}:\overline{U} \qquad \Delta^{\delta}\vdash\overline{U}<:\overline{V}}{\Delta;^{\delta}\Gamma\vdash \texttt{new C}\langle\overline{T}\rangle^{\beta}(\overline{e}) : C}\ (\text{GT-NEW})$$

extend judgment signatures, modify premises and conclusions

Proofs become more Complex too

FJ Fields of a Supertype Lemma		FGJ Fields of a Supertype Lemma
Lemma 2.1. *If* S<:T *and* fields(T) = T f, *then* fields(S) = S̄ ḡ *and* $S_i = T_i$ *and* $g_i = f_i$ *for all* $i \le \#(f)$.	⟾	**Lemma 2.2.** *If* Δ$^{\delta}$ ⊢ S<:T *and* fields(bound$_{\Delta}$(T)$^{\eta}$) = T̄ f̄, *then* fields(bound$_{\Delta}$(S)$^{\eta}$) = S̄ ḡ, $S_i = T_i$ *and* $g_i = f_i$ *for all* $i \le \#(f)$.
Proof. By induction on the derivation of S<:T		*Proof.* By induction on the derivation of Δ$^{\delta}$ ⊢ S<:T
		Case GS-VAR$^{\alpha}$ S = X and T = Δ(X). Follows immediately from the fact that bound$_{\Delta}$(Δ(X)) = Δ(X) by the definition of bound.
Case S-REFL S = T. Follows immediately.	⟾	**Case GS-REFL** S = T. Follows immediately.
Case S-TRANS S<:V and V<:T. By the inductive hypothesis, fields(V) = V̄ h̄ and $V_i = T_i$ and $h_i = f_i$ for all $i \le \#(f)$. Again applying the inductive hypothesis, fields(S) = S̄ ḡ and $S_i = V_i$ and $g_i = h_i$ for all $i \le \#(h)$. Since $\#(f) \le \#(h)$, the conclusion is immediate.	⟾	**Case GS-TRANS** Δ$^{\delta}$ ⊢ S<:V and Δ$^{\delta}$ ⊢ V<:T. By the inductive hypothesis, fields(bound$_{\Delta}$(V)$^{\eta}$) = V̄ h̄ and $V_i = T_i$ and $h_i = f_i$ for all $i \le \#(f)$. Again applying the inductive hypothesis, fields(bound$_{\Delta}$(S)$^{\eta}$) = S̄ ḡ and $S_i = V_i$ and $g_i = h_i$ for all $i \le \#(h)$. Since $\#(f) \le \#(h)$, the conclusion is immediate.
Case S-DIR S = C, T = D, class C extends D {S̄ ḡ; ...}. By the rule F-CLASS, fields(C) = Ū f̄; S̄ ḡ, where Ū f̄ = fields(D), from which the conclusion is immediate.	⟾	**Case GS-DIR** S = C⟨T̄⟩$^{\beta}$, T = [T̄/X̄]$^{\eta}$D⟨V̄⟩$^{\beta}$, C⟨X ◁ N⟩$^{\beta}$ extends D⟨V̄⟩$^{\beta}$ {S̄ ḡ; ...}. By the rule F-CLASS, fields(C⟨T̄⟩$^{\beta}$) = Ū f̄; [T̄/X̄]$^{\eta}$S̄ ḡ, where Ū f̄ = fields([T̄/X̄]$^{\eta}$ D ⟨V̄⟩$^{\beta}$). By definition, bound$_{\Delta}$(V) = V for all non-variable types V$^{\eta}$, from which the conclusion is immediate.

lots of micro changes

extend judgment signatures, modify premises and conclusions

add new syntax, modify syntax

Big Picture

- Ben's challenge:
  - start with a domain $\mathcal{FJ}$ of FeatherWeight Java dialects
  - constructed from a feature set $\overrightarrow{\mathcal{FJ}}$
  - goal is to automatically verify the type soundness property of any $\ell \in \mathcal{FJ}$ by composing modules for each feature's representation
- Scales homomorphisms to new heights…

Ben's Magic Sauce…

- All representations (syntax, typing rules, evaluation rules, theorems, proofs) are encoded in Coq Proof Assistant (CPA)

- Relies on 4 homomorphisms:

$$
\begin{aligned}
sn(A+B) &= sn(A) \;+_{sn}\; sn(B) \\
tp(A+B) &= tp(A) \;+_{tp}\; tp(B) \\
op(A+B) &= op(A) \;+_{op}\; op(B) \\
pf(A+B) &= pf(A) \;+_{pf}\; pf(B)
\end{aligned}
$$

Remember!

- Composing modules in CPA isn't syntax tree munging or projection
- Module is an open definition and module composition computes a fixed point
  - ask Shriram for details ☺

- Ben invented 2 operations $\oplus$ (for composing definitions) and another $\otimes$ (for composing proofs); both are defined in a CPA library

$$\delta(A + B) = \delta(A) \ \oplus \ \delta(B)$$

$$pf(A + B) = pf(A) \otimes \ pf(B)$$

Composition of Definitions

(syntax, typing rules, evaluation rules)

- Needed 2 levels of homomorphisms to make everything work – Look!

$$sn(A + B) \quad = \quad sn(A) \quad +_{sn} \quad sn(B)$$

```
e ::=  x                 e ::=  x
   |   e.f                  |   e.f
   |   e.m t (ē)            |   e.m t (ē)
   |   new C t (ē)          |   new C t (ē)
   ;                        ;
t  ::=                   t  ::= «T̄»
   ;                        ;
```

$+_{sn}$ `t ::= «`$\bar{T}$`»;`

Composition of Definitions

(syntax, typing rules, evaluation rules)

- Needed 2 levels of homomorphisms to make everything work – Look!

$$\delta(\, sn(A+B)\,) \;=\; \delta(\, sn(A)\,)\;\; \delta(+_{op}\,)\;\; \delta(\, sn(B)\,)$$
$$= \;\; \delta(\, sn(A)\,) \;\; \oplus \;\; \delta(\, sn(B)\,)$$

```
Inductive FJ_E : Set :=
  e_var : Var -> FJ_E
| fd_access : E -> F -> FJ_E
| m_call : m_call_ext -> E -> M -> Es -> FJ_E
| new : FJ_Ty -> Es -> FJ_E.
```

$\delta(\oplus_{op})$

```
Definition GTy_ext {ty_ext : Set} := (prod (list Ty) ty_ext).
```


Composition of Definitions

(syntax, typing rules, evaluation rules)

- Needed 2 levels of homomorphisms to make everything work – Look!

$$\delta(\, sn(A+B)\,) \;=\; \delta(\, sn(A)\,)\;\; \delta(+_{op}\,)\;\; \delta(\, sn(B)\,)$$
$$= \;\; \delta(\, sn(A)\,) \;\; \oplus \;\; \delta(\, sn(B)\,)$$

- Same for typing rules and operation semantic rules:

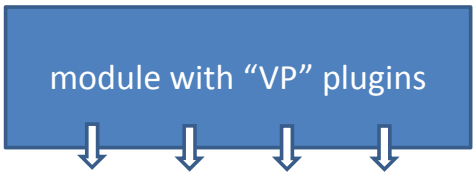
$$\delta(\, tp(A+B)\,) \;=\; \delta(\, tp(A)\,)\;\; \delta(+_{pf})\;\; \delta(\, tp(B)\,)$$
$$= \;\; \delta(\, tp(A)\,) \;\; \oplus \;\; \delta(\, tp(B)\,)$$

- And proof composition…

Semantic Composition ⊗

that guarantees the correctness of proofs

Case GS-TRANS [redacted] ⊢ S<:V and [redacted] ⊢ V<:T.
By the inductive hypothesis, fields([redacted]) = $\overline{V}\ \overline{h}$ and $V_i = T_i$ and $h_i = f_i$ for all $i \leq \#(f)$. Again applying the inductive hypothesis, fields([redacted]) = $\overline{S}\ \overline{g}$ and $S_i = V_i$ and $g_i = h_i$ for all $i \leq \#(h)$. Since $\#(f) \leq \#(h)$, the conclusion is immediate.

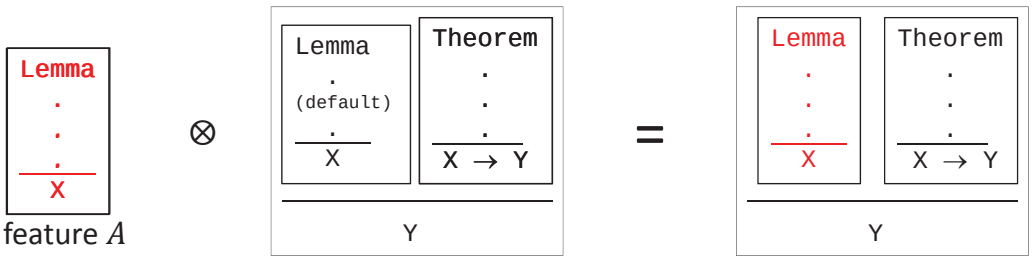


- Ben defined properties per VP that must be satisfied by *any* VP plug-in
  - stated as additional assumptions with default lemma(s)
- Allows a general theorem(s) to be proven per feature, independent of features that might "plug-in" specific definitions for its VPs
  - *in effect, the proof assumes a general behavior for all possible VP instantiations*
- Obligation: any feature that "plugs-in" a VP 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
- Once you certify general theorem, do not recertify, reuse as is
- Once you certify plug-in theorems, do not recertify, reused as is
- Must certify that general assumptions hold for plugins



composition overrides default lemma and replaces it with new lemma

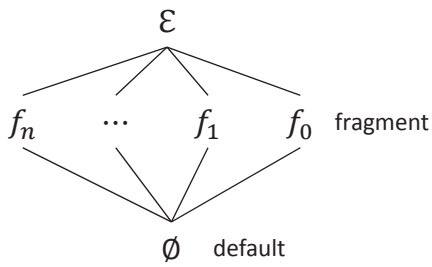
Summarizing

- Recall what I said about virtual/classical modules?
- Ben illustrates modules by coloring
- Actually uses language modules
- Given a feature expression can produce the target language's proofs of preservation and progress modularly, verifying proofs by "interface" checks

META THEORY AGAIN…

Work with Höfner and Möller

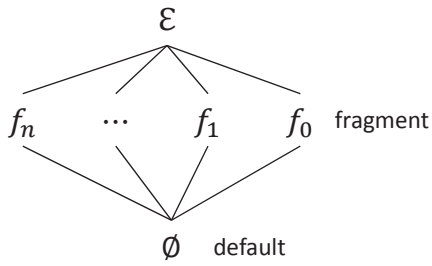
- What is going on in coloring?
- Look at the contents of a VP – given what we've seen in Ben's work



$$a + \emptyset = a$$
$$a + b = b + a$$
$$(a + b) + c = a + (b + c)$$

Work with Höfner and Möller

- What is going on in coloring?
- Look at the contents of a VP – given what we've seen in Ben's work



- Lattice: its join operation is our addition operation $(+)$:
  - identity $\emptyset + a = a$
  - commutative $a + b = b + a$
  - associative $(a + b) + c = a + (b + c)$

This is why + works as it does

Your Thoughts on Analyses

Jo Atlee: “Efficient analysis of entire product lines (v.s. analysis of products) – it ought to be more efficient than the work we’ve seen so far”

Vander Alves: “(we need) efficient and precise analysis of product lines”

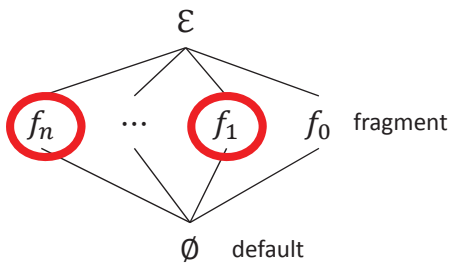
Roberta Coelho: “The current infrastructure for static analysis does not take into account that each piece of code may be related to one or more features. As a consequence, each tool was developed its own way to deal with features during static analysis. A common infra-structure should be developed.”

Sandro Schulze: “Analyzing variable software systems/software product lines with respect to metrics and code smells”

Ina Schaefer: “Analyze incomplete artifacts, such as feature modules or deltas”

We Know…

- That the features F_i that add non-default fragments must be mutually exclusive



$featureOf(f_n) \Rightarrow \neg featureOf(f_1)$ $\quad F_n \Rightarrow \neg F_1$

$featureOf(f_1) \Rightarrow \neg featureOf(f_n)$ $\quad F_1 \Rightarrow \neg F_n$

Check Compositions Statically

Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints

GPCE 06

Krzysztof Czarnecki Krzysztof Pietroszek
University of Waterloo, Canada
{kczarnec,kmpietro}@swen.uwaterloo.ca

Safe Composition of Product Lines

GPCE 2008

Sahil Thaker David Kitchin, and William Cook
of Computer Sciences
of Texas at Austin
exas, 78712 U.S.A.

- Let ϕ be the propositional formula of a feature model
- Let $atmost1(F_1, F_2, \ldots, F_n)$ be the prop formula that says at most one of the features can be chosen
- SAT solver can efficiently verify: $\phi \Rightarrow \neg atmost1(F_1, F_2, \ldots, F_n)$
- Proving all $p \in \mathcal{D}$ have no such composition errors is easy and efficient

Your Thoughts on Interactions

Chris Lengauer: "(how to) specify … feature interactions, have 'structured programming' with features"

Sven Apel: "(how to) detect, resolve, and manage feature interactions"

Bernhard Möller: "(we need) a good algebra for treating feature interaction"

Krzysztof Czarnecki : "(how to) understand and handle feature interactions (presence of one feature influences the behavior and/or performance of another feature) in complex systems."

Feature Interactions

- Are NOT feature dependencies, like:

$$F \wedge G \Rightarrow H \vee K$$

- Which arise (or are part of) feature models
- A 2-way interaction (or rather resolution) is an additional module $A\#B$ that is needed to integrate features A and B so that they work correctly together
  - mediator
  - coloring – it is the code that has the colors of both A and B
- Generalizes to n-way interactions

Feature Interactions

- There are other operations on features besides addition (+)
  - multiplication (×)
  - interaction (#)
- Instead of adding features, we multiply them instead $P = A * B * C$
- And one axiom to rule them all:

$$A \times B = A\#B + A + B$$

"the product of 2 features is their sum plus any additional interaction that makes them work together correctly"

Prior Discussions Extend Naturally

- Homomorphism on addition:

$$\theta(A + B) = \theta(A) \quad +_\theta \quad \theta(B)$$

- Easily generalizes to a homomorphism on multiplication

$$\theta(A \times B) = \theta(A) \quad \times_\theta \quad \theta(B)$$

Your Thoughts on Testing

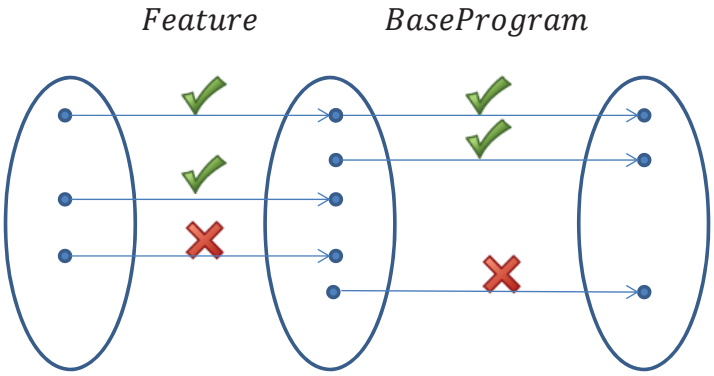
Holger Schlingloff: “(how do we) reuse testing artifacts within a SPL?”

Salva Trujillo: “(how do we) test inter-related design models with variability”?

Paulo Borba: “lack of efficient techniques for PL testing”

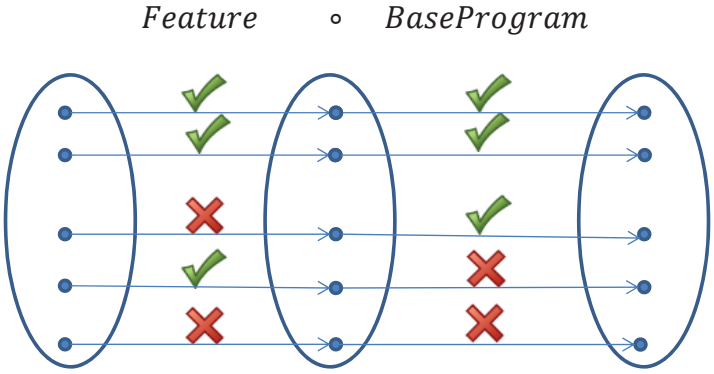
Gilles Perrouin: “(how do we) select tests for variable-intensive systems? Combinatorial test interaction techniques have made progress to significantly reduce the number of configurations to consider for testing and how to prioritize them?”

Problem With Testing



- Maxim: Testing programs only shows the presence of errors, not their absence
- Maxim: Testing features can only show the presence of errors, not their absence

Locating Errors in Compositions



I could not solve this problem!

- Problem: Last 3 seem indistinguishable – not obvious whether error is in:
  - feature, base program, or both
- Program assembled from n optional features, theoretically error could spread across any combination of features – $O(2^n)$ possibilities

Meta-Theory Again

- MT has something to say w.r.t. tests
- Yesterday's talk: Martin Johannsen used a simple homomorphism:

$$\tau(A + B) = \tau(A) \cup \tau(B)$$

"the tests of a program are the union of the tests for each of its features"

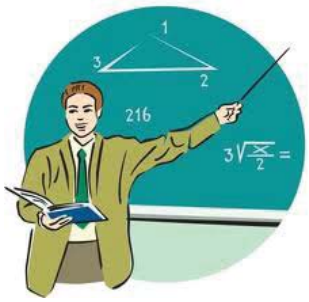
- More can be done…

You Have The Ideas Now…

Thomas Thüm: "how do we apply Design by Contract to SPLs, (as it allows one) to split a large verification problem into manageable pieces. How can we generate contracts, which could be used as documentation, for testing, and verification."

- Remember what Ben Delaware did: He required precise properties to be satisfied by a plug-in – this gave him the modularity that he needed
- Is Design by Contract the next step?

Yes!



Prior lectures by Ina and Thomas covered recent results

even after all this

FUNDAMENTAL PROBLEMS STILL REMAIN!

Fundamental Problems Still Remain!

- Can we have multiple copies of a feature?
  Are features classes that can be instantiated?

Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled

Kacper Bąk[1], Krzysztof Czarnecki[1], and Andrzej Wąsowski[2]

- Q: is this the right way to go?
- how will this generalize MT?

- you will discover the answer!

Fundamental Problems Still Remain!

- How does Model Driven Engineering fit in?
  - survey answers focused on “code”, few on “models”
  - MDE is where the future lies… Is a much more general framework in which to understand activities of software engineering in general and variability in particular
  - how will this generalize MT?
  - you will discover the answer…

Fundamental Problems Still Remain!

Paulo Borba: “lack of support for evolving (refactoring, maintaining, etc.) product lines”

Sandro Schulze: “refactoring in the presence of variability”

- How can we have (meta-)theories of variability without including refactorings?
- Major hole in our knowledge
- How will this generalize MT?
- You will discover the answer!

Fundamental Problems Still Remain!

Jörg Liebig: “how do we deal with large feature models (slicing)?”

Tiziana Margaria: “we really miss good case studies. There is not much out there that is realistically designed in this way, where we in academia can get ahold of the design”

Alexander von Rhein: “Scaling existing (analysis) techniques to really large product lines (linux kernels). (There are steps that we can take…) but they do not scale to the linux kernel.”

Christian Kästner: “how did developers implement 10K features (representing an unbelievably huge configuration space) with so few variability bugs – (perhaps we might learn from them) how to design and implement variations”

There are Many, Many More

FUNDAMENTAL PROBLEMS STILL REMAIN!!

A Last Word from Me…

Programmers are geniuses at making the simplest things look complicated – the hard part is finding the simplicity

- There is clear value in established thinking, but it is NOT everything

- Even a broken watch is correct twice a day

And A Final Thought from Woody

FUNDAMENTAL PROBLEMS STILL REMAIN!!


hvala
mulțumesc
Köszönjük!
danke!
tack!, tack så mycket!
merci!
citosl. děkuji
Благодарим ви
Takk
dank u!
ΣΑς ΕΥΧΑΡΙΣΤΩ
pakka þér
diolch i chi
THANK YOU!
mange tak!
Hvala vam na pažnji!
спасибо
дзякуй
gracias!
GO RAIBH MAITH AGAT
GRAZIE
obrigado!
falemnderit
Rzecz. podziękowanie
Aitäh
Dagstuhl2013-69