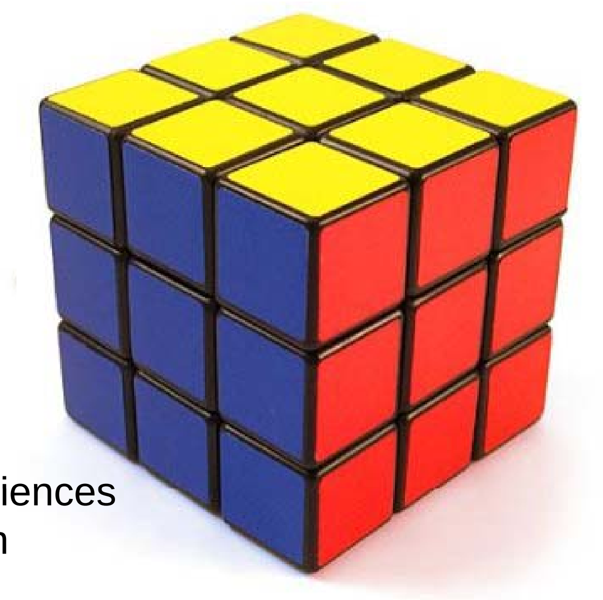
Program Kubes



Don Batory
Department of Computer Sciences
University of Texas at Austin

Introduction

- Future of **software development (SWD)** lies in automation
    - automate rote, time-consuming, error-prone tasks
    - three technologies that automate such tasks will converge
- **Model Driven Engineering (MDE)**
    - specify target program by a set of high-level models which are easy to understand, write, and maintain
    - program is synthesized by transforming models to executables
- **Refactoring**
    - reorganize programs/models using transformations to improve structure
- **Software Product Lines (SPL)**
    - create a family of related programs from a common set of assets
    - automatically synthesize an SPL member from a declarative specification

Convergence

- Unified by **transformations**
  - mapping of programs to programs (or models to models)
  - MDE – map models of one type to another
  - Refactoring – restructure models
  - SPL – elaborate models by adding more detail
- Emerging **Science of Automated SWD** from experiences of practitioners
  - **compositional** – based on function (transformation) composition
  - fundamental mathematical structures provide an **informal language** to express program/model designs
  - SPL modeled by categories (POPL 2007, ICSE 2007, GPCE 2008)
  - theoretical basis for future tools and models for synthesis

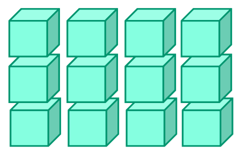
Why Transformations?

- Java and C# programs use methods to update and translate objects
  - "programming in the small"
- In SWD, objects are programs and methods are transformations
  - "programming in the large"
- Transformations provide a fundamental way to understand SWD
  - foundation for MDE, refactorings, and software product lines
- Thinking mathematically leads to unusual design techniques that take time to understand
  - this talk is about one particular example

Long, Long Ago…

- In 2001, we (Lopez-Herrejon, et al) discovered a multi-dimensional structure to express interacting dimensions of variability in SPLs
- Called "Origami"
  - design of a program P was defined by a matrix of transformations
  - rows, columns were features
  - matrix was folded in precise ways (thereby composing transformations) until a single term was produced
  - this term is an expression (composition of transformations) that synthesizes P

P =

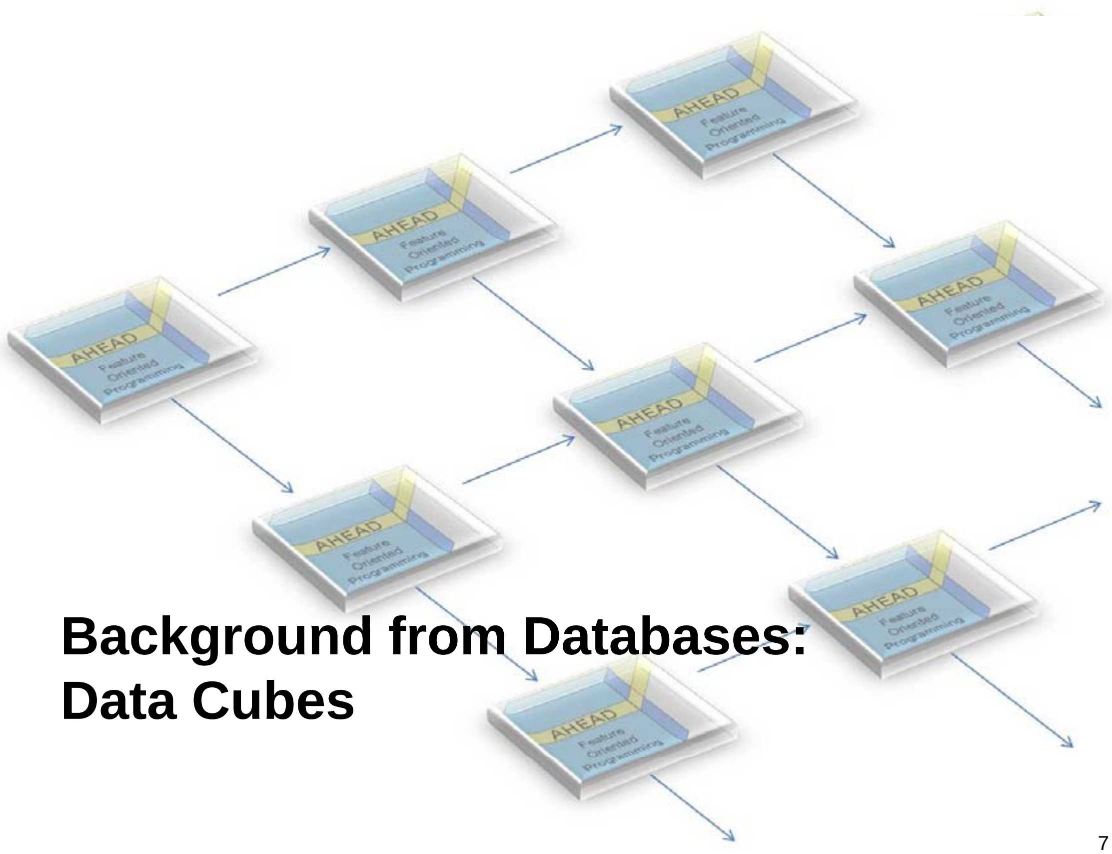


technique that others will use in the future

- **Essential concept in building ATS (250K+ LOC)**

"This Works???"

- Take a "physics" approach – small number of principles (in mathematical form) that hold this entire universe together
- Origami is sophisticated technique – taken years to connect seeming unrelated topics that it integrates
  - data cubes — database technology
  - tensors, categories — mathematics
  - expression problem — programming languages
  - feature interactions — software design
- This is a modeling talk aimed at practitioners
  - no special mathematical background
  - **program synthesis and variability expressed by modern mathematics**
  - **raise the level on how to think about SWD and variability**
- Your tour guide through a strange universe…



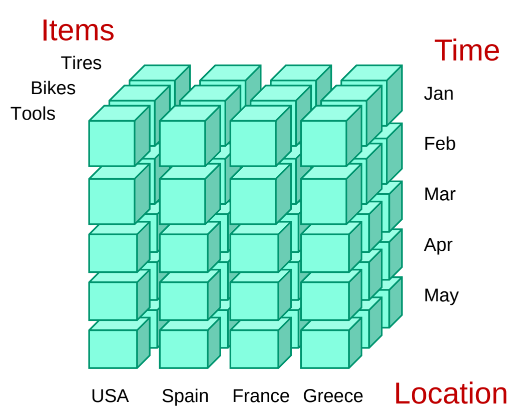
Background from Databases: Data Cubes

Data Cubes

- **Data Cubes** (Gray 1997) are a multi-dimensional array visualization of relational tables for data warehouses

Dimension Attributes — Measures Attribute

Items	Location	Time	QtySold
tools	usa	feb	4
tires	spain	jan	3
bikes	france	may	30
…	…	…	…

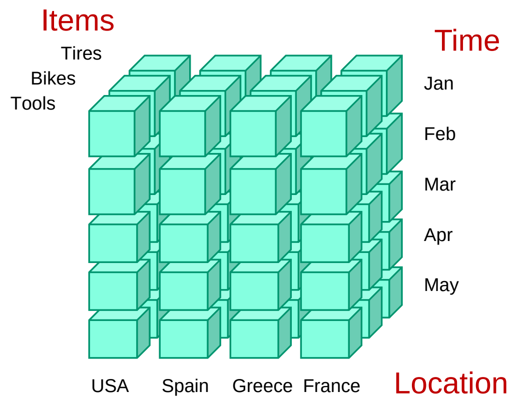


Data Cubes

- **Data Cubes** (Gray 1997) are a multi-dimensional array visualization of relational tables for data warehouses
  - tuples are cube entries

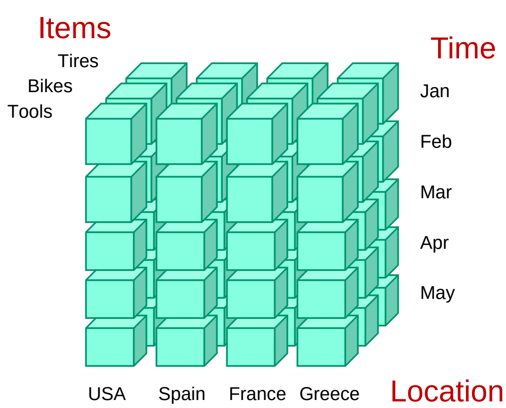
Dimension Attributes | Measure Attributes

Items	Location	Time	QtySold
tools	usa	feb	4 4
tires	spain	jan	3 3
bikes	france	may	30 30
…	…	…	…



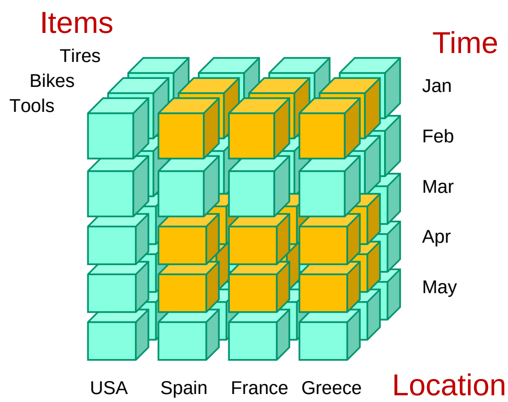
Cube Queries

- Subcubes restrict dimensional values
  - count # of bikes, tools sold in Europe in January, March, April



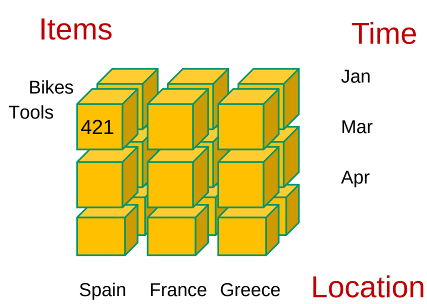
Data Cube Operations

- Projection eliminates unnecessary elements
  - count # of bikes, tools sold in Europe in January, March, April

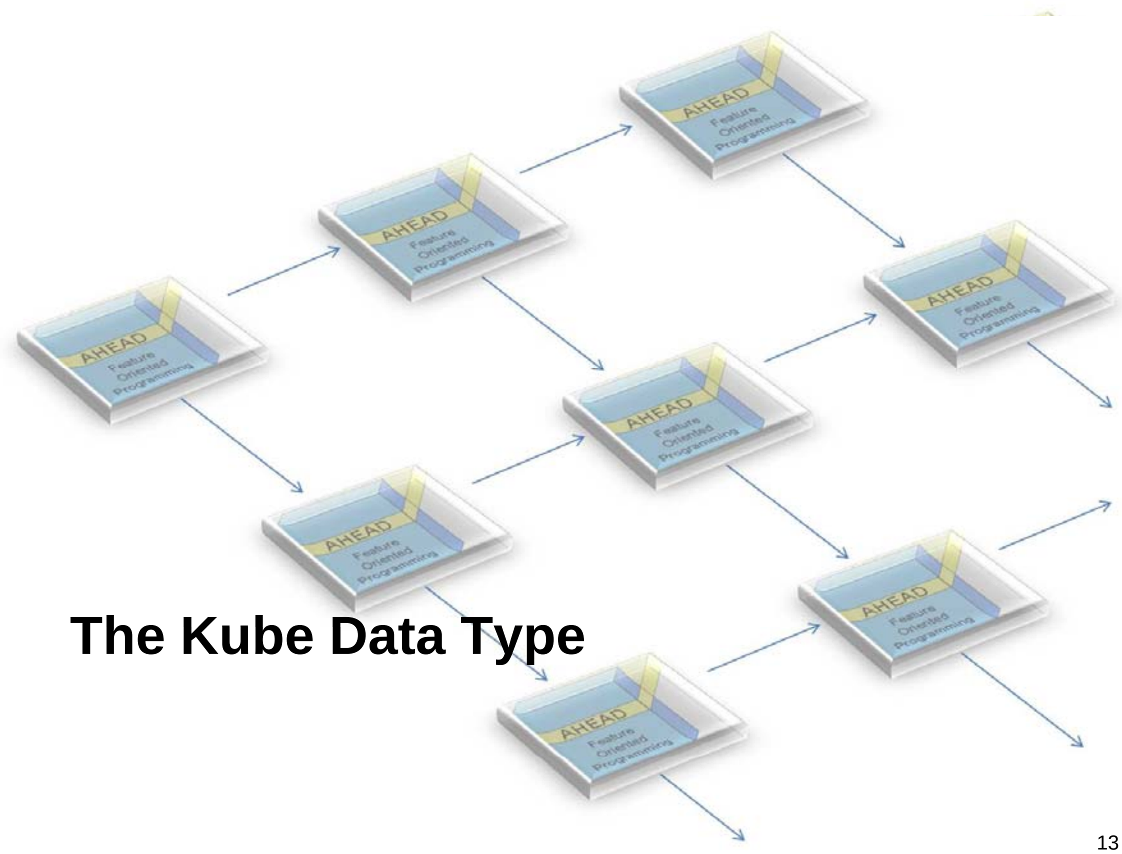


Data Cube Operations

- **Aggregate** or **roll-up** elements to produce a scalar
  - scalar totals the # of bikes, tools sold in Europe in January, …



Order in which dimensions are aggregated doesn't matter



The Kube Data Type

Kubes: N-Dimensional Arrays

- Borrow terminology of tensors
  - **rank** of **kube** is its dimensionality
  - scalar is a kube of rank 0
  - vector is a kube of rank 1
  - matrix is a kube of rank 2

- Notation: # of indices = rank of kube

S	– kube of rank 0	(scalar)
V_k	– kube of rank 1	(vector – linear array)
M_{ij}	– kube of rank 2	(matrix – 2D array)
C_{ijk}	– kube of rank 3	(cube – 3D array)

Kube Operations

- Roll-up is **contraction**
  - summing across dimensions i, j for kube C_{ijk}:

$$V_k = \Sigma_{ij} C_{ijk}$$

- Interested (in this talk) on contracting to scalars

$$S = \Sigma_{ijk} C_{ijk}$$

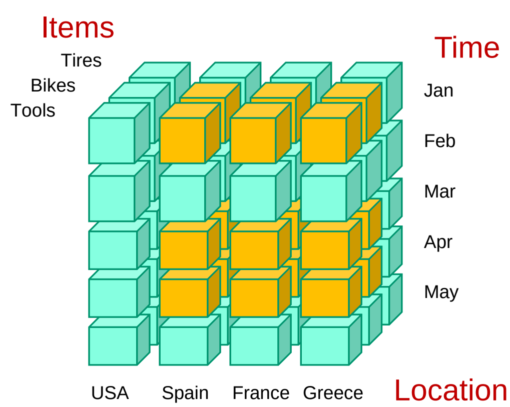
order in which dimensions are contracted does not matter

- Projection limits values of indices

$$S = \Sigma_{i\in I,\ j\in J,\ k\in K} C_{ijk}$$

Previous Example

$$C_{ilt} =$$



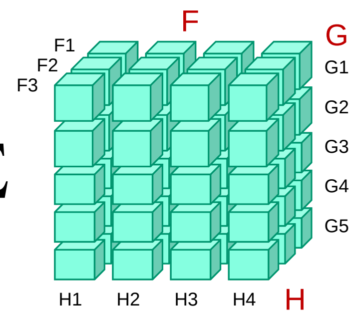
$$S = \Sigma_{i\in\{\text{Bikes,Tires}\}\ l\in\{\text{Spain,France,Greece}\}\ t\in\{\text{Jan,Mar,Apr}\}} C_{ilt}$$

Software Synthesis

- Kube expresses multidimensional models of variability

- Data cubes aggregate numbers,
  program synthesis composes transformations
    - element is a program transformation
    - dimensions we will see are defined by feature models

Program = Σ



Program Cube

Example: AHEAD Tool Suite

- Language, tool extensible IDE
    - synthesize tools for dialects of Java
    - optional extensions to Java

```
State_machine SM {

   States g, h, i;

   Transition e1 : g → h ...;

   Transition e2 : h → i ...;

   ...
}
```

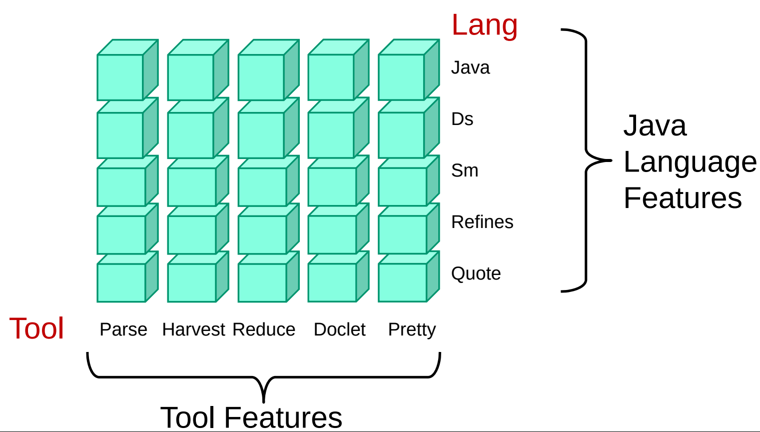
state machine

```
refines class CL {
   ...
}

refines interface IN {
    ...
}
```

refines

AHEAD Tool Suite

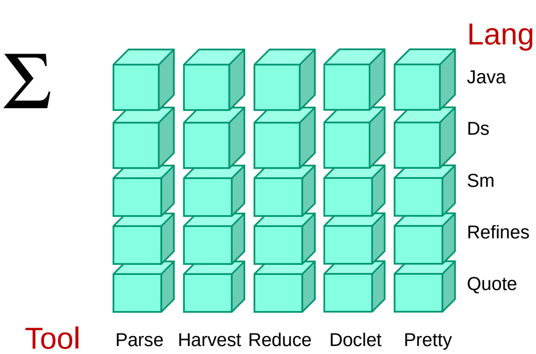
- Language, tool variability expressed as a 2D kube
  - ASE 2002 and SIGSOFT/FSE 2003



AHEAD Tool Suite

- AHEAD Tool specified by 2 sets of features
  - **Jedi** – javadoc tool for a dialect of Java

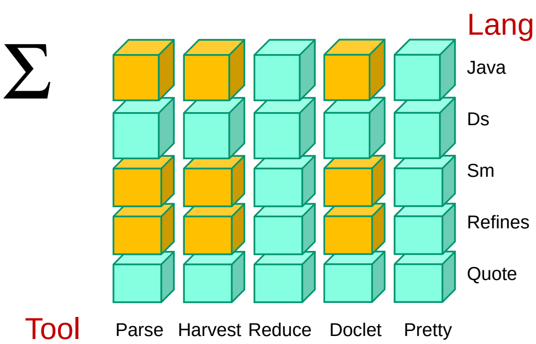
Jedi = Σ



Synthesize Jedi

- Project matrix

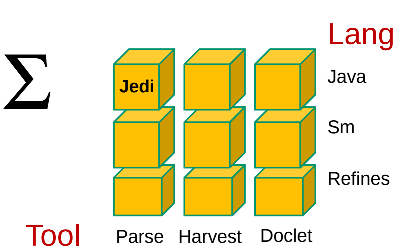
Jedi = Σ



Synthesize Jedi

- Contract matrix
- Defines composition of transformations to synthesize Jedi

Jedi = Σ

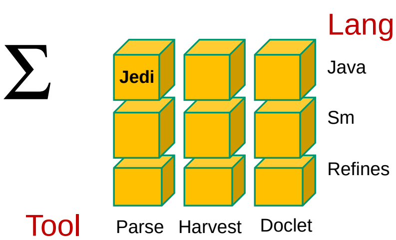


Most AHEAD Tools (250K+ LOC) are built by contracting 2D, 3D kubes; see ASE 2002, FSE 2003 papers

Here's What's Odd…

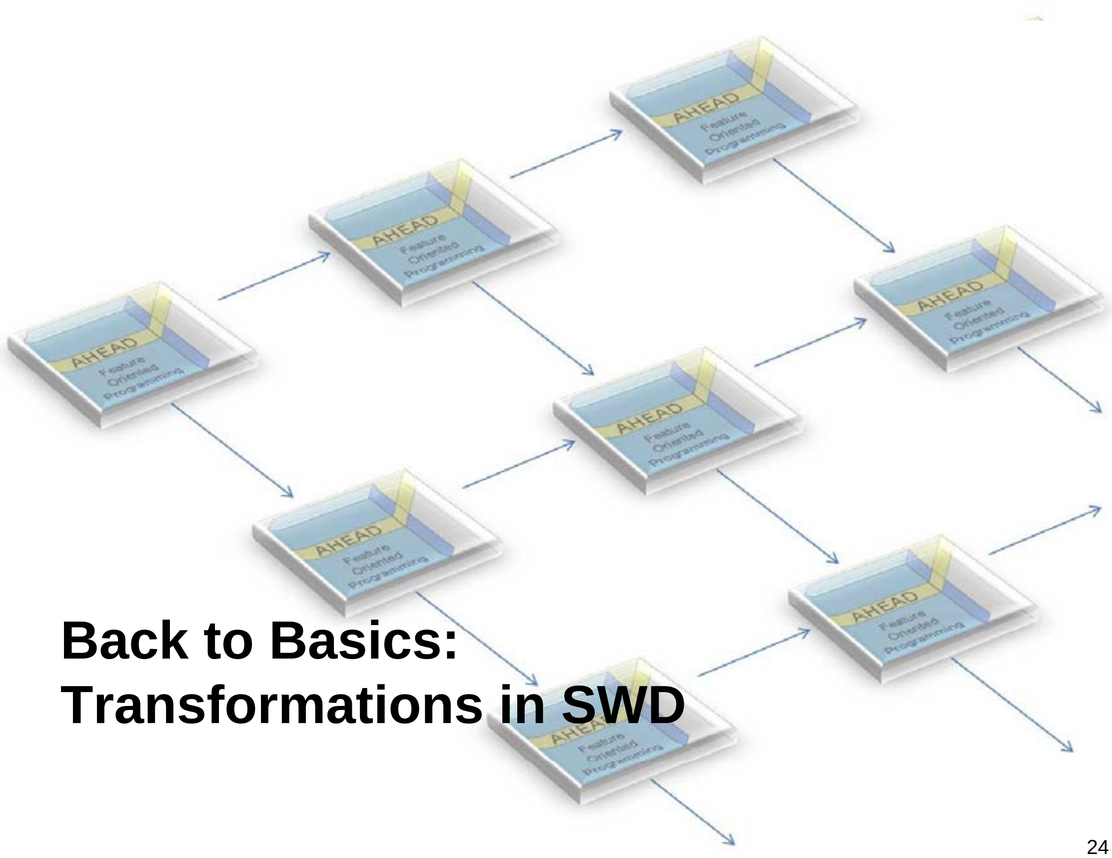
- Lots of other ways to contract matrix
  - Each contraction produces a different expression
  - Function composition (unlike integer addition) is **not** commutative

Jedi = Σ



Why all foldings (contractions) **produce the same result** (program) **?**

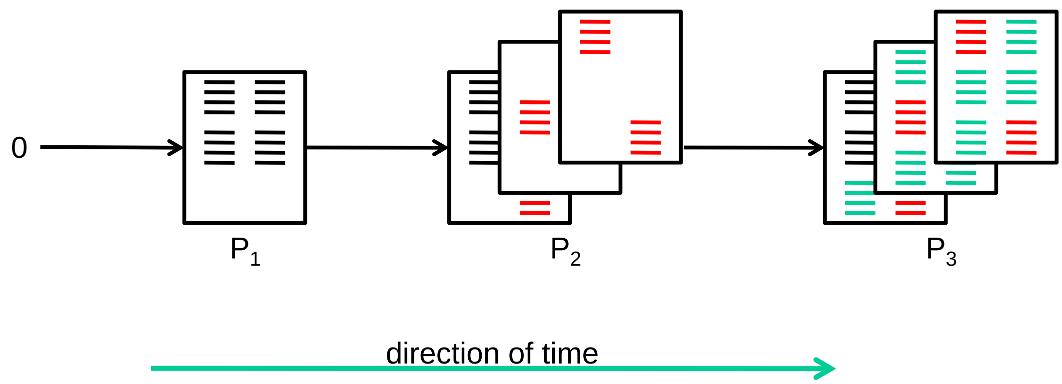
2007



Back to Basics: Transformations in SWD

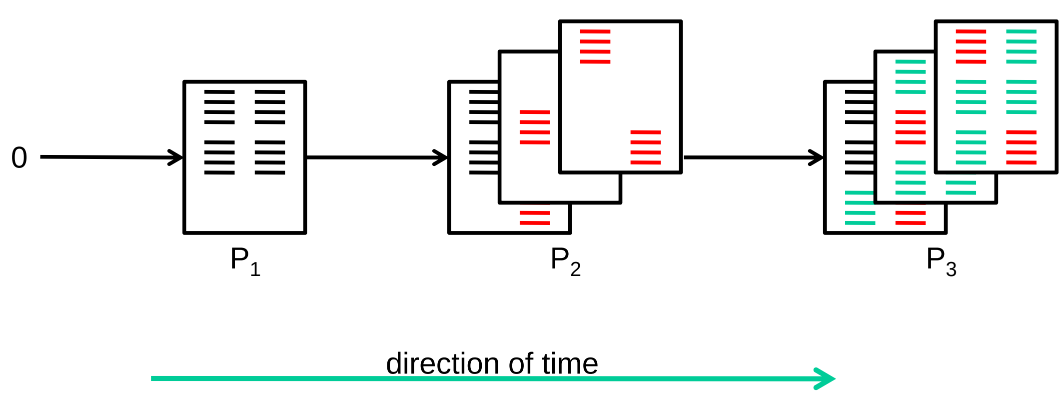
Basics

- **Fact: no program is created spontaneously**
  - no MDE model, Word document, etc…
  - created by extending simpler programs
  - and simpler programs come from simpler programs, recursively



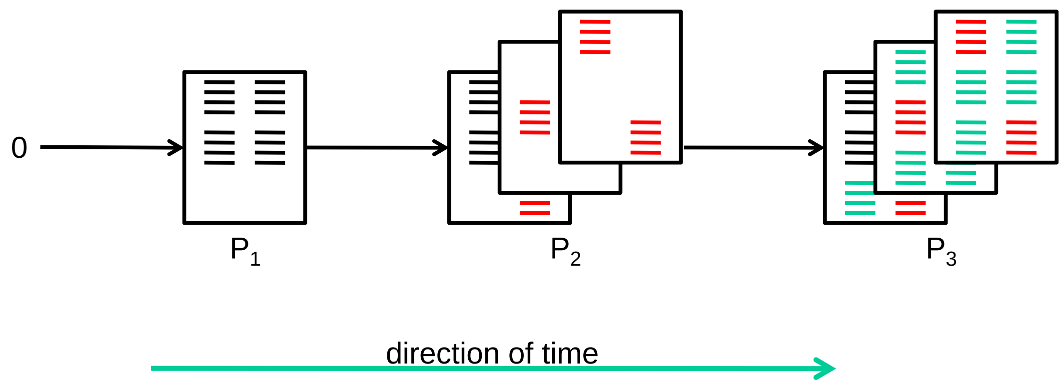
Incremental Development

- Going forward in time explains how a program's design was developed incrementally, in logical steps
  - **incremental development**
  - hallmark of automated program synthesis



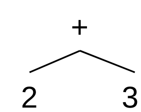
Timelines

- Think of arrows as **transformations**
  - mappings from one program to another
  - path from 0 to a program is its **timeline**
  - when arrows are given semantic meaning and are reusable they are called **features**

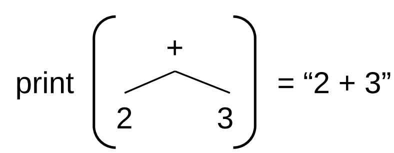


Example: Expression Trees

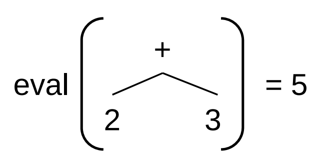
- Parse tree of an expression



- print() operation on trees:

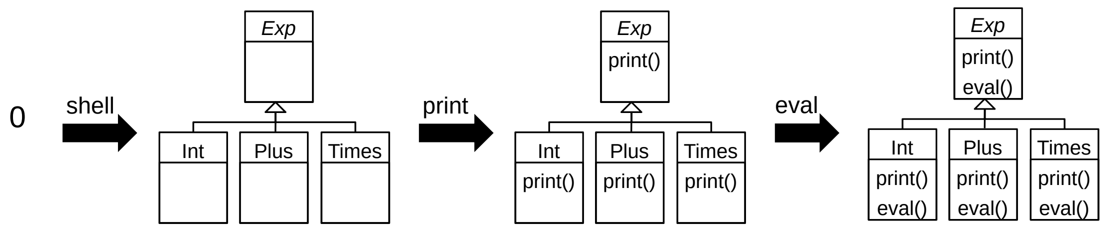


- eval() operation on trees:



Expression Tree Program

- Arrow either defines base program or adds operations

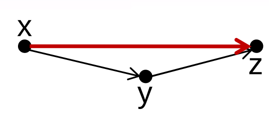


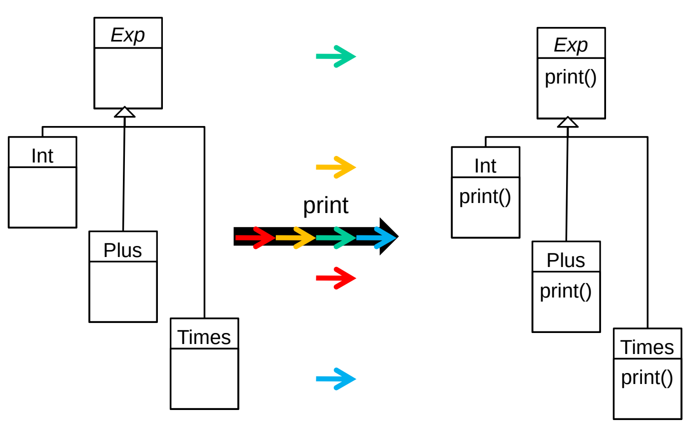
Methods Product Line

ET

Properties of Arrows

- Arrows are composable





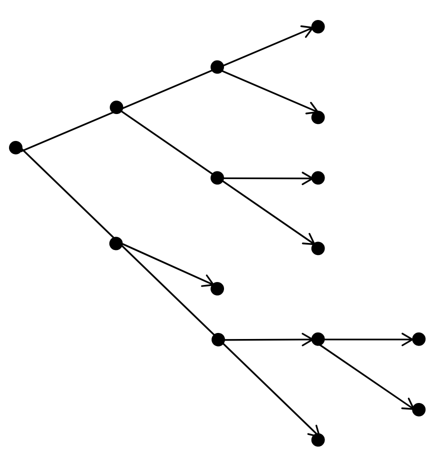
Software Product Lines

- Family of related programs
- **Features** are increments in program functionality (arrows)
  - SPL building blocks
- Conceptualize SPL as a directed graph
  - nodes are programs, arrows are features
  - paths from 0 are program **timelines**
  - superimpose the timelines of all programs in an SPL

Methods Product Line Graph

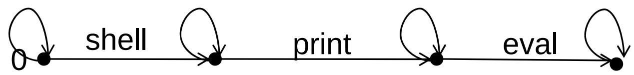
Typically

- Product line graphs are trees
  - one timeline for every program in a product line



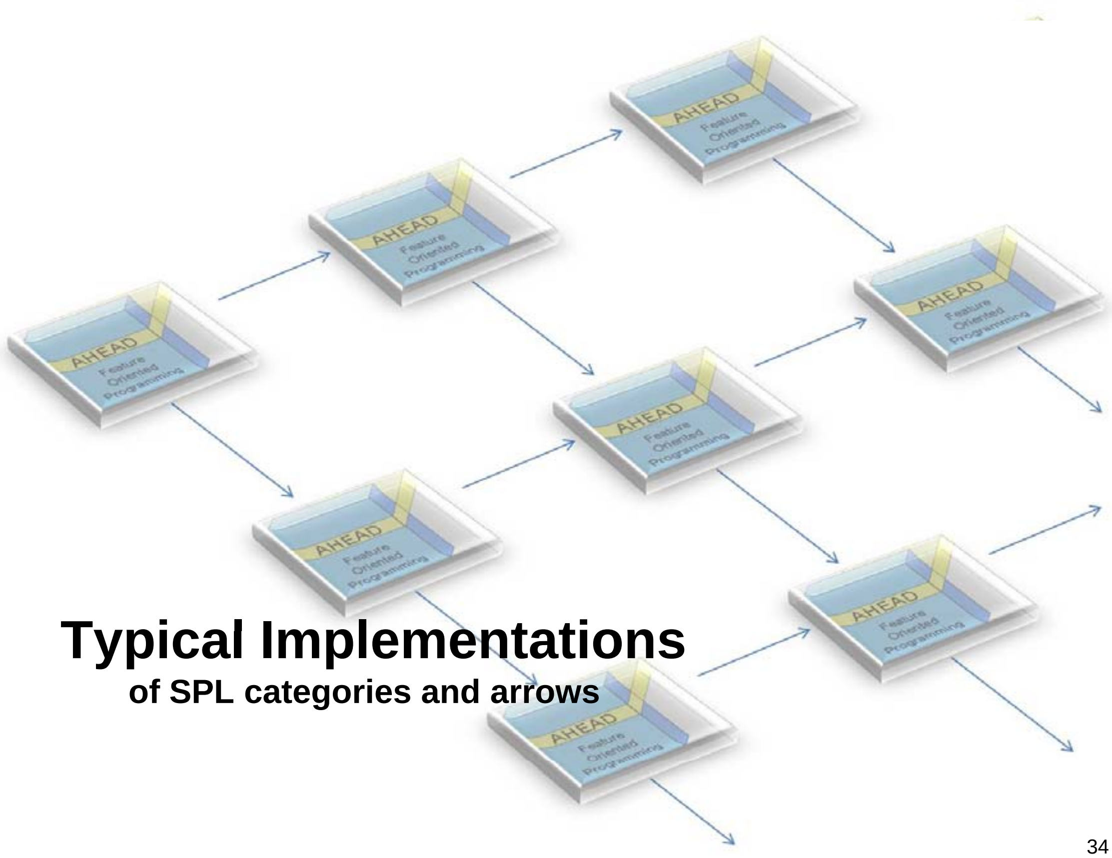
Categories

- Add identity arrow to each node



Category of the Methods Product Line

- Resulting graph (& arrow composition rules) is a **category**
  - object is a domain, arrow maps domain to co-domain
- Software product line = **micro (trivial) category**
  - each domain contains one element

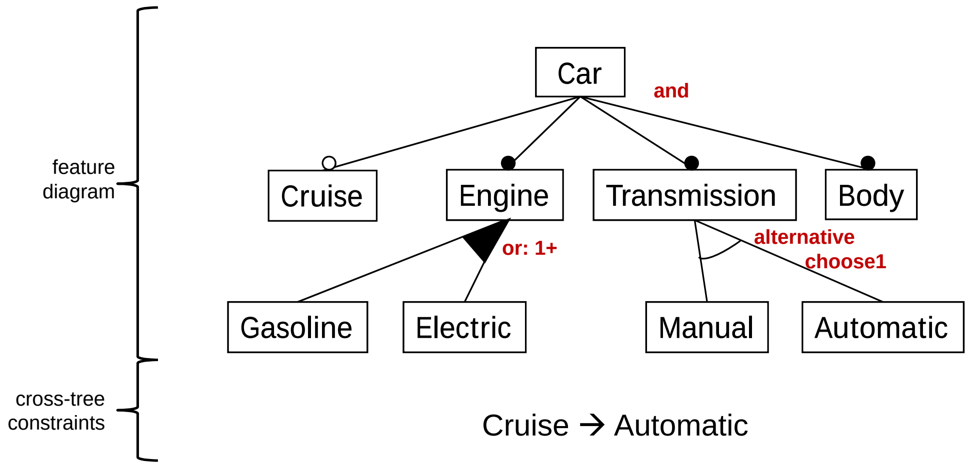


Typical Implementations

of SPL categories and arrows

Feature Models

- Standard representation of SPLs (Kang 1990)
  - and-or tree with cross-tree constraints
  - defines legal **combinations** of features

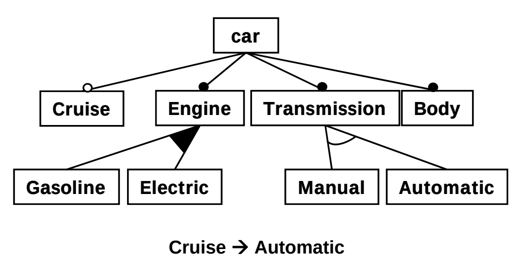


Ordered Feature Models

- Order in which features (arrows) are composed matters!
- Ordered feature model is a "GenVoca" grammar
  - tokens are features
  - productions define sentences
  - order in which tokens appear in sentences = order of composition

```
car : [cruise] engine+ trans body;

engine : gasoline | electric;

trans : manual | automatic;

##

cruise → automatic;
```

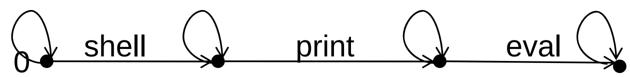
GenVoca Grammar



Feature Model

Encode Graph of SPL

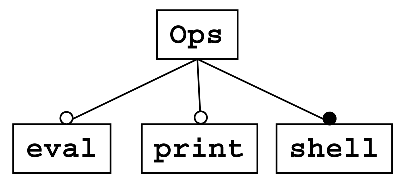
- As an ordered feature model



Methods Category/Graph

```
Ops: [eval] [print] shell;

##

eval → print;
```

GenVoca Grammar

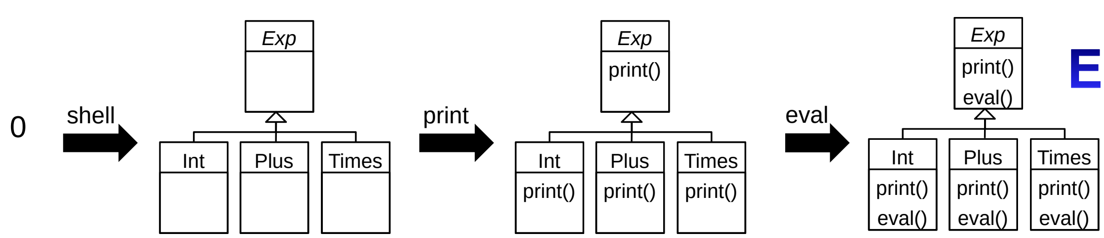


eval → print

Feature Diagram

Feature Implementations

- Arrows can change **at least** the following:
  - add new classes, packages
  - add new members (fields, methods) to existing classes
  - wrap existing methods …
- Lots of prototype technologies to do this….



ET

Methods Product Line

SPL Implementation

- Collection of arrows (features) whose compositions produce programs of an SPL is a **GenVoca Model**

$$F_i = [\ F_n \ldots F_3, F_2, F_1\ ] \qquad \text{// vector}$$

- Program P of an SPL is a particular composition of arrows

$$P = F_8 + F_4 + F_2 + F_1 \qquad \text{// P's timeline}$$

where + denotes feature (arrow) composition

Connection to Kubes

- Given $P = F_8 + F_4 + F_2 + F_1$
- Rewrite P as:

$$P = \Sigma_{i \in (8,4,2,1)} F_i$$

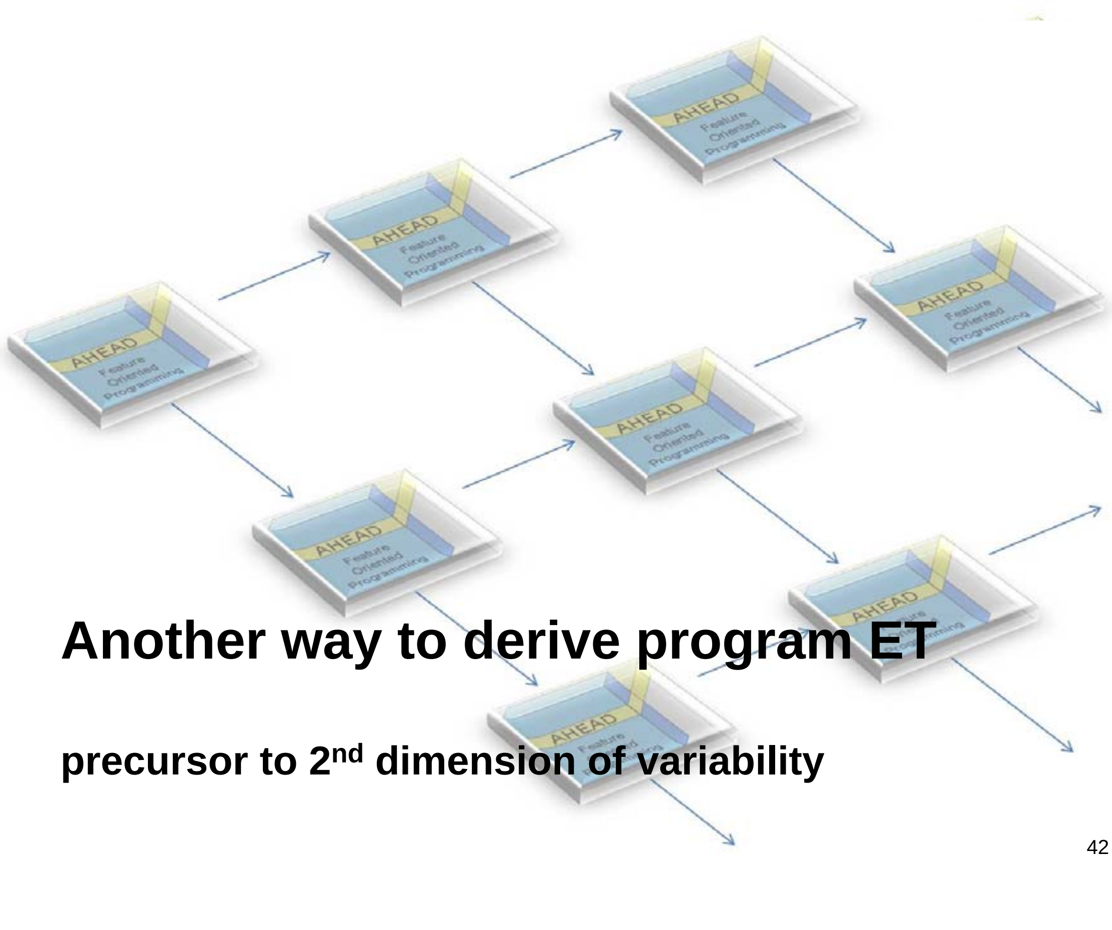
- Program synthesis is projection and contraction of 1D kube
- GenVoca grammar defines legal kube projections

Recap So Far…

- Conceptualize a SPL as a micro category
  - quirk: programs and objects are computed
- Implement by ordered pair:

SPL = (Grammar, Kube)

- 1 dimension of variability ➔ Kubes are vectors

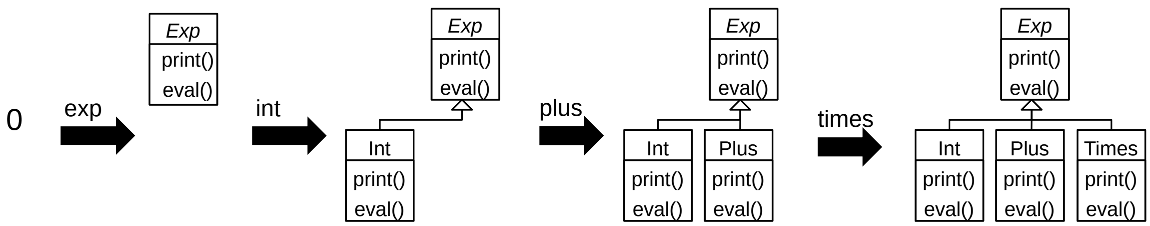


Another way to derive program ET

precursor to 2nd dimension of variability

Data Type Extensibility

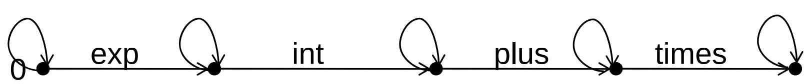
- Variability in the Methods SPL were methods a class supported
  - set of Exp subclasses (Int, Plus, Times) were fixed
- Another variability is data types
  - fix the set of methods
  - vary the set of subclasses of Exp



Types Product Line

ET

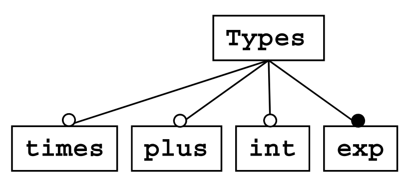
Types Product Line



Types Category/Graph

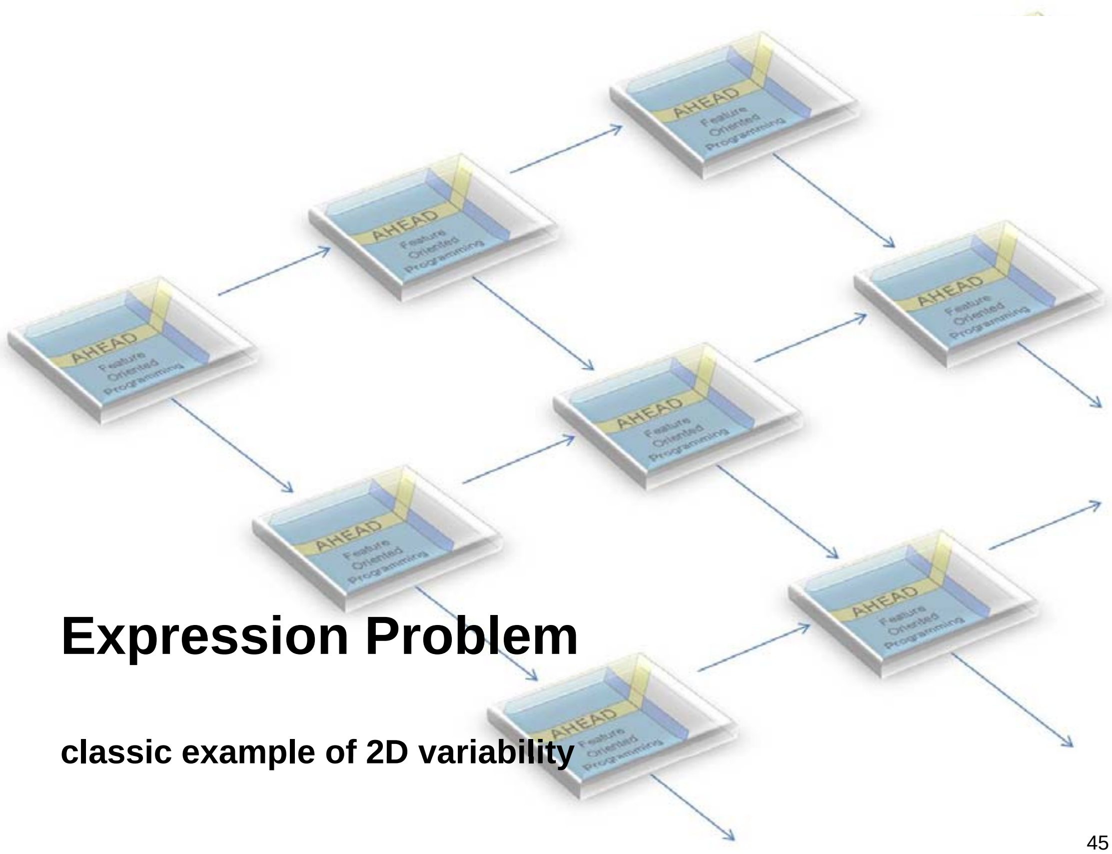
```
Types: [times] [plus] [int] exp;

##

plus → int;
times → plus;
```

GenVoca Grammar




```
plus → int
times → plus
```

Feature Diagram



Expression Problem

classic example of 2D variability

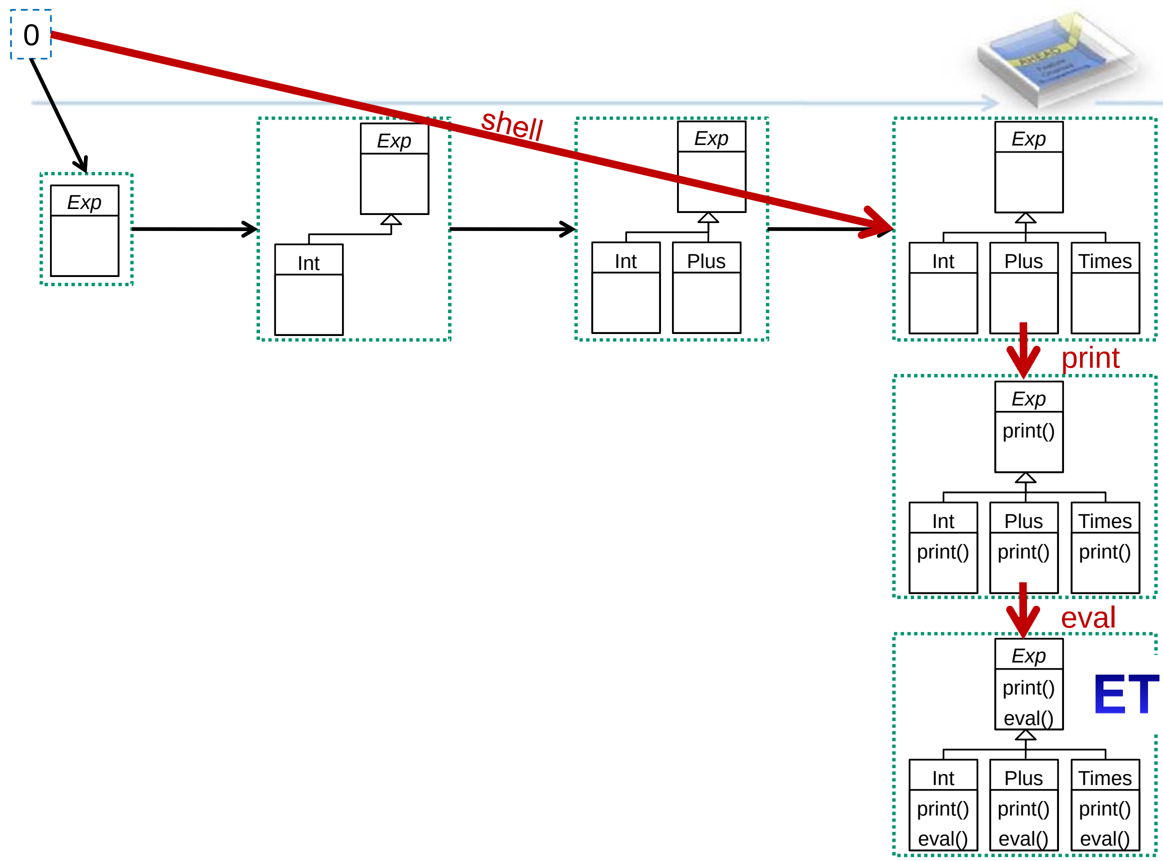
Expression Problem

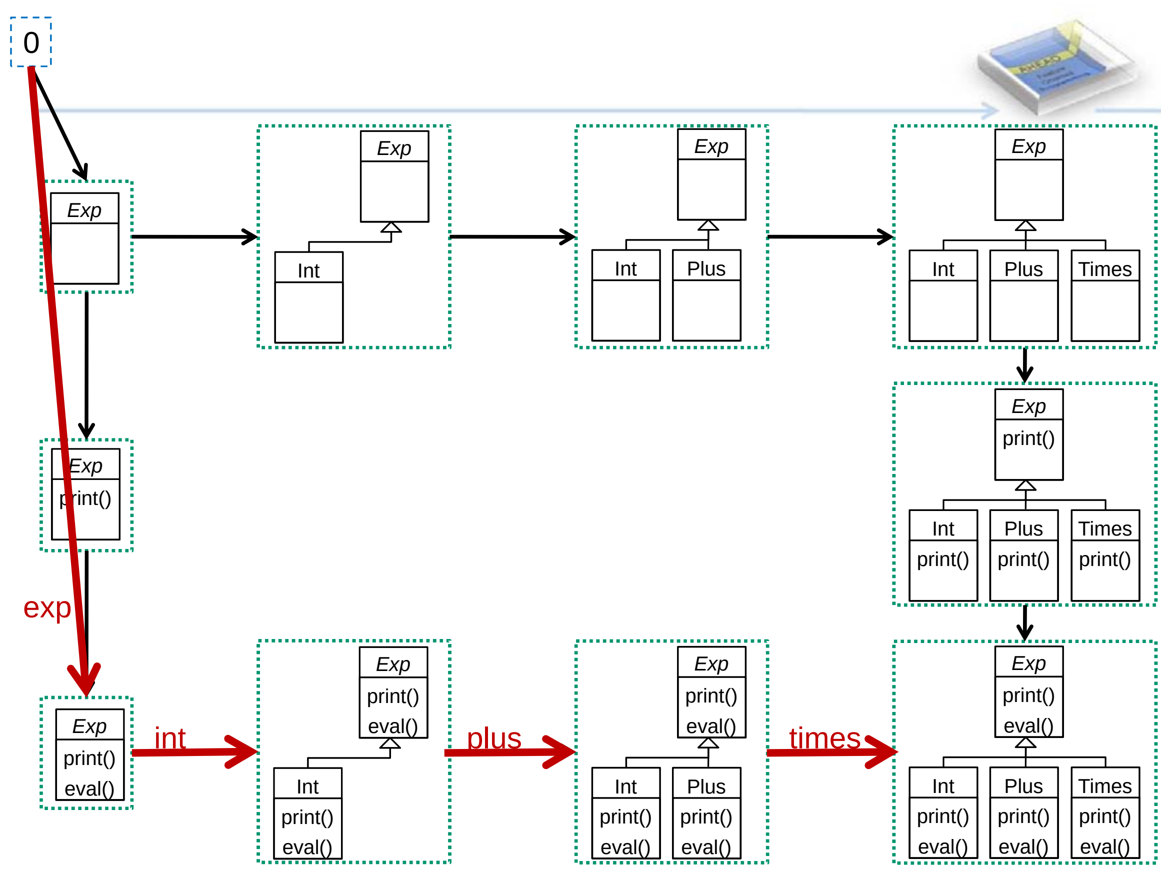
- Fundamental problem of program design and variability
  - 1975 J. Reynolds
  - 1990 W. Cook
  - 1998 S. Krishnamurthi, M. Felleisen, D. Friedman
  - 1998 P. Wadler

  - how to add new methods and data types in a type safe manner
  - SPL of 30-40 line programs

- ASE 2002, SIGSOFT/FSE 2003
  - R. Lopez-Herrejon, D. Batory, J-P. Martin, J. Liu, J. Sarvela

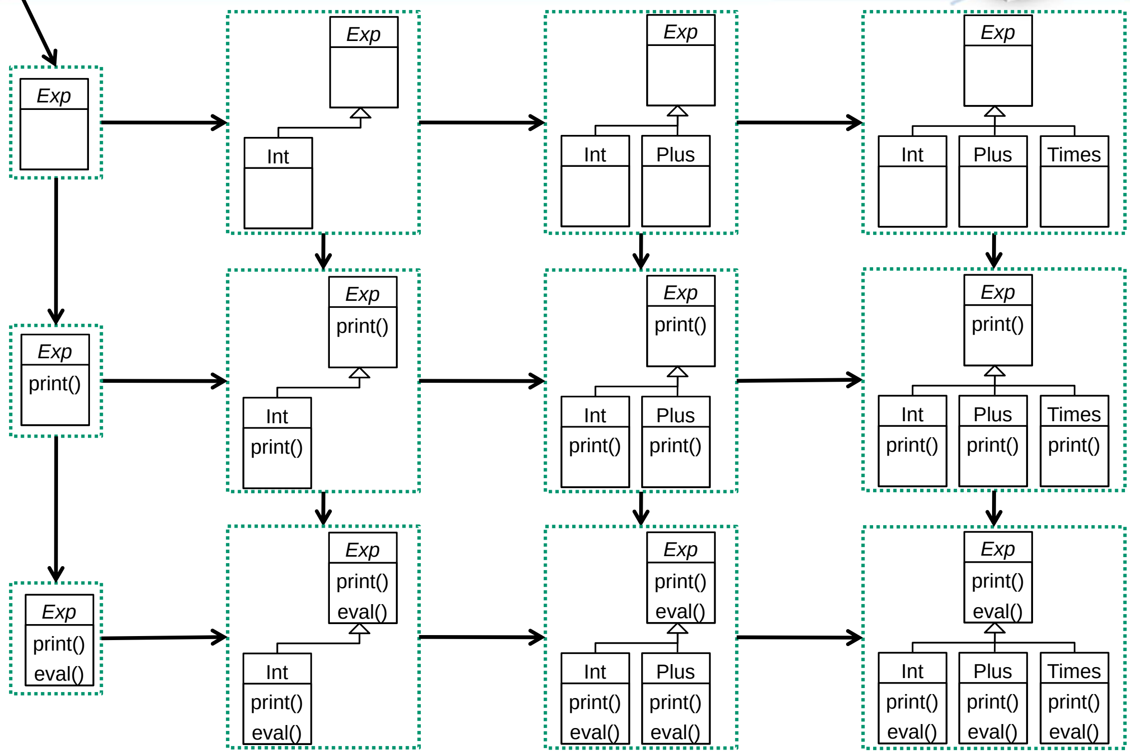
  - how ideas scale to synthesize large programs
  - SPL of 30-40K line programs (1000×)
  - how we synthesize the AHEAD Tool Suite


0
shell
Exp
Int
Plus
Times
print
print()
eval
eval()
ET


0
Exp
Int
Plus
Times
print()
exp
eval()
int
plus
times

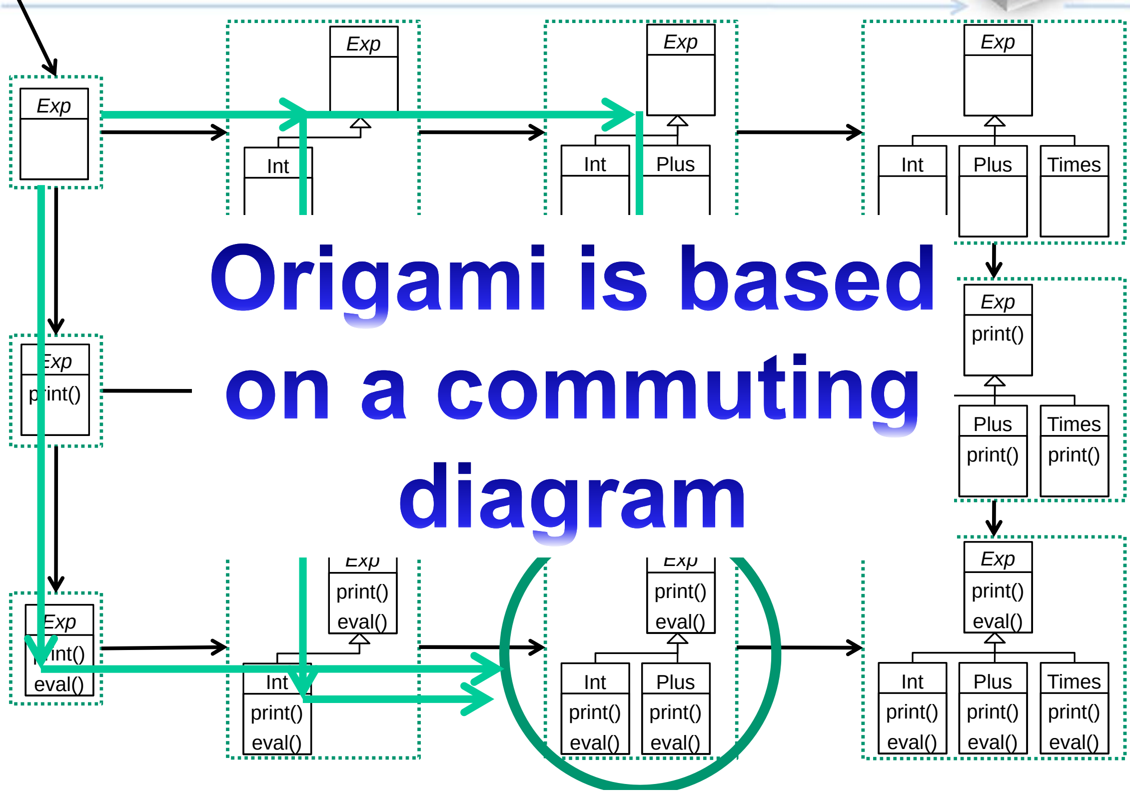
0

Expression Product Line (EPL)



0

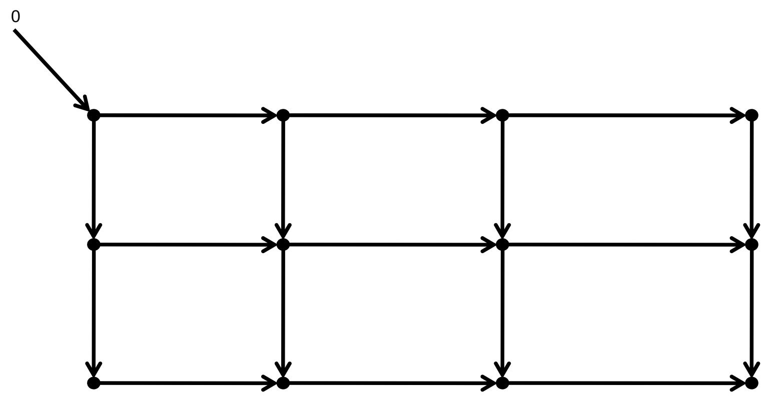
Commuting Diagrams in Categories



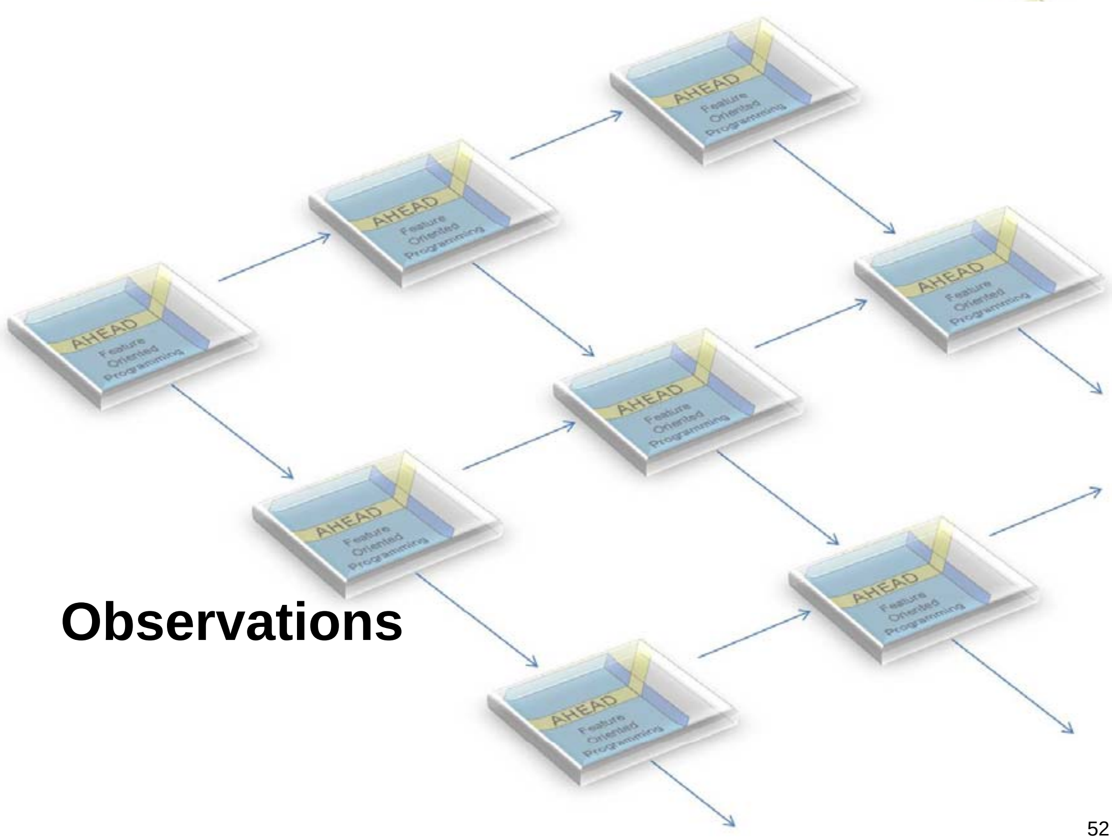
Origami is based on a commuting diagram

To Explain Origami, Few More Questions

- What is the EPL feature model?



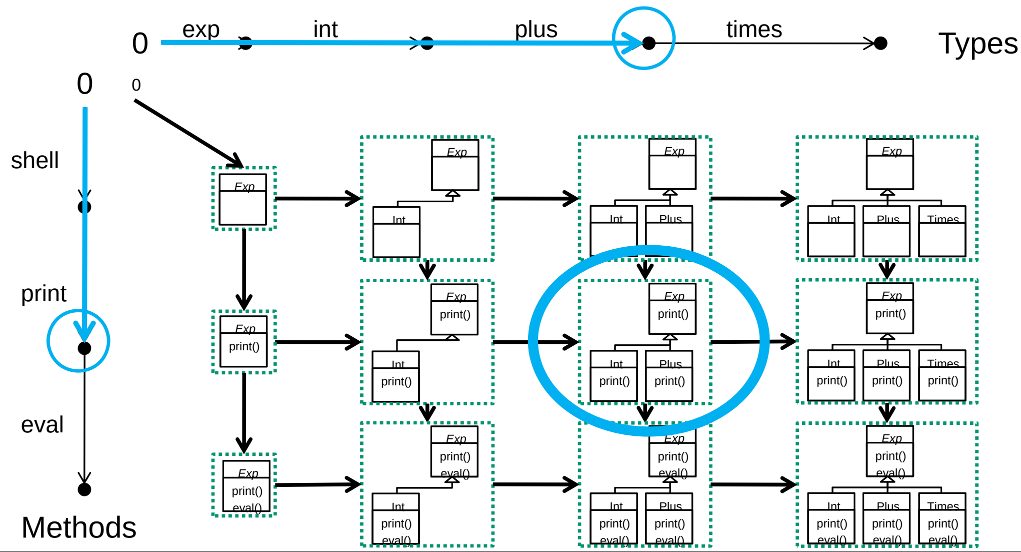
- What is the origin of the EPL graph?
- What arrows are stored?
- How is EPL related to kubes?



Observations

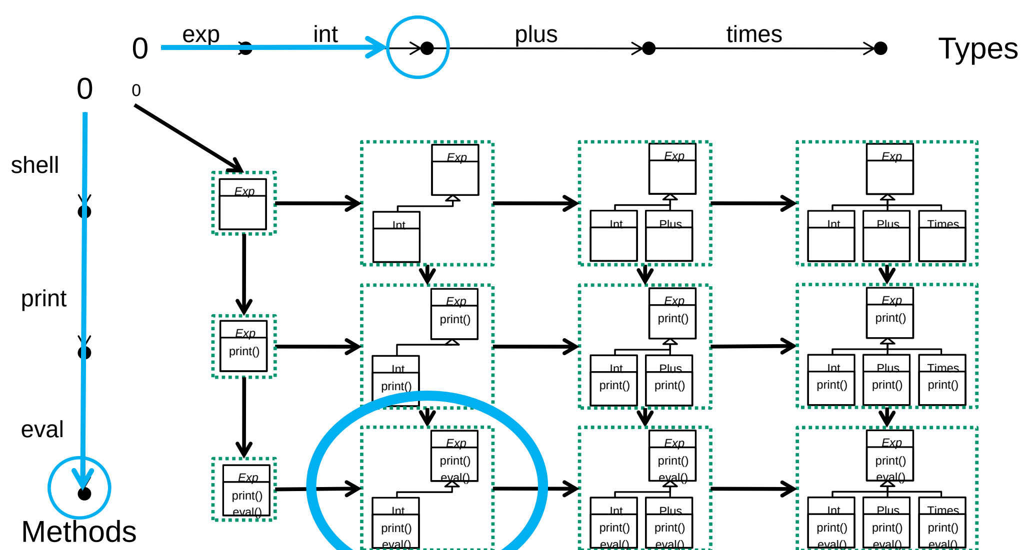
Observation

- Identify any program in EPL by a pair of axis timelines



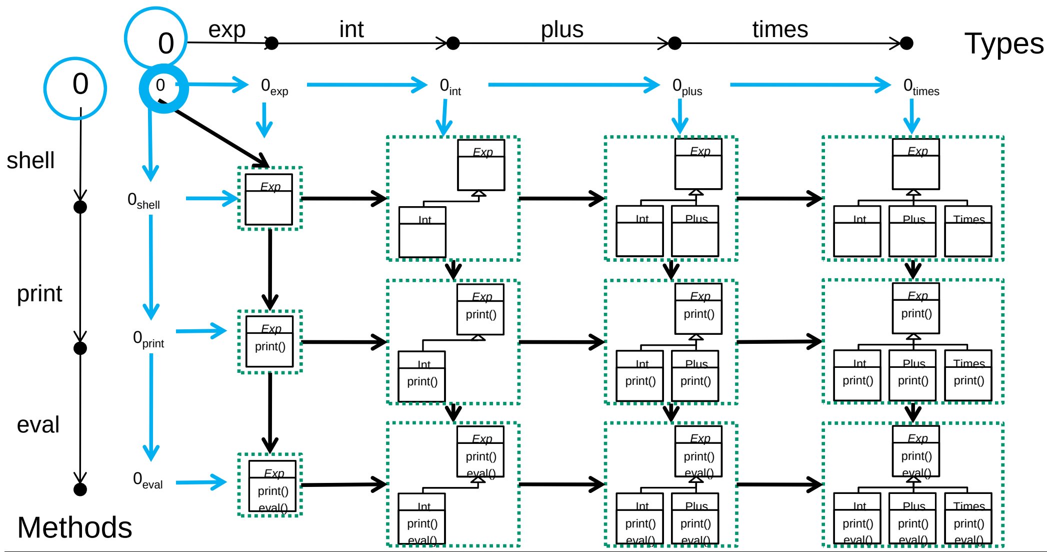
Observation

- Identify any program in EPL by a pair of axis timelines



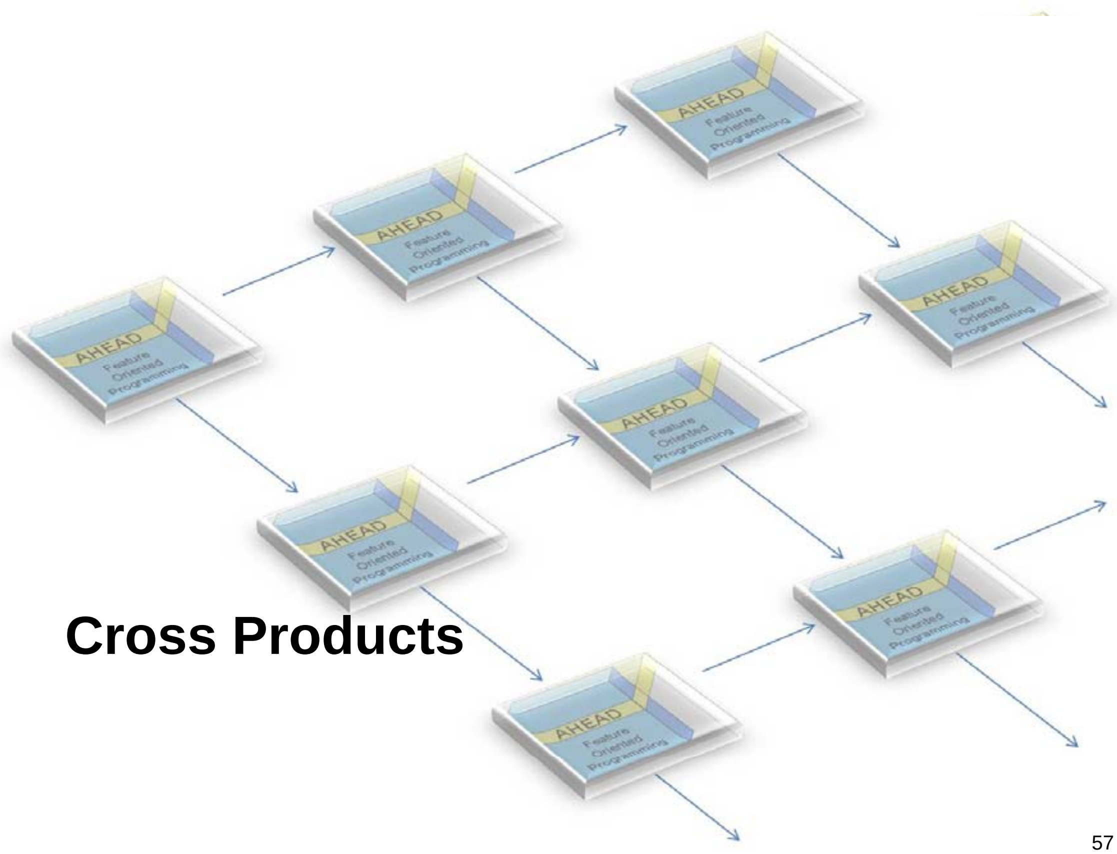
Boundary Cases

- Postulate extra null programs and arrows between them



Last Pieces to Puzzle

- Look at two seemingly unrelated topics
  - cross products of structures (SPLs)
  - feature interactions
- Putting them together explains Origami



Cross Products

Cross Products in Mathematics

- A common way to scale 1D structures to higher dimensions is by taking the **cross product** of 1D structures
  - 2D structure = cross product of a pair of 1D structures
  - nD structure = cross product of n 1D structures

- SPL structure is an ordered pair:

  SPL = (Grammar, Kube)

- Look at cross product of Methods and Types product lines

$$\text{Methods} \times \text{Types} = (G_{methods}, V_{methods}) \times (G_{types}, V_{types})$$

$$= (G_{methods} \times G_{types}, V_{methods} \# V_{types})$$

EPL Feature Model is…

- Union of Methods and Types feature models with extra rule

```
Methods: [eval] [print] shell;
%%
eval → print
```

×

```
Types : [times] [plus] [int] exp;
%%
plus → int
times → plus
```

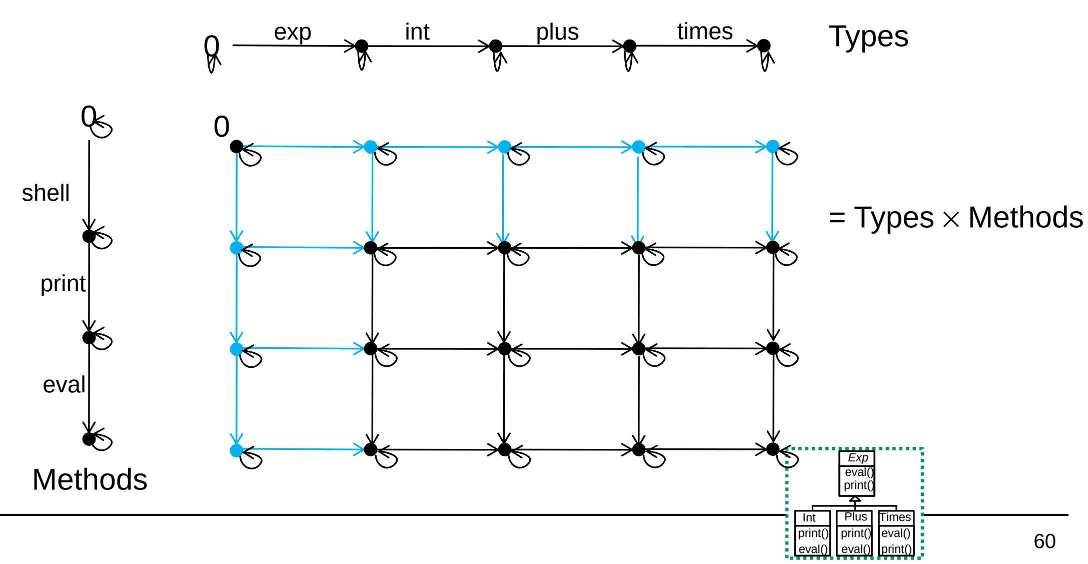
=

```
EPL : Methods Types ;

Methods: [eval] [print] shell;
Types : [times] [plus] [int] exp;

%%
eval → print
plus → int
times → plus
```

extra grammar rule

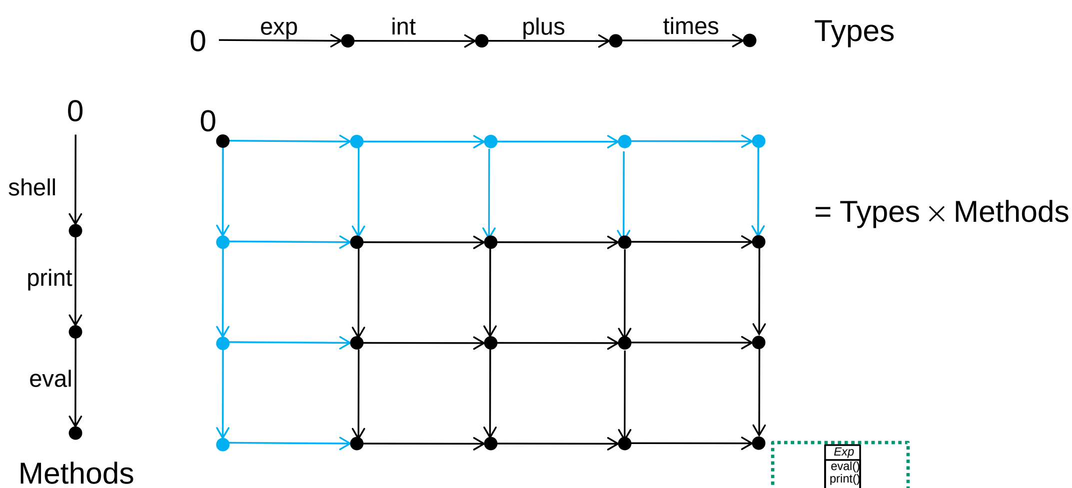
EPL Graph

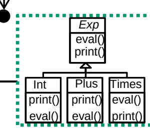
- **tensor product** (×) of Methods and Types graphs
  - shape we want
  - remember: programs/objects are computed, so too are the arrows



EPL Graph

- **tensor product** (×) of Methods and Types graphs
  - shape we want
  - remember: programs/objects are computed, so too are the arrows





Kube of EPL

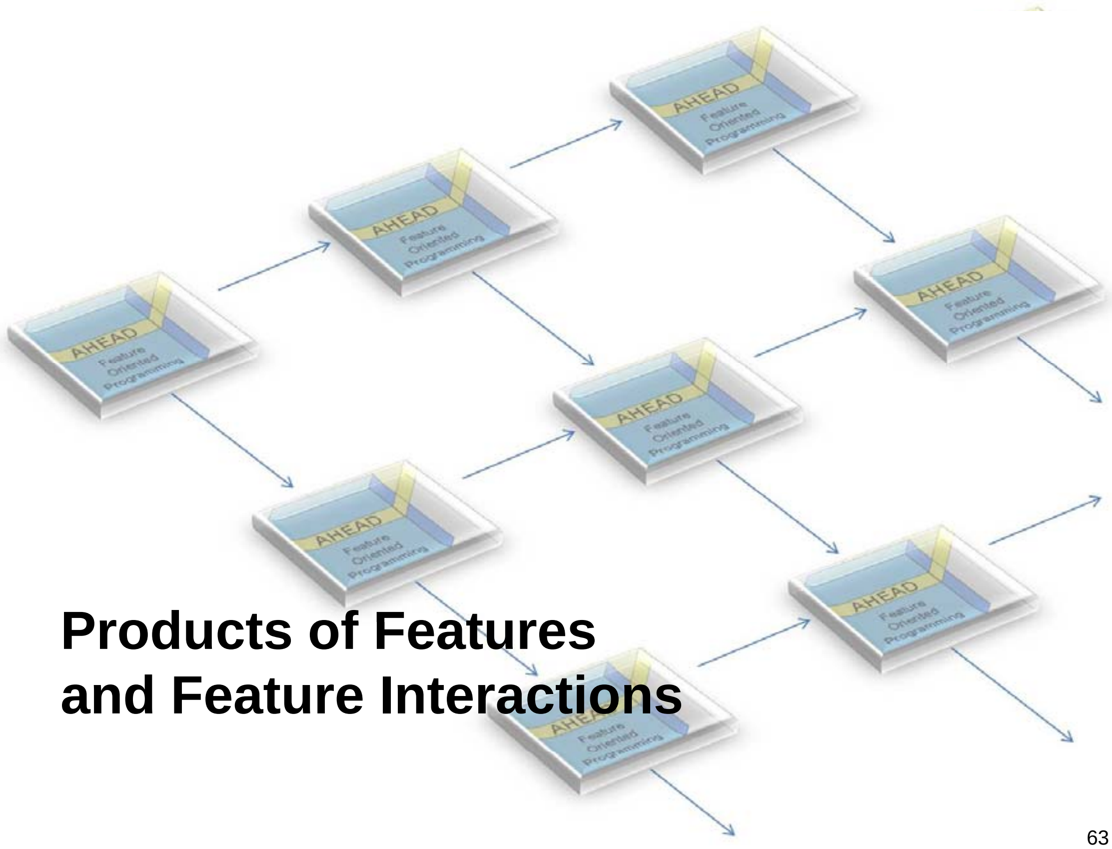
- **Interaction cross product** (#) of the Methods &Types kubes
  - another tensor product
  - GenVoca model of EPL is a matrix (2D kube)

$$EPL_{methods,types} = V_{methods} \# V_{types}$$

$$= [\text{ shell, print, eval }] \# [\text{ exp, int, plus, times }]$$

but what are these entries?

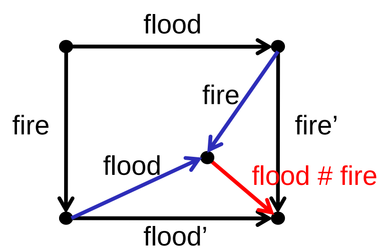
$$\begin{bmatrix} (\text{shell} \# \text{exp}) & (\text{shell} \# \text{int}) & (\text{shell} \# \text{plus}) & (\text{shell} \# \text{times}) \\ (\text{print} \# \text{exp}) & (\text{print} \# \text{int}) & (\text{print} \# \text{plus}) & (\text{print} \# \text{times}) \\ (\text{eval} \# \text{exp}) & (\text{eval} \# \text{int}) & (\text{eval} \# \text{plus}) & (\text{eval} \# \text{times}) \end{bmatrix}$$



Products of Features and Feature Interactions

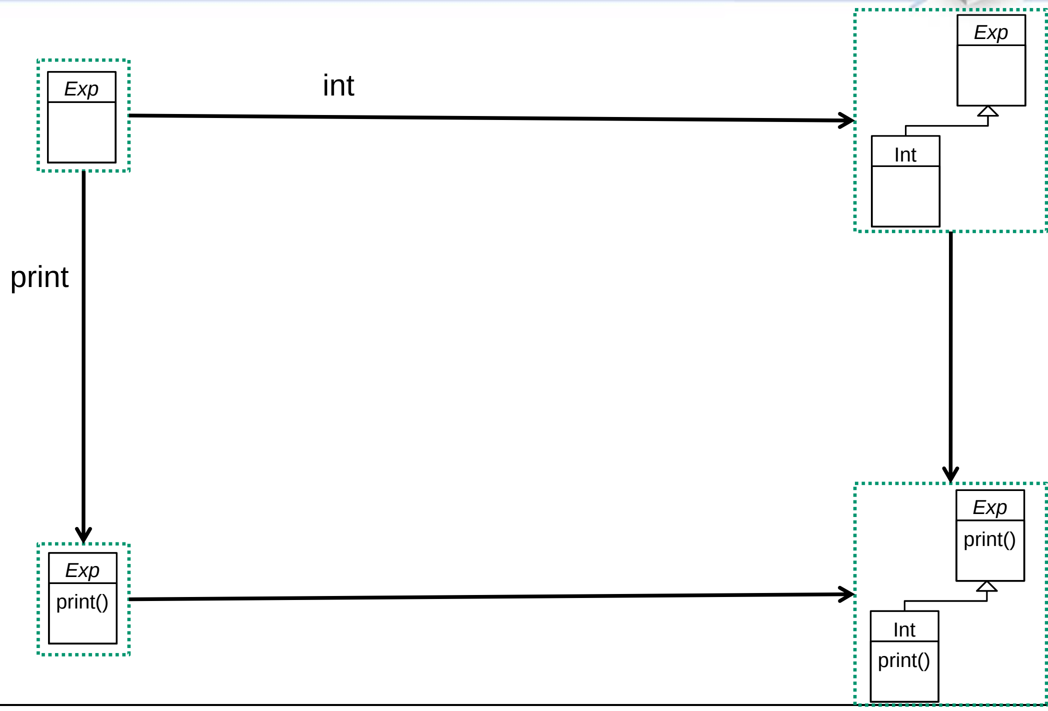
Products & Interactions

- Features change a design
- Structural feature interactions are changes to a design that are added conditionally based on the features that are present
- Classical example (Kyo Kang)
  - flood control, fire control
  - must add extra arrow → to control their interaction, i.e., so that they work together correctly
- Taking the “square” of features

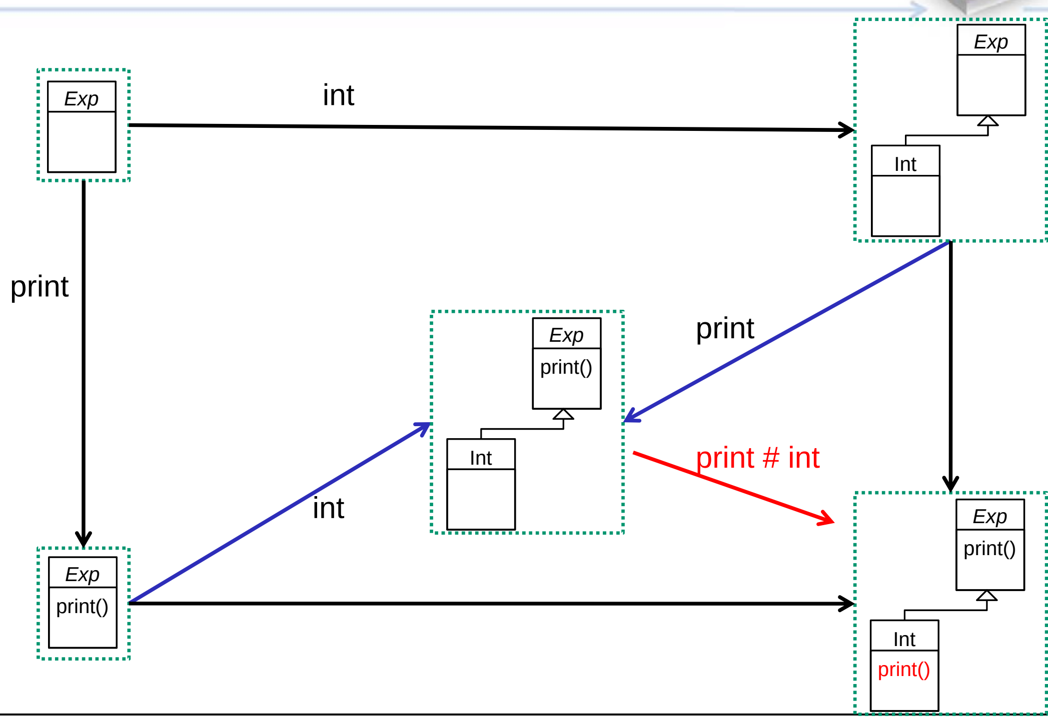


flood × fire = flood#fire + flood + fire

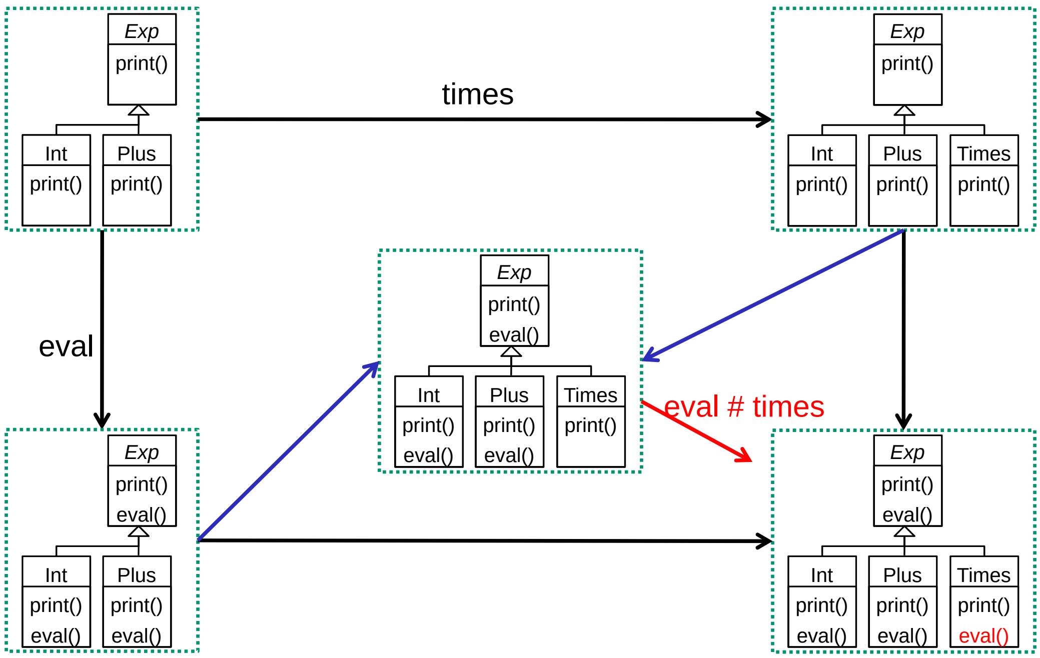
From EPL: int × print



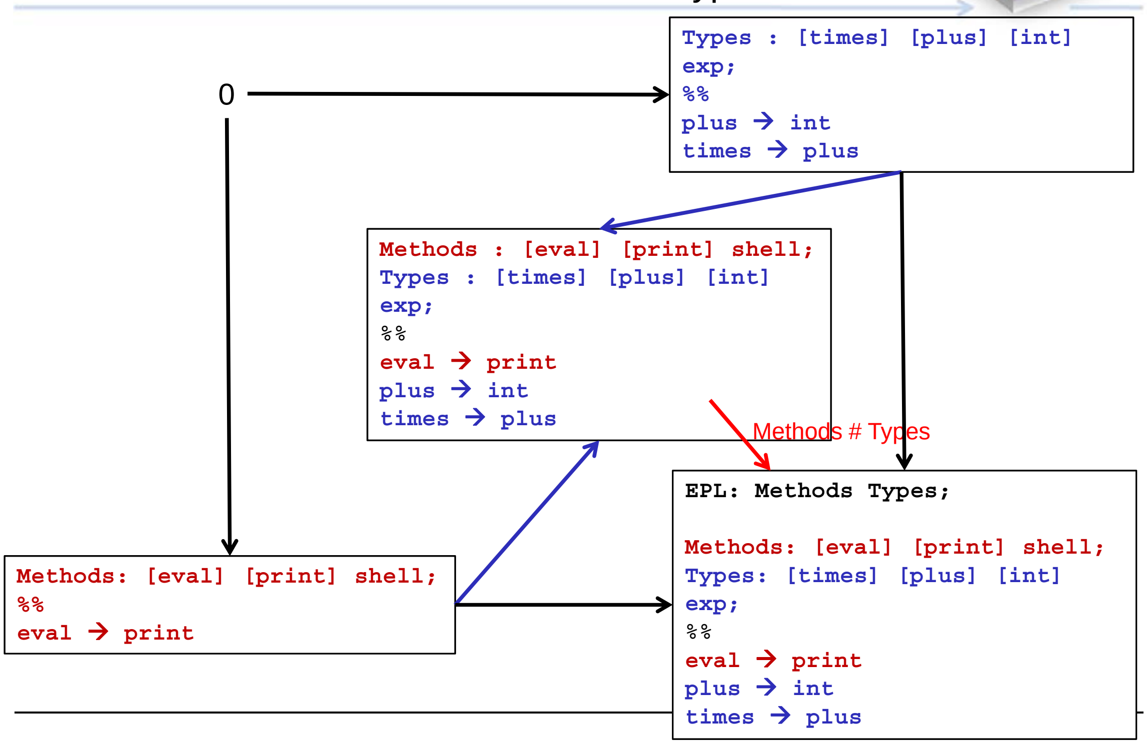
From EPL: int × print



From EPL: eval × times

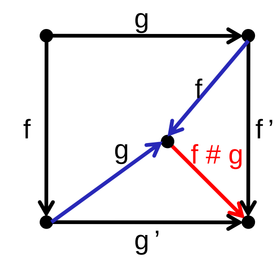


$G_{Methods} \times G_{Types}$



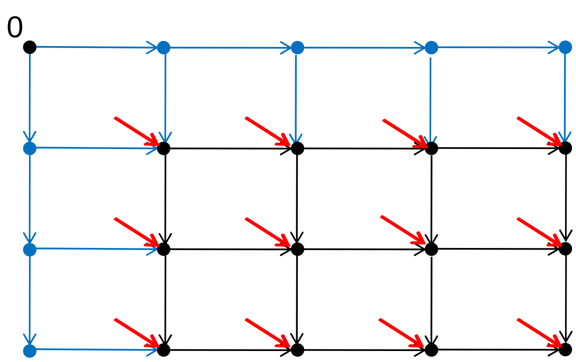
Taking the Square of Features

- Fundamental relationship among features & interactions
- The changes features make always commute
- Interactions are added to coordinate features correctly



Adding Interactions

- Recall the EPL graph
- Expose interaction arrows
- EPL Matrix = $V_{\text{Methods}} \# V_{\text{Types}}$

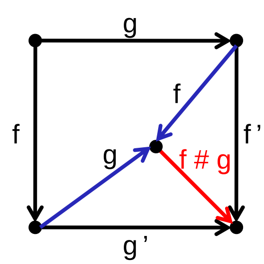


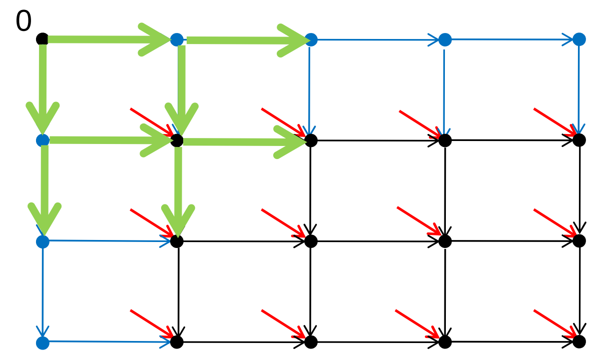
$$\begin{bmatrix} \text{(shell \# exp)} & \text{(shell \# int)} & \text{(shell \# plus)} & \text{(shell \# times)} \\ \text{(print \# exp)} & \text{(print \# int)} & \text{(print \# plus)} & \text{(print \# times)} \\ \text{(eval \# exp)} & \text{(eval \# int)} & \text{(eval \# plus)} & \text{(eval \# times)} \end{bmatrix}$$

Important Property

- Given the EPL matrix and boundaries,
  any arrow and any program of EPL can be computed

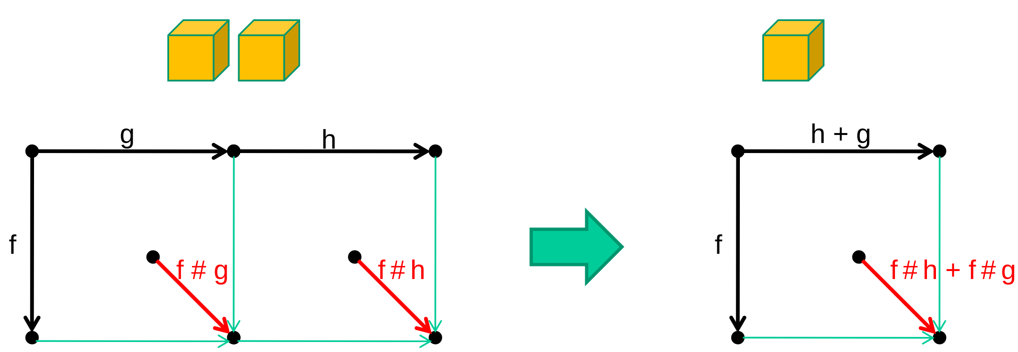
- Proof sketch
  - given f, g, f#g
  - can complete square



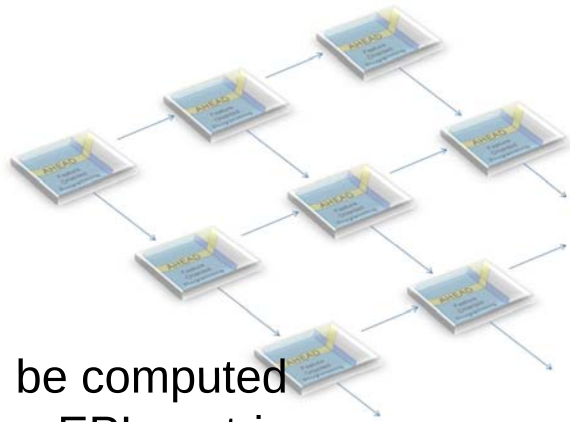


Another Important Property

- Aggregating adjacent squares sums interactions



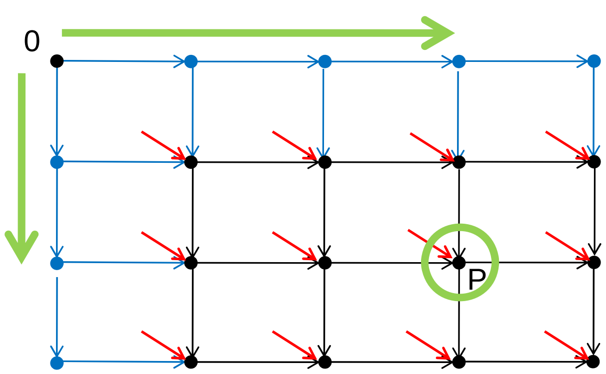
Finally, Why Origami Works



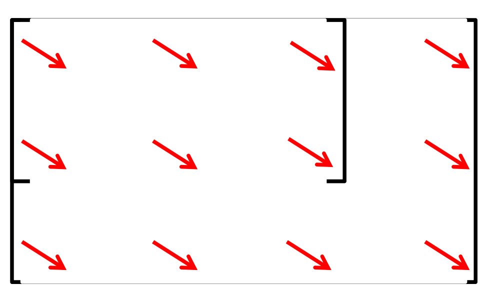
Any program of EPL can be computed
by projecting & contracting EPL matrix

Projection

- Projection of matrix = projection of category, yields a commuting diagram
  - any path from 0 to P will synthesize P



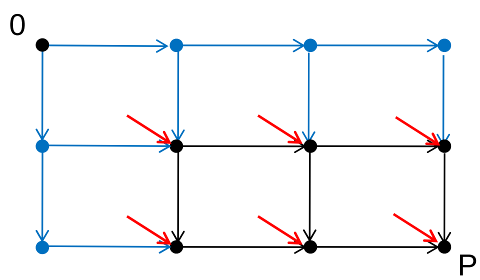
commuting diagram

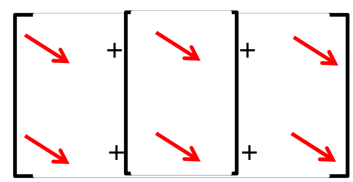


matrix

Contract by Columns

- Contraction aggregates adjacent squares and sums interaction arrows



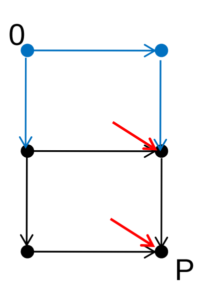


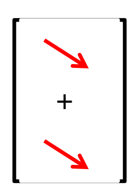
commuting diagram

vector

Contract by Rows

- Contraction aggregates adjacent squares and sums interaction arrows



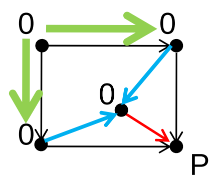


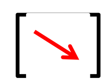
commuting diagram

scalar

Final Step/Square

- Boundary programs are 0s
- 0→0 arrows add NOTHING, and so too their compositions
  - interaction arrow defines an expression that synthesizes P
  - this is the arrow that is computed by matrix contraction



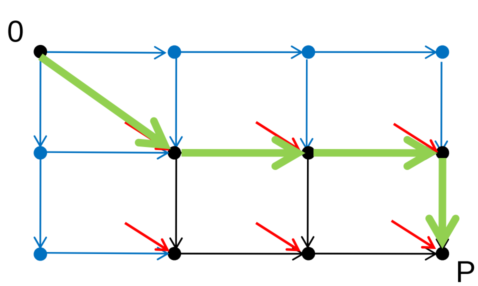


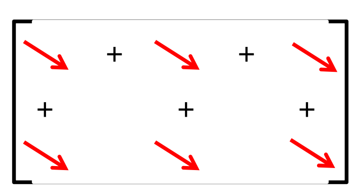
commuting diagram

scalar

Aside: Commuting Paths

- Start from desired program
- Aggregate by row moves vertically up
- Aggregation by columns moves vertically left
- Final square any path will do





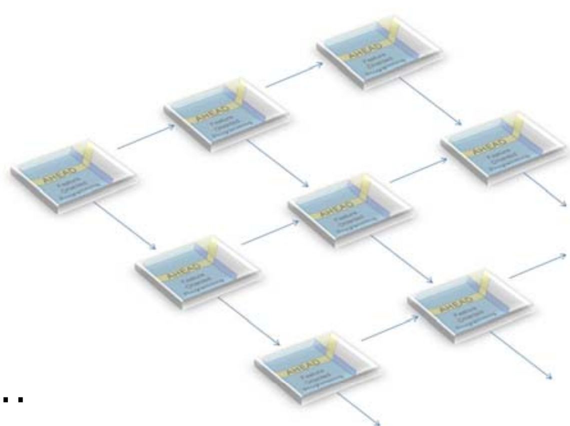
commuting diagram

matrix

Generality of Arguments

- Nothing special about how we contracted the matrix
  - any contraction (i.e., path) would do
- Nothing special about program P we selected
  - any program in the SPL would do
- Nothing special about this matrix
  - any SPL matrix would do – ex. AHEAD 2D kube
- Nothing special about 2D kubes
  - same ideas apply to higher-dimensions
  - e.g. a square becomes a cube with a single interaction arrow
- Result that can be applied to SPL in general
  - Origami is a fundamental structure of SPLs

So What?



and other final thoughts…

Perspective

- **Programming in the Small** – Java and C# objects, methods
- **Programming in the Large** – objects are programs, methods are xforms
- Transformations provide a fundamental way to understand SWD
  - foundation for MDE, refactorings, and software product lines
- Working toward a **Science of Automated Design**
  - start from experiences and abstract to a theory
  - belief: few principles hold this universe together
  - principles assume a mathematical form as in other sciences and engineering disciplines that manipulate structures
  - a promising alternative to the ad hoc design techniques in use today

Dimensions of Variability

- Preplanned variability is key to automated software design
  - much like design patterns helped novices design like experts
  - SPLs help novices create customized programs like experts
- Common in SPLs to have orthogonal and interacting sets of features
  - EPL – method variability vs. type variability
  - AHEAD – tool variability vs. language variability
- Origami expresses multiple dimensions of variability
  - powerful and elegant, it scales
- Expose new and basic relationships in mathematical form
  - projection and contraction of kubes – database technology
  - cross product of features (feature interactions)
  - cross product of SPLs
  - use of n-D kubes to represent n-dimensions of variability

Final Comments

- Clear that ideas are being reinvented in different contexts
  - not accidental – evidence we are working toward general paradigm
  - modern mathematics is a simple language to express these ideas
  - maybe others may be able to find deeper connections
- At the earliest stages
- Advice: think in terms of arrows, think in terms of kubes
  - if you look for kubes, you'll find them…
  - if you don't look, you won't find them…

Look for them!